

Excelsior JET

for Linux Operating System

Version 7.6



User's Guide



<http://www.excelsior-usa.com>

Copyright © 1999-2011 Excelsior LLC. All rights reserved.

Information in this document is subject to change without notice and does not represent a commitment on the part of Excelsior, LLC.

Excelsior's software and documentation have been tested and reviewed. Nevertheless, Excelsior makes no warranty or representation, either express or implied, with respect to the software and documentation included with Excelsior product. In no event will Excelsior be liable for direct, indirect, special, incidental or consequential damages resulting from any defect in the software or documentation included with this product. In particular, Excelsior shall have no liability for any programs or data used with this product, including the cost of recovering programs or data.

Excelsior JET is trademark of Excelsior LLC.

Java (tm) and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

All trademarks and copyrights mentioned in this documentation are the property of their respective holders.

Contents

Preface	1
About this manual	1
Conventions used in this manual	2
Language descriptions	2
Source code fragments	2
Customer Support	3
1 Overview	5
1.1 Excelsior JET at a glance	5
1.2 Established features	6
1.2.1 JET Optimizer	6
1.2.2 JET Runtime	6
1.2.3 Installation Toolkit	6
1.3 What is new in Excelsior JET 7.6	7
2 Installation and setup	9
2.1 System requirements	10
2.2 Installation	12
2.3 Profiles	12
2.4 Updates	15
2.4.1 Update installation	15
2.4.2 Update management	16
2.4.3 Cleaning up backed up configurations	16
2.5 JET Setup options summary	17
2.6 Reproducing the build environment	18
3 Excelsior JET Launcher	19
3.1 Creating a new project	19
3.2 Opening an existing project	20
4 JET Control Panel	21
4.1 Before you begin	21
4.2 Starting Excelsior JET Control Panel	21
4.3 JET Control Panel overview	22
4.4 Working with projects	23
4.4.1 Creating a new project	23
4.4.2 Opening a project	23
4.4.3 Saving a project	24
4.4.4 Saving a project under a different name	24

4.4.5	Closing a project	24
4.5	Step 1: Describing an application	24
4.5.1	Description form	24
4.5.2	Manual setting	26
4.6	Step 2: Performing a test run	27
4.6.1	Why is it?	28
4.7	Step 3: Selecting a compilation mode	28
4.7.1	Optimization presets	28
4.7.2	Classpath grid	29
4.7.3	Global Optimizer	31
4.7.4	Inconsistency warnings	31
4.7.5	Class viewer	32
4.8	Step 4: Setting options	36
4.8.1	Optimizations	36
4.8.2	Memory Management	37
4.8.3	JIT compiler	38
4.8.4	Advanced settings	39
4.9	Step 5: Specifying target settings	40
4.9.1	Executable settings	40
4.9.2	Runtime selection	41
4.9.3	Creating trial versions	41
4.9.4	Stack trace support	41
4.9.5	Multi-app executable	41
4.10	Step 6: Selecting application appearance	41
4.10.1	Splash screen	42
4.11	Step 7: Compiling your application	42
4.11.1	Configuring Startup Accelerator	42
4.11.2	Building the executable	43
4.11.3	Running compiled application	43
4.11.4	Creating build script	43
4.11.5	Packaging your application	44
4.11.6	Fixing project inconsistencies	44
4.11.7	Correcting build errors	45
5	Intellectual property protection	47
5.1	Code protection	47
5.1.1	Enabling code protection	48
5.2	Data protection	48
5.2.1	Enabling data protection	49
6	Deployment automation	51
6.1	Intro	51
6.2	Excelsior deployment technology	51
6.2.1	JetPackII	51
6.2.2	Excelsior Installation Toolkit	52
6.3	Before You Begin	53
6.4	Starting JetPackII	53
6.5	JetPackII Overview	54

6.6	Working With Projects	55
6.6.1	Creating a new project	55
6.6.2	Opening a project	55
6.6.3	Saving a project	56
6.6.4	Saving a project under a different name	56
6.6.5	Closing a project	56
6.7	Package Files Panel	56
6.7.1	System files	57
6.7.2	Controls	57
6.8	Selecting installation type	58
6.8.1	Creating a new installation	58
6.8.2	Creating an update	58
6.9	Step 1: Adding files to package	59
6.9.1	Adding files using filechooser	59
6.9.2	Adding files using the Difference Tracker	60
6.9.3	Packaging JIT cache	61
6.9.4	Caching of JITed classes on end user system	61
6.10	Step 2: Configuring Resources And System Properties	62
6.10.1	Setting the classpath entries	63
6.10.2	Editing Java system properties	64
6.11	Step 3: Configuring the JET Runtime	65
6.11.1	Selecting optional components	65
6.11.2	Selecting detached components	67
6.11.3	Enabling disk footprint reduction	68
6.12	Step 4: Performing a Trial Run	69
6.12.1	Selecting a trial run directory	70
6.12.2	Working with trial run configurations	70
6.12.3	Configuring a trial run	70
6.12.4	Performing all trial runs	71
6.12.5	Troubleshooting	71
6.12.6	Skipping trial runs	71
6.13	Step 5: Selecting a Back-End	71
6.13.1	Excelsior Installer setup	72
6.13.2	Self-contained directory	72
6.13.3	Installation into multiple directories	73
6.14	Step 6: Specifying Installation Settings	73
6.14.1	Vendor information	73
6.14.2	Installation directory	74
6.14.3	Installer appearance	75
6.14.4	Extra facilities	75
6.15	Step 7: Specifying Installation Settings Of Windows Services	76
6.16	Step 8: Creating The Installation Package	76
6.17	Packaging native method libraries	77
6.18	Installations protected by license managers	78
6.19	Relocation of JetPackII projects	79
6.20	Updating JET Runtime	80
6.20.1	When a JET Runtime update is necessary	80
6.20.2	JET Runtime Update dialog	81

6.21	Deployment Utilities	81
6.21.1	xpack	81
6.21.2	xbind	82
6.22	Excelsior Installer	82
6.22.1	Installer	82
6.22.2	Uninstaller	84
6.22.3	Callback shared libraries	84
7	Eclipse RCP	87
7.1	Overview	87
7.1.1	The main functionality	88
7.1.2	Extra features	88
7.1.3	Automated builds	88
7.2	Compilation	89
7.2.1	Step 1. Describing RCP application	89
7.2.2	Step 2. Test Run for RCP application	89
7.2.3	Step 3. Settings for OSGi bundles	90
7.2.4	Step 4. Optimizer and Runtime options	90
7.2.5	Step 5. Settings for the RCP executable	90
7.2.6	Step 6. Application appearance	91
7.2.7	Step 7. Building the executable	91
7.3	Packaging for deployment	91
7.3.1	Files to package	91
7.3.2	Configuring the project	92
7.3.3	Building the package	92
7.4	Known limitations	92
7.5	FAQ	92
7.6	Troubleshooting	93
8	Tomcat Web applications	95
8.1	Overview	95
8.1.1	The main functionality	96
8.1.2	Extra features	96
8.1.3	Automated builds	96
8.2	Compilation	97
8.2.1	Step 1. Describing Web applications	97
8.2.2	Step 2. Test Run for Web applications	98
8.2.3	Step 3. Settings for Web applications	98
8.2.4	Step 4. Optimizer and Runtime options	99
8.2.5	Step 5. Settings for the Tomcat executable	99
8.2.6	Step 6. Application appearance	100
8.2.7	Step 7. Building the executable	100
8.3	Packaging for distribution	100
8.3.1	Files to package	100
8.3.2	Configuring the project	101
8.3.3	Building the package	101
8.4	Known limitations	102
8.5	FAQ	102

8.6	Throubleshooting	102
9	Java Runtime Slim-Down	105
9.1	Introduction	105
9.2	Java SE components	106
9.2.1	What is a detachable Java SE component	106
9.2.2	Used and unused Java SE components	106
9.2.3	How to identify unused components	107
9.3	Deployment	107
9.4	Execution	108
9.5	Runtime Slim-Down HOWTO	108
10	Application considerations	111
10.1	Implementation-specific differences between JVMs	111
10.2	Runtime Selection	114
10.2.1	Available runtimes	114
10.2.2	Which runtime fits best for my application?	115
10.2.3	How to enable a particular runtime	116
10.3	Code performance	116
10.4	Scalability	117
10.4.1	Multiprocessor systems	117
10.4.2	Number of threads	117
10.5	Java system properties	118
10.5.1	How to set system properties	118
10.5.2	Standard runtime options	119
10.5.3	JET Runtime-specific properties	120
10.6	Memory management	121
10.6.1	GC ratio	121
10.6.2	Maximum heap size	122
10.6.3	Handling low memory	123
10.6.4	Large heaps	124
10.6.5	Parallel hardware	124
10.7	Multi-app executables	125
10.7.1	Command-line syntax	125
10.7.2	Default main class	126
10.7.3	Splash screen	126
10.8	Disk footprint	127
10.8.1	Medium reduction	127
10.8.2	High reduction	128
10.8.3	Complementary methods	128
10.8.4	On-demand decompression	128
10.9	Creating trial versions	129
10.10	Application appearance	130
10.10.1	Splash screen	130
10.11	Stack trace	133
10.11.1	How to enable stack trace	133
10.11.2	How to disable stack trace	133
10.12	Assertions	133

10.13	Resource packing	134
10.13.1	Resource packing modes	134
10.13.2	Setting resource packing options	135
10.14	Character encodings	135
10.15	Bytecode consistency checks	135
10.16	Baseline compilation	137
10.17	Customizing Runtime error messages	137
11	Mixed Compilation Model	139
11.1	Overview	139
11.2	Quick start	140
11.3	When the mixed model operates	140
11.3.1	When JET Runtime uses MCM	140
11.3.2	When you may use MCM	140
11.4	Using the mixed model	141
11.4.1	JIT compiler selection	141
11.4.2	Caching JIT'ed code	142
11.5	Reflective shield	142
11.6	JIT cache optimization	143
11.6.1	Challenges	143
11.6.2	The solution	144
11.6.3	How to optimize a JIT cache	144
11.6.4	Incremental fusion	145
11.7	xjava launcher	146
11.7.1	Synopsis	146
11.7.2	Options	147
11.7.3	JIT cache optimization with xjava	149
11.7.4	Usage example	149
11.8	JIT compilation and code protection	149
12	Dynamic linking	151
12.1	Intro	151
12.2	Invocation shared libraries	152
12.3	Multi-component applications	152
12.3.1	Building multi-component applications	153
12.3.2	Using multiple components	156
13	Global Optimizer	159
13.1	Introduction	159
13.2	Application domain	160
13.3	How it works	160
13.3.1	Global optimizations	160
13.3.2	Code mixing	161
13.3.3	Dynamic compilation	161
13.3.4	Code protection	161
13.4	Using the Global Optimizer	162
13.4.1	Why Test Run is necessary	162
13.4.2	Performing a Test Run	162

13.4.3	Enabling the Global Optimizer	163
13.5	Deployment	164
14	Startup time optimization	165
14.1	Introduction	165
14.1.1	Startup Optimization Toolkit	165
14.1.2	Application considerations	166
14.2	Enabling startup optimizations	166
14.2.1	How to turn the Startup Optimizer on	166
14.2.2	How to turn the Startup Accelerator on	167
14.3	Automated builds	167
14.3.1	Startup Optimizer in automated builds	167
14.3.2	Startup Accelerator in automated builds	168
15	Troubleshooting	171
15.1	Compilation	171
15.2	Execution	172
15.3	Deployment	173
16	JC reference	175
16.1	Configuring the compiler	175
16.1.1	System search paths	175
16.1.2	Portable filename notation	175
16.1.3	Working configuration	176
16.1.4	Directory hierarchies	176
16.1.5	Redirection file	176
16.1.6	Options	178
16.1.7	Configuration file	178
16.1.8	Compiler memory usage	179
16.2	Invoking the compiler	179
16.3	Precedence of compiler options	180
16.4	Compiler operation modes	180
16.4.1	MAKE mode	181
16.4.2	PROJECT mode	181
16.4.3	ALL submode	181
16.4.4	OPTIONS submode	182
16.4.5	EQUATIONS submode	182
16.4.6	HELP mode	182
16.5	Project Data Base	182
16.6	Make strategy	183
16.7	Control file preprocessing	183
16.8	Project files	184
16.8.1	The !classpathentry directive	187
16.8.2	The !classloaderentry directive	189
16.8.3	The !module directive	189
16.8.4	The !batch directive	189
16.8.5	The !uses directive	190
16.9	Compiler options and equations	190

16.9.1	Options	190
16.9.2	Options reference	191
16.9.3	Equations	195
16.9.4	Equations reference	195
16.9.5	Internal options and equations	201
16.10	Compiler messages	202
16.10.1	Java bytecode front-end errors and warnings	202
16.10.2	Project system errors	202
16.10.3	Symbol files read/write errors	205
16.10.4	Warnings	206
16.10.5	Native back-end warnings	208
16.10.6	Native back-end errors	208
16.10.7	Internal errors	209
A	Customizing the compiler	211
A.1	Customizing compiler messages	211
A.2	Error message format specification	211
A.3	Regular expression	212
A.4	Template files	213
A.4.1	Using equation values	213
A.4.2	File name construction	214
A.4.3	Iterators	214
A.4.4	Examples	215

Preface

About this manual

Excelsior JET is a toolkit and complete runtime environment for optimizing, deploying and running applications written in the Java programming language. JET is a powerful solution that enables your desktop and server side Java applications to take full advantage of the underlying hardware. Moreover, JET effectively protects your intellectual property by making your programs as hard to reverse engineer as if they were written in C++. Auxiliary tools that come with Excelsior JET, simplify deployment of your Java applications to target systems.

This manual is both a reference and learning guide. Its chapters are as follows:

Chapter 1 is a general introduction to Excelsior JET that explains some of the basic features.

Chapter 2 describes JET installation and setup procedures.

Chapter 4 provides detailed usage instructions for the JET Control Panel, a graphical wizard to the command-line compiler.

Chapter 16 covers command-line compiler usage in detail: operation modes, configuration and project files, options, error messages and warnings, etc.

Chapter 10 discusses fine tuning and running your applications. This includes information on adjusting a compiled application's performance and memory consumption, setting system properties, resource packing, and the like.

Chapter 12 is a discussion of JET support for the creation of dynamic link libraries.

Chapter 11 contains an explanation of the Mixed Compilation Model, which lets JET-compiled applications to load and run Java class files that were not precompiled.

Chapter 6 describes deployment tools and utilities that enable distribution of compiled applications to other computers.

Chapter 15 is a collection of solutions to common problems.

Appendix A contains information of interest to advanced users. Topic covered include messages customization, complete regular expression syntax, and template files.

Conventions used in this manual

Language descriptions

Where formal descriptions for language syntax constructions appear, an extended Backus-Naur Formalism (EBNF) is used.

These descriptions are set in the `Courier` font.

```
Text = Text [ { Text } ] | Text .
```

In EBNF, brackets "[" and "]" denote optionality of the enclosed expression, braces "{" and "}" denote repetition (possibly 0 times), and the vertical line "|" separates mutually exclusive variants.

Non-terminal symbols start with an upper case letter (`Statement`). Terminal symbols either start with a lower case letter (`ident`), or are written in all upper case letters (`OPTION`), or are enclosed within quotation marks (e.g. `' = '`).

Source code fragments

When fragments of a source code are used for examples or appear within a text they are set in the `Courier` font.

```
public class hello
{
    static public void main (String args[])
    {
        System.out.println("Hello world!");
    }
}
```

Customer Support

In case you experience technical problems using an Excelsior product, please contact Excelsior Technical Support Department at:

Excelsior, LLC
6 Lavrenteva Ave.
630090 Novosibirsk RUSSIA

Tel: +7 (383) 330 55 08 (6 AM to 3PM GMT)

Fax: +1 (509) 271 5205

Email: support@excelsior-usa.com

Web: <http://www.excelsior-usa.com/jetsupport.html>

Chapter 1

Overview

1.1 Excelsior JET at a glance

Excelsior JET is a toolkit and complete runtime environment for optimizing, deploying and running applications written in the Java™ programming language. The product package includes Excelsior JET Optimizer, Excelsior JET Runtime and Excelsior JET Installation Toolkit.

The JET Optimizer enables you to convert your application's components (classes, JAR and WAR files, OSGi bundles) into optimized x86 (IA-32) code on the developer's system. As a result, you get high performance native executables for Microsoft Windows or Linux. The produced native code is highly resistant to reverse engineering and tampering.

The JET Runtime includes the Java SE API licensed from Oracle and provides all low-level Java routines which the optimized executables need to run such as garbage collection.

Excelsior JET Optimizer and Runtime are certified for compliance with the specification of the Java platform, Standard Edition (Java SE), version 6.

If you want to distribute your application to a number of clients, you use the Installation Toolkit to prepare an installation package including your optimized Java application and the JET Runtime engine. Your clients simply install the package and start using your application. Additional software installation, such as the Java Runtime Environment (JRE), *is not required* on target systems.

1.2 Established features

1.2.1 JET Optimizer

The JET Optimizer transforms Java bytecode into high performance native code on the developer's system *before* program execution. The resulting optimized executables run directly on the hardware.

The Optimizer employs various optimization techniques commonly used in traditional C/C++ or FORTRAN compilers. To resolve Java performance challenges, the JET Optimizer features state-of-the-art optimizations such as inlining virtual calls, safe removal of ubiquitous synchronization, allocating objects on the stack and others.

The Optimizer comes with the JET Control Panel, a graphical tool that lets you quickly set up a project file and convert your application into a highly optimized executable (see Chapter 4.)

1.2.2 JET Runtime

The JET Runtime is a complete Java Virtual Machine (JVM). In particular, it includes a just-in-time (JIT) compiler to load and run Java class files. The key difference from other JVMs is that the JET Runtime can work with *both* Java classes and native executables produced by the JET Optimizer. For example, a JET-compiled application will be able to load Java plug-ins supplied in the form of jar files.

If your program uses plug-ins, loading them with the JIT compiler may take some time. To reduce the overhead, you may enable *JIT caching* in the JET Runtime. As a result, the native code produced by the JIT compiler will be retained in the JIT cache and reused on subsequent launches of the application. See Chapter 11 for details.

1.2.3 Installation Toolkit

To facilitate rapid creation of installation packages, Excelsior JET includes JetPackII, yet another graphical wizard. Using it, you prepare an installation package including your optimized Java application and the JET Runtime.

The deployment procedure is as simple as possible. You specify the JET-compiled application(s) and other files to be deployed, edit few settings, and the JetPackII generates a self-extracting installation executable, automatically including the JET runtime.

The installation executable is powered by the bundled Excelsior Installer that features both unattended and interactive installations.

In the interactive mode, the installer guides the users through the installation process allowing them to specify installation settings as usual. You can customize the installer to show end-user license agreement and to display installation interface in the English, German, French, Italian, Brazilian Portuguese, Spanish, Polish, Russian or Japanese languages, depending on your choice or the system locale settings.

JetPackII can also prepare the resulting package in the form of self-contained directory

that can be *simply copied* to target systems. It contains the clickable application's executable that requires neither extra environment settings, such as PATH, nor additional software installation, such as JRE. You can easily integrate such package into any setup generator or run the application off a USB flash drive without installation.

To learn more about the JetPackII and Excelsior Installer tools, please refer to Chapter 6.

1.3 What is new in Excelsior JET 7.6

This release enhances application performance, improves stability of compiled applications, and supports newer versions of Eclipse RCP and Apache Tomcat.

Application performance improvements

Excelsior JET 7.6 improves performance of the optimized applications, especially those running on multi-core/multi-CPU systems. In our results, the performance increase ranged from **1.2x** to **2.6x** on certain benchmarks.

Stability improvements

In the course of the JET Runtime redesign, a few issues that caused

- sporadic access violations (TRAP#3) in long-running applications
- unexpected `OutOfMemoryError` throwing

have been now fixed.

Alignment with HotSpot behavior

Some peculiarities of JVM behavior have always been *implementation-dependent*: they have not been covered by the Java SE platform specification, nor by the Java Compatibility Kit test suite. Formally, a VM implementation can behave differently from the Oracle VM provided the exact behavior is not enforced by the specification. In practice, however, that may provoke issues in (widely used) Java components/libraries that were not coded strictly to the Java specification and rely on HotSpot-specific features. This is why we have aligned our implementation with HotSpot in the following aspects:

- the order of elements in arrays returned by the methods `Class.getFields()` and `Class.getDeclaredFields()` is now the same
- JNI functions now continue working even if there is a pending exception in the thread (in the previous versions of the JET Runtime they could have immediately returned)
- When getting JNI IDs, the lookup order of class members including those declared in superclasses/superinterfaces is now the same

Installation Toolkit enhancements

- For Tomcat-based applications, you may now configure Excelsior Installer to prompt the end user to change the default HTTP port for the Web Server during installation. See [8.3.2](#) for more information.

Other improvements

- **Eclipse RCP**
Support for Eclipse RCP 3.7 (Indigo) is included.
- **Apache Tomcat**
Support for Apache Tomcat 7.0 is added.
- **Java SE updates**
Java SE 6 Update 27 is supported.

Chapter 2

Installation and setup

Excelsior JET comes preconfigured for one of the recent Java SE versions, such as 1.6.0_27 (Java SE 6 Update 27.)¹ If your application can be run successfully on that particular version of the Oracle JRE *and* does not require endorsed standards override, you can begin using Excelsior JET right after installation (see [2.2](#)). If your application requires a different Java SE microversion or overrides endorsed standards, you also have to configure Excelsior JET accordingly before compiling your application. See [2.3](#) for details.

¹This is a microversion of the latest major Java version which compliance test suite (JCK) Excelsior JET has passed.

2.1 System requirements

The system requirements for **Excelsior JET itself** are as follows:

CPU:	Intel Pentium III or compatible at 800 MHz or higher
RAM:	1.5GB minimum, 2GB or more recommended
Disk space:	650 MB minimum Note: Creation of a new profile (see 2.3) requires up to 750MB extra for temporary use.
Display:	The JET Control Panel (see Chapter 4) and JetPackII (see Chapter 6) graphical tools require display resolution of 1024x768x256c minimum (HiColor recommended). If such resolution is not available, the product may still be used in the command-line mode.
Operating system:	kernel 2.6 or above glibc 2.3 or above NPTL 2.3 or above X11 and <code>xterm</code> are required by graphical tools. <code>xterm</code> must be reachable via the command search path.

The system requirements for **applications compiled by Excelsior JET** are as follows:

CPU:	IA-32 (Pentium Pro and above) or compatibles
RAM:	Most applications will need the same or somewhat lower amount of RAM as if run on the Oracle JRE
Disk space:	A binary executable created by Excelsior JET in the default mode will occupy 2-3 times more disk space than the original jar file. It also requires the Excelsior JET Runtime (set of shared libraries and data files) to run, but does not require the Oracle JRE .
Display:	Same requirements as if run on the Oracle JRE
Network:	If you deploy the optimized applications using the Java Runtime Slim-Down model, target systems must have Internet/Intranet connection for possible download of the detached package. For more details, see Chapter 9.
Operating system:	Excelsior JET 7.6 for Linux has passed Java SE 6 compliance tests on the following Linux distributions: <ul style="list-style-type: none">• RedHat Enterprise Linux AS 4.0• RedHat Enterprise Linux AS 5.0• SUSE Linux Enterprise Server 9• SUSE Linux Enterprise Server 10

In general, the compiled applications require the following system libraries:

- kernel 2.6 or above
- glibc 2.3 or above
- NPTL 2.3 or above

Note: Excelsior provides full Technical Support Service for RedHat Enterprise Linux 4.0, RedHat Enterprise Linux 5.0, SUSE Linux Enterprise Linux 9 and SUSE Linux Enterprise Linux 10 under the terms of available Support Contracts.

Providing support services for other Linux flavors is at discretion of Excelsior and *may* require an additional consultancy agreement. For more information, contact Excelsior Support Service department at

`support@excelsior-usa.com`

Excelsior JET has been certified for compliance with the specification of the Java platform, Standard Edition 6, and contains the licensed code of this version of the Java SE platform API.

2.2 Installation

Excelsior JET for Linux is distributed as a self-extracted ZIP archive. To install it, change the working directory to `/usr/local` or another location of your choice and run the `.bin` file you have downloaded or received on CD:

```
chmod a+x ~/downloads/jet-760-pro-en-linux.bin
cd /usr/local
~/downloads/jet-760-pro-en-linux.bin
```

It will invoke the `more` utility to display the license agreement. After you page through it, the following message shall display:

```
Do you agree to the above license terms? [yes or no]
```

Type "yes" and hit **Enter** to accept the license and the installer will unpack the product files into a subdirectory named in accordance with product version and edition, e.g. `./jet7.6-pro`.

Now you need to set up the environment. Suppose you have unpacked JET into the directory *JET-home*. The directory *JET-home/lib/x86/shared* must be listed in the `LD_LIBRARY_PATH` environment variable, and *JET-home/bin* in the `PATH` environment variable.

For instance, if you are using bash or Bourne shell, use the following commands:

```
export PATH=JET-home/bin:$PATH
export LD_LIBRARY_PATH=\
JET-home/lib/x86/shared:$LD_LIBRARY_PATH
```

There is a bash script in the Excelsior JET installation directory that you can use to set these variables:

```
cd /usr/local/jet7.6-pro
source setenv
```

Note you have to run the script as `source setenv` (can be abbreviated as `<dot><space>setenv`) so that it affects the current shell.

Run the command

```
jetsetup -show-profiles
```

If your application works fine on the version of the Oracle JRE indicated in the existing profile, you are done with setup and can begin optimizing your application

Otherwise, use JET Setup to create a new profile (see 2.3).

2.3 Profiles

A *profile* is the set of Excelsior JET Runtime files² for the given Java SE version and the given set of jars overriding endorsed standards. A new installation of Excelsior JET

²shared libraries with precompiled platform API classes, shared libraries with native methods, resource files, etc.

contains a single *default* profile. It is typically based on the latest Java SE microversion available at the moment of publishing the current release of Excelsior JET. This profile has no Endorsed Standards overridden.

Using the JET Setup program, you may create additional profiles and switch between them according to your applications' requirements. New profiles are based on any of Java SE microversions supported by Excelsior JET. The list of supported microversions is displayed by JET Setup when you start creating a new profile. Typically, the just released Excelsior JET supports only one Java SE microversion. As new Java SE updates are released by Oracle, Excelsior provides updates for JET that add support for these Java SE microversions. Visit Excelsior's Web site to check what versions are currently supported.

Creation of a new profile includes precompiling of Java platform classes for a particular Java SE microversion into a set of shared libraries. It is important to note that if your application relies on the Endorsed Standards Override Mechanism, the overriding classes have to be precompiled instead of their counterparts shipped with the Java platform.

To display the list of available profiles, execute the following command:

```
jetsetup -show-profiles
```

The output will look as follows:

```
Please wait...
The following profiles were found by JET Setup:

  0: profile1.6.0_27
Profile status: Active
Java SE: 1.6.0_27
```

Each item in the list of profiles contains a profile index and name, a status tag, and the names of jar files that contain classes overriding endorsed standards, if any were specified during profile creation.

Profile names are based on respective Java SE version numbers, for example name "1.6.0_27 (2)" means that the profile is based on Java SE 6 Update 27.

Status tags have the following meanings:

Active

The profile is consistent and up-to-date and Excelsior JET is currently set up to compile your applications against it.

Ready

The profile is consistent and up-to-date. Select it and click **Activate** to make it the active profile.

Outdated

The profile may not be used, because an Excelsior JET update (see [2.4](#)) was applied or rolled back, so a (partial) rebuild of the profile is necessary for correct operation.

Damaged

The profile was damaged and may not be used. Usually this indicates that one or more of its files are missing.

RemoveOnly

There is a profile directory on the disk, but JET Setup has no information about that profile. Usually this happens as a result of installing Excelsior JET on top of the previous installation. All you can do with such a profile is remove it.

To create a new profile, run the `jetsetup` utility as follows:

```
jetsetup -create-profile supported-java-se-version [options]
```

where *supported-java-se-version* is the exact Java SE version on which the profile will be based, for instance:

```
jetsetup -create-profile 1.6.0_27
```

To find out which Java SE microversions are supported by your Excelsior JET installation, run `jetsetup` without parameters.

To specify classes overriding endorsed standards, use one of the following options:

```
-endorsed-dirs dir-list
```

dir-list is a colon-separated list of directories containing jar files overriding endorsed standards. Ignored if the option `-create-profile` is not specified or the option `-endorsed-jars` is specified.

```
-endorsed-jars jar-list
```

jar-list is a colon-separated list of jar files overriding endorsed standards. Ignored if the option `-create-profile` is not specified.

JET Setup will display output similar to the following:

```
Please wait...
Compiling Java SE API classes. Please stand by...
Please wait while JET Setup prepares for compilation
. . .
```

The process of compilation can take from 30 minutes to several hours, depending on your system configuration. In the end JET Setup should report that the new profile was successfully created.

To change the active profile, run `jetsetup` as follows:

```
jetsetup -activate-profile index
```

where *index* is the numerical index of the desired profile in the list displayed by the command “`jetsetup -show-profiles`”.

To refresh an outdated profile, run `jetsetup` as follows:

```
jetsetup -refresh-profile index
```

where *index* is the numerical index of the profile you want to refresh in the list displayed by the command “`jetsetup -show-profiles`”.

To make an outdated profile ready for use without refreshing it, use the Update Manager (see 2.4.2) to rollback to the update level matching that profile.

To repair a damaged profile, run `jetsetup` as follows:

```
jetsetup -repair-profile index
```

where *index* is the numerical index of the profile you want to repair in the list displayed by the command “`jetsetup -show-profiles`”.

To remove a profile you no longer need, run `jetsetup` as follows:

```
jetsetup -remove-profile index
```

where *index* is the numerical index of the profile you want to remove in the list displayed by the command “`jetsetup -show-profiles`”.

2.4 Updates

Excelsior periodically issues updates to the Excelsior JET product which fix bugs, add support for the latest Java SE microversions and minor feature enhancements. The JET Setup tool includes an update manager that enables you to install and apply product updates, rollback to previous update levels and select custom combinations of updates.

The set of files constituting the product is split into logical components, such as compiler, runtime, graphical tools and utilities, samples, documentation, and so on. When a defect is resolved in one of the components, the engineering team may issue a *hotfix* containing the corrected version of that component.

Since components are dependent on each other, a hotfix may include two or more updated components, for instance, a fix in the compiler module that emits object files may require changes in the linker.

Hotfixes are always *cumulative*: a component included in a hotfix is built from sources containing fixes for all bugs addressed by previously issued hotfixes containing that component. This rule substantially simplifies update management.

Hotfixes are available on-demand. If you report a particular problem that is already resolved through a hotfix, the support team will email that hotfix to you.

A *Maintenance Pack* is essentially a hotfix for all components updated since the initial release of the given version of the product. Maintenance Packs are announced through mailing lists and are made available for download.

To minimize download size, any hotfix issued after the latest Maintenance Pack requires prior installation of that Maintenance Pack.

2.4.1 Update installation

An update is distributed as an executable file. When run, it prompts you for the location of the respective version of Excelsior JET on your system. It then unpacks the files constituting the update into a special location inside the JET directory and invokes JET Setup to apply the update.

Note: Updates that make changes in the compiler and runtime are most likely to change the status of the **Active** and **Ready** profiles (see 2.3) to **Outdated**. After applying such an update use, use JET Setup to refresh the **Outdated** profiles or create new ones as

described in 2.3. If you need to install several updates, e.g. a Maintenance Pack and a hotfix issued after that Maintenance Pack, first install them all and then run JET Setup to refresh or create profiles.

2.4.2 Update management

Once you have installed one or more product updates, you can rollback to any update level or return to the latest level at any time as follows:

1. Display the list of installed updates using the command:

```
jetsetup -show-updates
```

2. Locate in the list the update level to which you want to revert and memorize its numerical index. Then run `jetsetup` again as follows:

```
jetsetup -switch-to-update index
```

3. Display the list of profiles available for the new update level:

```
jetsetup -show-profiles
```

4. If necessary, refresh one or more profiles or create new ones as described in 2.3.

If an Excelsior support engineer has advised you to select a particular combination of updates, do the following:

1. Display the list of installed updates using the command:

```
jetsetup -show-updates
```

2. Locate in the list the updates which you need to activate. Then run `jetsetup` again as follows:

```
jetsetup -activate-update list
```

where *list* is a colon-separated list of update names.

3. Display the list of profiles available for the new combination of updates:

```
jetsetup -show-profiles
```

4. If necessary, create new profiles as described in 2.3.

2.4.3 Cleaning up backed up configurations

Each time you install an update that invalidates the currently set up profiles (see 2.3), JET Setup backs up those profiles into a subdirectory of your Excelsior JET installation so as to enable rollback (see 2.4.2). Profiles occupy dozens of megabytes of disk space, so you may wish to remove profiles for update levels to which you do not plan to revert.

To remove backed-up profiles, run the `jetsetup` tool as follows:

```
jetsetup -cleanup-backup
```

2.5 JET Setup options summary

On Linux, JET Setup is only available as a command-line utility. To access JET Setup functionality from the command line, invoke the `jetsetup` utility with one or more of the following options:

`-create-profile` *full-java-se-version*

Initiates creation of a new profile (see 2.3). *full-java-se-version* is the complete version number of the Java platform for the new profile, for instance:

```
jetsetup -create-profile 1.6.0_20
```

`-endorsed-dirs` *dir-list*

dir-list is a colon-separated list of directories containing jar files overriding endorsed standards. Ignored if the option `-create-profile` is not specified or the option `-endorsed-jars` is specified.

`-endorsed-jars` *jar-list*

dir-list is a colon-separated list of jar files overriding endorsed standards. Ignored if the option `-create-profile` is not specified.

If this option is specified, the option `-endorsed-dirs` is ignored.

`-show-profiles`

Displays the list of currently available profiles.

`-activate-profile` *index*

Activates the profile denoted by *index* in the list of profiles. The profile must have **Ready** status.

`-refresh-profile` *index*

If the status of the profile denoted by *index* in the list of profiles is **Outdated**, refreshes it.

`-repair-profile` *index*

If the status of the profile denoted by *index* in the list of profiles is **Damaged**, repairs it.

`-remove-profile` *index*

Removes the profile denoted by *index* in the list of profiles.

`-show-updates`

Displays the list of installed and applied updates.

`-activate-update` *update-names*

Activates the updates denoted by *update-names*, which is a colon-separated list of update names.

`-switch-to-update index`

Reverts to the configuration of updates denoted by *index* in the list of applied updates.

`-cleanup-backup`

Cleans up the backup directory.

`-version`

Displays JET Setup version.

2.6 Reproducing the build environment

You may need to reproduce the Excelsior JET build environment for the given version of your application in the following circumstances:

- your hard disk has failed and your Excelsior JET installation was not backed up;
- you are moving the build to another system, e.g. a build server;
- you need to issue a maintenance update for an old version of your application built using a previous version of Excelsior JET;
- etc.

If you do not have an SCM system or if your Excelsior JET installation is not managed, make sure to write down the following information for each version of each application you ship to end users or install in your production systems:

- the version of Excelsior JET used (e.g. 7.6)
- the exact combination of applied Excelsior JET updates (see 2.4), e.g. `jet-760-mp1`, `jet-760-hotfix14`
- the exact version of Java SE (e.g. 1.6.0_27) in the active profile (see 2.3)

If you are using the endorsed standards override (see 2.3) feature, you also need to keep copies of the respective jars or at least write down their exact versions.

Chapter 3

Excelsior JET Launcher

Excelsior JET Launcher is provided for quick setup of Excelsior JET projects. The Launcher has the same command line interface as the standard `java` launcher but instead of immediately running the program, it collects the application's settings, such as classpath, the main class, working directory and so on, and then invokes the JET Control Panel wizard (see Chapter 4).

This simplifies project creation to a great extent because most of required settings are made automatically. After that, you use the GUI wizard to choose how to optimize your application¹, set a few options and start the JET Optimizer to produce a native executable.

Note: the JET Launcher is the easiest way to set up a project if your Java application is started via a complex script file. You just need to replace the pathname of the standard launcher in the script and start it.

3.1 Creating a new project

To start the JET Launcher, use its full path name for the current JET profile², e.g.

```
<JET Home>/profile1.6.0_20/jre/bin/java
```

and specify the same arguments as for the standard `java` launcher.

If you use an IDE, such as IntelliJ IDEA, JBuilder, or Eclipse, the `java` launcher command line is typically displayed at the top of a Run window that appears when you start your application from within the IDE. You can simply cut-and-paste it, replace the "java" command with the full path to the JET Launcher and enter the line on a command prompt.

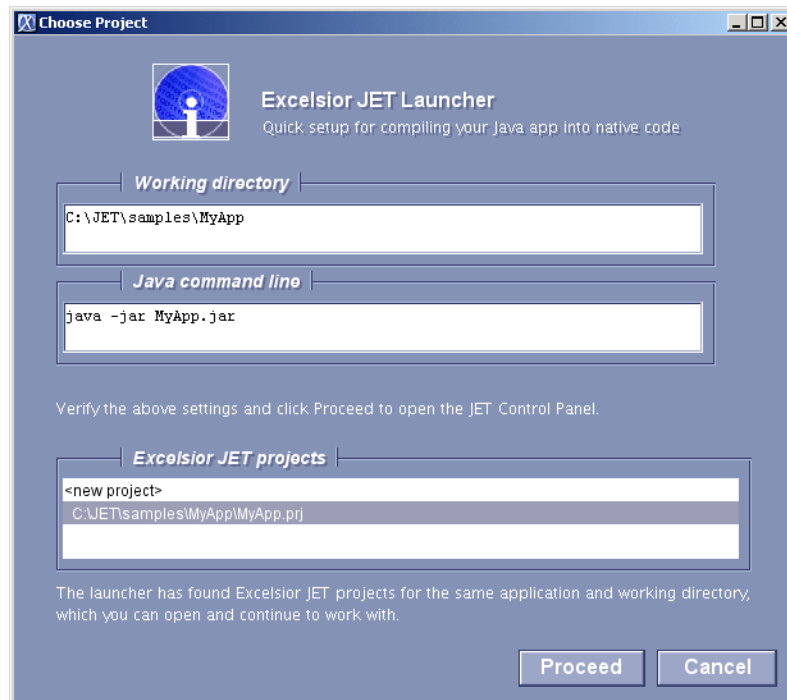
This brings up a dialog that displays the exact java command line and the application's working directory. Click **Proceed** to create a new project and open it in the JET Control Panel (see Chapter 4).

¹For example, you can select what is more important for your Java application: performance enhancement or download size reduction

²Subdirectories `profile<java-version>` of the JET installation directory contain JET profiles, that is, all the files specific to a Java SE version supported by Excelsior JET, such Java SE 6 Update 20 (1.6.0_20). The active (current) profile is managed through the JET Setup tool (see 2.3).

3.2 Opening an existing project

Once you have created a project file, you may continue using the Launcher to open the project with JET Control Panel. When you start the Launcher from the same working directory and supply the same arguments, it looks for existing projects, matches the settings, and displays the names of appropriate projects in the **Excelsior JET projects** pane:



Select the desired project or the **<new project>** option from that pane and click **Proceed**.

Chapter 4

JET Control Panel

Excelsior JET Control Panel is a graphical tool that you use to quickly set up JET project files and compile your Java applications to native code. Its wizard-like interface guides you through the process of JET project creation step-by-step.

4.1 Before you begin

Before using Excelsior JET to compile your Java application, make sure that the application runs properly on the reference implementation of the Java platform, such as Sun JRE 6.0.

4.2 Starting Excelsior JET Control Panel

To start the JET Control Panel, type

```
jetcp
```

at the command prompt and press **Enter**.

A splash screen with a “Loading...” progress indicator will appear. When loading is over, the Welcome Screen will be displayed.

In the middle column, click:

- **Plain Java SE application** to create project for a conventional Java application.
- **Eclipse RCP application** to create project for RCP application (see Chapter 7)
- **Tomcat Web application** to create project for Web applications based on Apache Tomcat (see Chapter 8)
- **Invocation library** to create project for to build DLL callable from a non-Java environment (see Chapter 12.2)

The setup procedure is different for each type of project, thus you have to select it at the very beginning. You may change it later by selecting the respective item from the **Project/Convert to** menu.

4.3 JET Control Panel overview

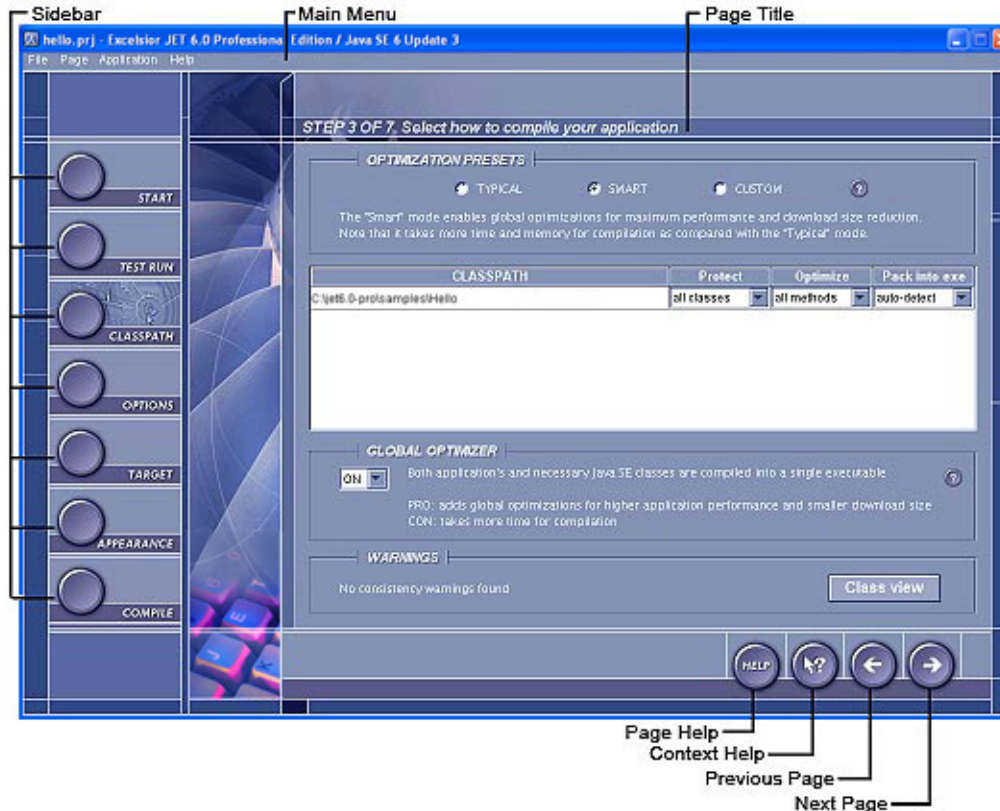


Figure 4.1: Excelsior JET Control Panel

The user interface of JET Control Panel is organized as a series of “pages”. Each page represents one logical step in the process of setting up your project. As shown in Figure 4.1, the most page space is occupied by page-specific controls, but there are also some common controls: sidebar on the left, main menu at the top, and help/navigation buttons in the bottom right corner.

You may navigate through the pages using the **Next Page / Previous Page** buttons, or go directly to a page by either clicking on the respective sidebar button or selecting that page in the **Page** menu.

Online help is available to assist you at any point. To display help information for the current page, click the **Page Help** button. To get help on a particular control, click the **Context Help** button. The mouse cursor will change shape to an arrow with a question-mark — click the control you want to learn about to get displayed the respective portion of the online help.

4.4 Working with projects

A JET project consists of configuration settings which together fully describes the final executable to be produced by the compiler. A project defines:

- Type of the application: plain Java application, Tomcat Web application or Eclipse RCP application
- Type of the resulting binary file: executable or shared library (see 4.5)
- Classpath — a list of directories and jar/zip files which contain application's class files and resources
- The main class (unless you create a shared library)
- Java system properties required by the application, if any
- *Compilation mode* — The JET Optimizer can operate in different modes that affect application performance, code protection, download size of the resulting installation package, etc. You can make the settings for each compiled application (see 4.7)
- *Resource packing mode* specified for each jar/zip file from the classpath. It enables you to pack application's resources (e.g. images, media, and property files) into the executable, much like it is done for a native application (see 4.7)
- Most important options that control code optimizations, application memory usage and other run time settings (see 4.8 and 4.9)
- *Application appearance* — a splash screen that you may assign to the executable (see 4.10)

This information is stored in the *project file* which you create as described below in this chapter. JET project file names have the extension `.prj`

4.4.1 Creating a new project

You may create a new JET project by selecting the desired type of the application in the middle column of the Welcome Screen or by choosing those options in the **File/New Project** menu. If there is already an opened project, the JET Control Panel will prompt you to save it first.

4.4.2 Opening a project

You may open a previously created project by clicking the **Open** icon on the Welcome Screen or by selecting the **File/Open** menu item. A conventional file dialog will appear; use it to browse for the JET project file you want to open or select the desired project in the **File name** combo box.

Later you may quickly open an existing project by selecting it in the **Recent Projects** column of the Welcome Screen or using the **File/Reopen** menu item.

You may also open project files you created manually for `jc`, the JET command-line compiler (see Chapter 16.) However, the Control Panel recognizes only a subset of the project file syntax (see 16.8), so it may refuse to open a project file that uses advanced features, such as preprocessing (see 16.7.)

4.4.3 Saving a project

You save a project by selecting the **File/Save** item from the **Main Menu**. When you save a newly created project for the first time, a conventional file dialog is displayed. Use it to specify the directory and name for your project. The default extension `.prj` is added to the file name automatically.

Note: during compilation the JET Optimizer will store Project Data Base (see 16.5) into the project directory.

4.4.4 Saving a project under a different name

To save the current project under a different name or into another directory, select the **File/Save As** item from the **Main Menu**. A conventional file dialog will appear; use it to specify the new directory and/or name for your project.

Note: during compilation the JET Optimizer will store Project Data Base (see 16.5) into the project directory.

4.4.5 Closing a project

You may close the current project by selecting the **File/Close Project** item from the main menu. This brings up the Welcome Screen. If the current project has been modified, the Control Panel will prompt you to save it first.

4.5 Step 1: Describing an application

This is the first step of creating a new project, where you describe your application to Excelsior JET Optimizer. To do that quickly, you can fill a simple two-question form placed in the **Description form** pane.

Note: if you are creating a project for a Tomcat or Eclipse RCP application, refer to sections 8.2.1 or 7.2.1, respectively.

4.5.1 Description form

Note: if you use Excelsior JET Launcher (see Chapter 3) to create a new project, all settings from the description form are made automatically so you do not need to fill it

again. You may look through the form and the right-side panes to verify that the displayed settings are correct.

Question 1: Application's working directory

Working directory of an application is the current system directory at the application's start. When answering Question 1, you should specify the directory from which you launch your Java application on the system. For that, type the full path of the directory (e.g. ~/MyApp) in the entry field below, or click **Browse** to select it using a file dialog.

Note: the compiler will place the resulting executable into the specified directory to guarantee that the compiled application has the same working directory. Many Java applications should be run only from a particular directory, e.g. if they use relative path names to refer to classes or resources. That is why it is important to keep the same application's working directory after compilation.

If you are pretty sure that your application runs from any directory, simply specify that where you want the compiler to place the executable.

Question 2: Classpath, main class, Java properties

Classpath is an ordered list of jar/zip files and directories which contain the application's classes and resources. You specify the classpath when answering Question 2. In addition, you should select the application's main class unless you want to produce a shared library. To complete the description, you may also set Java system properties required by your application, if any.

To specify all the settings at once, type or paste the command line you use to launch your application on a JVM, e.g.

```
java -cp ./MyLib/MyApp.jar -Xmx128m MyMain
```

in the **Command line** field and click the **Parse** button to the right.

If you use an IDE, such as IntelliJ IDEA, JBuilder, or Eclipse, the java launcher command line is typically displayed at the top of a Run window that appears when you start your application from within the IDE. You can simply cut and paste it into the **Command line** field and then click the **Parse** button.

The JET Control Panel automatically extracts the parameters from the specified command so all you need to do is see if values of the settings displayed in the panes **Classpath**, **Application main class**, and **Java system properties** on the right appear to be correct.¹

Note: if your application is launched with a complex script file that requires substitutions of environment variables, it is hard to figure out an exact command line. In such case, exit the Control Panel and use Excelsior JET Launcher (see Chapter 3) to create a new project.

After completing the form, you may go to the next step unless you wish to modify some settings manually as the next section describes.

¹Other settings specified in the command line, such as maximum heap size, are also parsed and added to the project. The Control Panel displays them on subsequent pages, for example, see 4.8.2.

4.5.2 Manual setting

You can correct the description of your application, if necessary. For example, you may change values of Java system properties or add a new classpath entry. In such case, there is no need to fill the form again - you can use the controls placed on the right side of the page.

Editing classpath

The **Classpath** pane shows the application's classpath. When looking for a class, the JET compiler inspects the classpath entries in the order they appear in the pane, from top to bottom.

Note: It is therefore important that you place classpath entries in the same order as a Java Virtual Machine accesses them when loading the application's classes². Otherwise, there is a chance that an other than intended version of a particular class will be used at run time. It may cause unpredictable changes of the application's behavior. The same applies to resource files: the JET Runtime looks for them using the classpath so the order of classpath entries affects lookup of resources as well.

To edit the classpath, check **Enable manual settings** checkbox. These four buttons appear to the left of the **Classpath** pane:



To add entries to the classpath, do the following:

1. Click the **Add** button or right-click anywhere in the **Classpath** pane and then select **Add classpath entries** from the pop-up menu. A conventional file dialog will appear.
2. Do one of the following:
 - Choose the application's jar/zip files and click **Select**.
 - Choose the directories that contain the application's classes and/or resources, then click **Select**.

To remove a classpath entry, do the following:

1. Select an item you want to remove and click the **Remove** button or right-click the item and select **Remove the entry** from the pop-up menu. A confirmation box will appear.

²This is the order in which classpath entries are listed in the `java.class.path` system property set by a "java" launcher.

2. Click **Yes** to remove the entry or **No** if you decided to leave it in place.

To change the order of classpath entries, select the item you want to move and click the **Move Up / Move Down** buttons until the item is placed properly.

Specifying the main class

As you know, execution of a Java application starts with a `public static void main (String[])` method. However, more than one class may contain such a method. When running your program on a JVM, you specify the name of the desired main class to the Java launcher as a command line parameter or in the manifest file of the jar file. Similarly, the JET Optimizer needs to know the name of the main class to make control passed to its `main` method on application startup.

You specify the main class as follows:

- Type the full name of the main class, e.g. `com/myCompany/myApp/myMain`, in the **Application Main Class** field
- Click the **Select** button next to that field. A dialog will display with the list of all “potential” main classes found in the classpath. Select the desired class and click **Ok**

You do not need to select the main class if you want to build a shared library.

Setting Java system properties

To specify Java system properties required by your application (those passed to the Java launcher using the `-D` prefix), type them in the entry field from the **Java System Properties** pane, one on a line.

For instance, if you run your application using the command

```
java -Dmyapp.debug -Dmyapp.log=/var/log/myapp MyApp
```

type the following in the field:

```
myapp.debug
myapp.log=/var/log/myapp
```

4.6 Step 2: Performing a test run

On this step, you can run your Java application *before* compiling it to native code³ which helps Excelsior JET:

- verify the settings you made on the previous step

³Excelsior JET Runtime will be used to execute application’s bytecodes

- collect profile information to optimize your app more effectively

To perform a test run, type application's arguments in the field **Arguments**, if any, and click **Run**. The application's output⁴ will be displayed in the **Output** pane. After the application terminates, the exit code will be also shown in this pane.

Note: the application is run in a special TEST MODE so disregard its modest start-up time and performance on this step.

4.6.1 Why is it?

It makes sense to perform a test run at least once when creating a new project. Doing so, you make sure that the description of your application is correct. For example, if you forgot to specify an application's jar file on the previous step, the application will not work due to absence of some classes. You can see the respective exception message in the **Output** pane, add the missing jar file to the project, and run the application again. Thus, you will be sure that all project settings are correct when starting optimizing compilation on Step 7 (see 4.11).

In addition, when you run the application, its execution profile is collected and saved into a file that will be added to the project automatically. The profile information will be used by:

- the Startup Optimizer (see 14.2.1)
- the Global Optimizer, if you enable it on the next Step (see 4.7.3).

4.7 Step 3: Selecting a compilation mode

Note: if you are creating a project for a Tomcat or Eclipse RCP application, first read sections 8.2.3 or 7.2.3, respectively.

The purpose of using the JET Optimizer varies for different Java applications. One may need to protect certain Java classes from decompilers, improve application performance, reduce download size and disk footprint of the installation or combine these requirements.

On this page, you choose a compilation mode suitable for your Java applications. The **classpath grid** and **Global Optimizer** controls (see 4.7.2 and 4.7.3) allow you to make the required settings. In addition, the Control Panel provides two "pre-defined" compilation modes which you can quickly select using *optimization presets*.

4.7.1 Optimization presets

To enable either pre-defined compilation mode, select "TYPICAL" or "SMART" radio button from the **Optimization presets** pane. Description of each option is given below.

⁴Information printed into the standard output and error channels

Typical mode

This is the default compilation mode in which all classes from the application's classpath are compiled into an executable. The standard Java classes need not be compiled because they are included in the JET Runtime (also in the native compiled form.)

Choosing the Typical mode, you enable most of performance optimizations and save compilation time. In addition, if you compile several executables, they will share the JET Runtime components. Note, however, that the Runtime includes *all* Java platform classes pre-compiled and it should be distributed "as is"⁵. Even if your application actually uses only a small fraction of the Java SE API, size of the resulting installation package will be rather large.

If you need to reduce the download size and/or disk footprint of your application, consider the Smart mode.

Smart mode

The Smart mode relies on a simple observation: an application using a Java API actually requires *only a part* the API's classes so there is no need to native compile the entire API implementation.

Choosing the Smart mode, you enable the Global Optimizer (see 4.7.3) that detects the Java platform API classes which your application actually uses. Then, both application and used platform classes are compiled into a single executable. The remaining classes are kept in the bytecode form and handled by the JIT compiler, if the application tries to load them at run time.⁶

Such mixed code packages are very effectively compressed by Excelsior Installer (see 6.13.1) so download size of the optimized Java applications is significantly reduced. In addition, the Global Optimizer improves start-up time and performance and reduces application disk footprint and memory usage. Note however that it takes more time and memory for compilation as compared with the Typical mode.

Custom mode

This option is selected automatically, if you change the default settings in the **classpath grid** or **Global Optimizer** controls described below.

4.7.2 Classpath grid

The classpath grid allows you to select how to compile your application at discretion of classpath entries (jar/zip files or directories.) That is, for each entry, you can enable/disable:

⁵Any compatible Java implementation *must* include all the classes described in the Java SE platform specification even though a particular application never uses all of them

⁶Due to dynamic features of Java, the Global Optimizer cannot detect an exact set of used classes, e.g. those accessed from within native methods via JNI. That is why a few classes may be JITed at run time (see chapter 13 for details)

- code protection for all classes
- selective compilation of classes
- packing resource files into executable

Note: by selecting an optimization preset (see 4.7.1), you reset the classpath grid in a default state. Therefore, you should use this control only if you need to modify the default settings.

The **Classpath** column shows the classpath entries you specified on Step 1. The next three columns display the properties, which you can change for each entry. You do that by selecting options from combo boxes placed into cells of the **Protect**, **Optimize**, and **Pack into exe** columns described in details below.

Code protection

If you need to protect your classes from decompilers, make sure that the respective classpath entries have the protection property enabled. It is displayed in **Protect** column under the name "all classes". For example, if your proprietary classes reside in a jar file, verify that protection is enabled for it. For sure, you can inspect which classes are contained in the jar file using the **Class view** dialog (see 4.7.5.)

If you need to force protection for a certain classpath entry, select the "all classes" option from the combo box placed to the right.

Finally, if you have noticed that some entry has protection enabled but you actually do not need it, select the "not required" option from the respective combo box. It may reduce compilation time and the size of the resulting executable in some cases.

Selective optimization

By default, the JET Optimizer compiles all application's classes into native code and all methods of every class are optimized for performance. For each classpath entry, this property is displayed in the **Optimize** column of the classpath grid under the name "all methods". For example, if you opted for the Typical optimization preset (see 4.7.1), all classpath entries have the "all methods" property enabled.

The other option you can select in this column is "auto-detect". It means that the Optimizer detects which classes from the classpath entry are used by the application and compiles the entry selectively leaving a part of classes in the bytecode form. It helps reduce compilation time and download size of the application⁷.

You may enable selective optimization for third-party APIs (jar files), if you are sure that your application uses only a small part of their implementing classes. However, it is *not* recommended to choose the "auto-detect" option for your own jar files, because, in general, the Optimizer cannot detect an exact set of used classes. The "missed" classes are handled by the JIT compiler (see Chapter 11) at run time, which incurs some performance overheads.

⁷Significant reduction in download size is provided only if the Global Optimizer is turned on (see 4.7.3)

Note that when you choose the Smart optimization preset (see 4.7.1), the Control Panel uses an heuristics to separate the classpath entries into the *application part* (your own classes) and *API part* (third-party classes). As a result, certain jar files related to the API part, have the "auto-detect" option enabled. If the heuristics proves to fail for some classpath entries, you can correct the settings using the combo boxes placed in the **Optimize** column.

Resource packing

Jar/zip files often contain resource files such as images, icons, media files, etc. By default, the JET Optimizer packs such files into the resulting executable. If selective optimization is enabled for a jar/zip file (see 4.7.2), the classes not compiled are also packed into the executable. As a result, original jar/zip files are no longer needed for the running application.

This is the default option displayed in the **Pack into exe** column under the name "auto-detect". Notice that it works only for jar/zip files so directories from the classpath are marked with "n/a" (not available) in this column.

Note: in most cases, you *do not need* to change the default setting. Other available options are considered advanced and described in section 10.13.

4.7.3 Global Optimizer

The Global Optimizer provides higher performance and lower memory usage for the compiled application. In addition, the installation created for an executable built with the Global Optimizer has much smaller download size and disk footprint. In exchange, the Optimizer requires more time and memory to compile a Java application.

The Global Optimizer is enabled as a part of default settings for the Smart optimization preset (see 4.7.1). To enable it separately, select the "ON" option from the combo box placed in the **Global Optimizer** pane.

Important note: For optimal usage of the Global Optimizer, perform a test run for your application on the previous Step (see 4.6). During a test run, application profile is automatically collected and then added to the project. The Optimizer will use it when creating the executable to achieve maximum performance for your Java application. Though test run is an optional step, it is recommended to perform it at least once for a new project.

For more details, consult Chapter 13 that presents an in-depth look at the Global Optimizer.

4.7.4 Inconsistency warnings

The information panel placed at the bottom of the Classpath page displays the number of *inconsistency warnings* detected by the Control Panel in the classpath. For instance, if two classes with the same full name are found in two different jar files from the classpath, class duplication warning is issued. Another typical example is absence of a referenced

class. If the Control Panel detects that one class refers to another class that cannot be found in the classpath, class absence warning is issued.

If you want the Control Panel to analyze your project, click the **Check consistency** button on the right. It brings up the class view dialog that displays detailed information about each warning.

4.7.5 Class viewer

The class view dialog allows you to get information about:

- what classes and packages are contained in a classpath entry
- import dependency between classes
- inconsistencies found in the classpath

Note that this viewer is provided for reference only and does *not affect* compilation in any way. You may open the class view dialog via the **Project/View Classes** menu.

Inspecting classpath entries

The **Classpath** pane on the left shows the application classpath, an ordered list of JAR/ZIP files and directories that include the application's classes. You can inspect what classes and packages are contained in a classpath entry by expanding the entry's node in this pane.

Examining import dependency

Import dependency of class *A* is a set of imported classes that includes

- the classes referenced by *A*
- the classes referenced by the imported classes, recursively

Import dependency of a package or classpath entry includes all import dependencies of the contained classes.

You need to examine the dependencies if you are going to compile your application into an executable and one or more dynamic libraries as described in chapter Dynamic Linking (see 12).

All classes and packages found in the classpath are displayed in the **Classes** pane. Each class or package name appears with a color tag displayed to the left. These tags indicate import dependencies between classes and packages as described below:

- - the class or entire package is marked by the user to examine its dependencies
- - the class or entire package is imported, that is, some classes marked by the user depend on it
- - the class or package is neither marked nor imported
- - the class is ignored because another class with the same full name was found earlier in the classpath or because this class belongs to the Java SE platform API. See 4.7.5 for details
- - the package is partially imported, that is, it includes both imported and non-imported classes

When you open the class viewer for the first time, all class and package names have the tag ■ displayed. Click on the tag of the class or package whose import dependency you want to see. As a result, the tag will change to the tag "selected" (■) and the Control Panel will automatically mark all imported classes with the tag ■.

To cancel your selection, click the tag ■ and the marking will be cleared. Note that you can also examine import dependency of an entire classpath entry by clicking on its tag in the **Classpath** pane. You can do the same actions using a pop-up menu: right-click the class, package, or classpath entry name and select the **Mark** item from the menu.

Classpath inconsistencies

The JET Control Panel can detect the following inconsistencies in the classpath:

- Two or more classes with the same fully qualified name are included in the classpath
- A class from the Java SE platform API is found in the application classpath
- A class has unresolved references

If the Control Panel has detected any of such problems, the **Warnings** tab appears in the information area at the bottom of the dialog. Under this tab, the overall numbers of "problematic" classes are displayed for each type of inconsistency. Click the respective **Details** button on the right to display a dialog box with a list of involved classes.

Duplicated classes

When multiple classpath entries contain identically named class files, the entry listed in the classpath first prevails. As a result, the "overrides" tag ■ is displayed next to the first occurrence of such a class in the **Classpath** pane. All other occurrences are displayed with the tags "overridden" ■ and "ignored" ■.

To view the list of duplicated classes, do the following:

1. Select the **Warnings** tab.
2. Click the **Details** button next to the message "*N duplicated classes*". A dialog box will appear.


3. In the box, expand a class name to view the list of classpath entries that contain the duplicated classes.

You can get this report in a plain text form: click the **Copy to clipboard** button and perform the paste operation in any text editor.

To resolve this issue, it is sufficient to check that the desired versions of duplicated classes appear in the classpath first. If necessary, you can reorder the classpath entries as described in section 4.5.2. After that, you may safely remove the overridden class files from the respective directories and JAR/ZIP files.

Conflicting classes

An application's classpath should not contain classes from the Java SE platform API, because they are already included in the JET Runtime. When processing the classpath, the JET Optimizer automatically excludes such classes from compilation.

Conflicting classes are displayed with the "ignored" tag in the **Classpath** pane. In addition, classpath entries and packages that contain at least one conflicting class are marked with the "conflict" tag .

To inspect the list of conflicting classes, do the following:

1. Select the **Warnings** tab.
2. Click the **Details** button next to the message "*N* classes coinciding with Java SE API classes" to open a dialog box that displays the full list of conflicting classes.

You can get this report in a plain text form: click the **Copy to clipboard** button and perform the paste operation in any text editor.

To resolve this inconsistency, remove all conflicting classes from the classpath. If such classes appear in third-party jars qualified as *Endorsed Standard*, you may create a new JET profile with Endorsed Standards Override (see 2.3).

Unresolved import

If a class *A* references another class not found in the classpath or among the platform classes, the "unresolved import" tag  is displayed next to class *A*.

To browse the list of classes that have unresolved import dependencies, do the following:

1. Select the **Warnings** tab.
2. Click the **Details** button next to the message "*N* classes with unresolved import dependencies". A dialog box will appear.
3. Select the **Unresolved** tab to display the list of classes that have unresolved references. In the list, expand a class name to display the referenced classes not found.

4. Select the **Not Found** tab to display the list of classes that are referenced by classes from in the classpath, but whose class files were not found.


You can get these reports in a plain text form: click the **Copy to clipboard** button and perform the paste operation in any text editor.

To resolve this inconsistency, complete the classpath by adding directories or JAR/ZIP files that contain necessary classes. It may happen, however, that this problem is detected in a code that is not under your control, such as a third-party API or component. Fortunately, the JET Optimizer is able to compile the application even if it comprises classes that have unresolved references.

Note: The JET Runtime correctly handles the situation when a referenced class is not present at compile time but becomes available at run time. See [11.5](#) for details.

To close this issue, you may file a complaint to the vendor that distributes software components with an incomplete set of Java class files.

Unresolved super classes

If class *A* extends another class (or implements an interface) which cannot be found in the classpath, the "unresolved super class" tag  is displayed next to class *A*.

This is unfortunate because in such case, the JET Optimizer cannot compile class *A* into native code.

To browse the list of the concerned classes, do the following:

1. Select the **Warnings** tab.
2. Click the **Details** button next to the message "*N* classes with unresolved super classes". A dialog box will appear.
3. Select the **Unresolved super classes** tab to display the list of classes that have unresolved super classes. In the list, expand a class name to display the required super classes not found.
4. Select the **Not found super classes** tab to display the list of super classes not found in the classpath. Expand the name of a super class to display the classes that extend it.

You can get these reports in a plain text form: click the **Copy to clipboard** button and perform the paste operation in any text editor.

To resolve this serious issue, do complete the classpath by adding directories or JAR/ZIP files that contain necessary classes.

If, however, it is not possible, e.g. the required super class becomes available only at run time, the following happens:

1. the classes that extend an absent class are packed into the executable in the bytecode form⁸
2. if the application loads one of them, the JET Runtime compiles it on-the-fly using the JIT compiler (see chapter 11 for details.)

This technique provides correct behavior of the running application, though some performance loss is possible.

Important note: classes that have unresolved super classes are never compiled into native code before execution and distributed in the bytecode form. As a result, Excelsior JET *cannot protect* them from decompilers. Use the class view dialog to make sure that the classes you need to protect are not affected.

4.8 Step 4: Setting options

On this step, you set various options of the JET compiler and Runtime, which affect performance and memory footprint of your application. The most important controls are available under the **General** tab, whereas the **Advanced** tab contains the settings that need to be changed only in rare cases.

Default values of the available options are suitable for a "typical application" so the controls are preset accordingly. Most probably, you will need to adjust the settings only after you get your application compiled and run. For example, you may wish to control its memory footprint or reduce the size of the resulting executable.

4.8.1 Optimizations

The optimizations implemented in the JET compiler are always enabled by default and most of them cannot be disabled at all. There are just two exceptions: allocation of objects on the stack, which may compromise scalability, and method inlining, which improves performance but also increases code size. You therefore have an option to disable stack allocation and adjust the criteria for inlining.

Allocating objects on the stack

As you know, the Java memory model has no stack objects — class instances put on the stack frame. All objects have to be allocated on the heap by the `new` operator and reclaimed by the garbage collector, even though the lifetimes of some objects are obvious. The JET compiler performs so called *escape analysis* to approximate object lifetimes and allocate some objects and small arrays on the stack where possible. As a result, compiled applications benefit from both faster object allocation and less intensive garbage collection.

⁸Provided that the classes reside in a JAR/ZIP file from the classpath. If they come from a directory, you need to distribute such non-compiled classes along with the executable.

This optimization may however result in higher consumption of stack memory by application's threads. Therefore, in some cases, you need to increase the maximum stack size (see 4.8.4), which may compromise program's scalability (see 10.4.2). If you are compiling a server-side Java application that runs thousands of threads simultaneously, you may wish to disable this optimization by unchecking the **Allocate objects on the stack** box.

Method inlining

When optimizing a Java program, the compiler often replaces method call statements with bodies of the methods to be called at run time. This optimization known as *method inlining*, improves application performance, especially when tiny methods are inlined such as get/set accessors. However, the inlining of larger methods increases code size and its impact on performance may be uncertain.

To control the aggressiveness of method inlining, select the desired level from the **In-line expansion** combo box. If you need to reduce the size of the executable, select the "Low" or "Tiny methods only" option. Note that it does not necessarily worsen application performance.

4.8.2 Memory Management

Maximum heap size

During application execution, the JET Runtime considers many factors, e.g. system workload, to control the Java heap size and collects garbage, when necessary.

Nevertheless, you may restrict the heap size by specifying an upper bound that the heap may not exceed at run time. For that, type it into the **Maximum heap size** field or drag the slider below. You may use suffixes "K" and "M" to denote kilobytes and megabytes, respectively.

Note: If the entered value is too small or too large, the warning icon is displayed next to the field until you correct the value appropriately.

To return to the default behavior, drag the slider to the leftmost position. The word "Adaptive" will be displayed in the **Maximum heap size** field.

See 10.6.2 for more information.

GC ratio

GC ratio is the percentage of execution time that the JET Runtime is allowed to spend for garbage collection. This setting effectively controls memory footprint because the greater GC ratio, the higher frequency of GC invocations, and, hence, the smaller Java heap needs to be allocated. At the same time, it may worsen performance to some extent because the greater GC ratio, the less time is left for execution of the application's code. Thus, you may adjust GC ratio to set an optimal balance between the performance of your application and its memory footprint. See 10.6.1 for a detailed description of this technique.

GC ratio is measured in tenths of a percent, e.g. the default value equals 1.1%. To specify another value, type a fixed point number with one digit after the point into the **GC ratio** field or drag the slider below. The greatest value you can specify is 40.0%.

Note: If the entered value is too large or has a wrong syntax, the warning icon is displayed next to the field until you correct the value appropriately.

Though it's not recommended, you can disable the mechanism. Drag the slider to the leftmost position, the word "Off" will be displayed in the **GC ratio** field.

Handling low memory

Under heavy system workload, your Java application may lack enough physical memory. This is the case, for example, if it runs simultaneously with another application that occupies a good part of total memory.

The JET Runtime invokes the garbage collector if it has detected the low memory state of the system. If that does not help, the Runtime exceeds the physical memory threshold and starts using virtual memory. Note that it acts very tactfully and tries to avoid the painful saving of heap data into paging files (see 10.6.3 for details.)

You may however override the default behavior and opt for throwing `OutOfMemoryError` instead of using virtual memory under such circumstances. To do that, select the "throw `OutOfMemoryError`" option from the **Handling low memory** combo box.

4.8.3 JIT compiler

The JET Runtime includes a just-in-time (JIT) compiler, which allows a statically compiled application to load and use classes that were not pre-compiled (see Chapter 11 for details.) In this pane, you select and configure the JIT compiler to be used for your application. You have the following options:

1. In the **JIT operation mode** combo box, select the desired JIT compiler: "Fast" or "Optimizing". See 11.4.1 for details.
2. In the **JIT caching mode** combo box, define whether the dynamically compiled code have to be retained in the JIT cache:
 - Select "Disabled" if you want the results of JIT compilation to be discarded when your application terminates
 - Select "Enabled (use default directory)" to have the results of JIT compilation cached in the default directory See 11.4 for details.
 - Select "Enabled (use specified directory)" to designate a particular directory for the JIT compiler cache, then type the full path of that directory in the **JIT Cache Directory** field, or click **Browse** to select it using a conventional file dialog.

Note: In general, using an absolute pathname for the cache directory is not recommended as it may cause problems on deployment. If you want to rename

the cache directory or change its location on the end user system, specify the `jet.jit.cache.dir` property when packaging your application for deployment (see 6.10.2 for details.)

Note: The optimizing JIT compiler and the JIT Caching Technology are not available in the Standard Edition.

4.8.4 Advanced settings

Classpath consistency checks

These options control how the compiler treats classpath inconsistencies, such as absence of an explicitly imported class. By default, the compiler works around all inconsistencies, however, it may be worth fixing them instead. If you turn the checks on, the compiler will issue compilation errors on detected inconsistencies to help you correctly fix them.

Note: Sometimes inconsistencies are detected in a code that is not under your control, such as third-party APIs. In such case, you usually do not have a chance to remedy it quickly. However, you can at least check new versions of such API or file a complaint to its creators.

You can turn on the following checks:

- Clear the **Ignore conflicting classes** checkbox to make sure that no classes from the Java platform API are found in the classpath (see 4.7.5). If the compiler detects such class, it issues “Class/sym conflict” error message. To fix it, remove the detected class from the classpath.
- Clearing the **Ignore member absence** checkbox forces the compiler to detect all references to absent fields or methods. Detecting such references will result in the “Undeclared identifier *C.m*” error message which means that some class from the classpath (let us denote it by *A*) refers to absent member *m* of class *C*. Typically, it means that either class *C* or class *A* is obsolete, i.e. class *A* expects another version of class *C* to be found in the classpath.

To fix the inconsistency, make sure that you have the latest versions of classes *A* and *C*. Recompiling Java source files for these classes may help to find out more information about the problem.

By default, the check is turned off and the compiler inserts the code throwing `java.lang.NoSuchMethodError` or `java.lang.NoSuchFieldError` for all locations where absent members are referenced.

- Select the "Compile-time error" option from the **Class absence results in** combo box to make sure that all classes referenced by your application are found in the classpath. For any referenced but absent class *X* the compiler will issue the error message “Class *X* not found”.

By default, the "Dynamic compilation" option is selected, which means that the JIT compiler (see 11.5) will search and compile the class *X* at run time if it will be actually requested by your application. Note that

`java.lang.NoClassDefFoundError` will be thrown if the class will not be found.

See [10.15](#) for details.

Stack size

The default thread stack growth limit in a JET-compiled application is 900 kilobytes, unless it was overridden using the `-Xss` Java launcher parameter you specified on Step 1.

You may need to increase the stack limit if your program employs highly recursive algorithms or the "allocating objects on the stack" optimization (see [4.8.1](#)) results in high consumption of stack memory.

You reduce it if your program needs to run lots of threads simultaneously (see [10.4.2](#) for more information.)

To specify the maximum thread stack size for your application, type it into the **Maximum stack size** field or drag the slider below. You may use suffixes "K" and "M" to denote kilobytes and megabytes respectively. The value you enter is rounded up to a multiple of 4 KB.

Note: If the entered value is too small or too large, the warning icon is displayed next to the field until you correct the value appropriately.

Additional options

The **Additional options** pane lets you explicitly specify command line options and equations (see [16.9](#)) of the JET compiler without editing the project file by hand. To do that, type the desired settings into the entry field, one on a line, for example:

```
DISABLESTACKTRACE+  
INLINETOLIMIT=200
```

4.9 Step 5: Specifying target settings

On this step, you specify settings for the executable that will be produced by the compiler.

4.9.1 Executable settings

In the **Executable settings** pane, you specify the file name of the resulting executable and enable encryption of the application data.

Executable name

Type or select the desired name of the executable in the **Executable Name** combobox.

Application data protection

If you do not wish constant data, such as reflection info, Java string literals and packed resource files, to be visible in the resulting executable, you may enable data encryption by checking the **Encrypt strings and resources** box. For more details on data protection, refer to section [5.2](#).

4.9.2 Runtime selection

In the **Runtime** pane, you may select a particular version of the JET Runtime that will serve the compiled executable. Available versions of the Runtime are optimized for certain hardware and application types.

See [10.2](#) for a detailed description of the JET Runtime flavors.

4.9.3 Creating trial versions

You can create a trial version of your Java application that will expire in a specified number of days. As the evaluation period is over, the application no longer starts and shows the expiration message you specified when compiling it.

A detailed description of the functionality is given in section [10.9](#).

Note: this feature is not available in Standard Edition of Excelsior JET.

4.9.4 Stack trace support

If your application requires printing stack trace of Java exceptions thrown, select the desired stack trace option from the **Stack trace support** combo box. The options are trade-off between availability of full stack trace information, such as line numbers, on the one hand, and application performance and the executable size on the other hand. For details, see [10.11](#).

4.9.5 Multi-app executable

If you need to compile more than one application into a single executable, check the **Multi-app executable** box to create the executable that may have multiple entry points.

When starting such an executable, you will be able to select the application to be launched using the command line arguments. This way you may also specify Java properties and VM options at launch time. For more details, refer to section [10.7](#).

4.10 Step 6: Selecting application appearance

On this page, you may also customize appearance of your application by assigning it a splash screen.

Note: if you are creating a project for an Eclipse RCP application, this page simply shows the properties. You cannot change them from within the Control Panel. Refer to section [7.2.6](#) for more details.

Note: if you are creating a project for a Tomcat Web application, this page is not available.

4.10.1 Splash screen

In this pane, you specify settings for a splash screen to be displayed at the very startup of the resulting executable. A detailed description of each control is given in section [10.10.1](#).

4.11 Step 7: Compiling your application

On this step, Excelsior JET Optimizer compiles your application to native code and produces a conventional Linux executable. You may invoke the compiler right from the JET Control Panel, optionally enabling the Startup Accelerator (see [4.11.1](#)) integrated with the whole build process. Also, when building your application from the Control Panel you may get assistance in correcting possible project inconsistencies.

Another build option is to create a script that invokes the `jc` command-line compiler (see Chapter [16](#)) against the current project, which is suitable for automated builds.

If you compile a newly created project, the Control Panel prompts you to save the project first. Click **Save**, a conventional filechooser dialog will appear. Use it to specify the directory and name for the project. The default extension `.prj` is added to the file name automatically.

Note: during compilation the JET Optimizer will store Project Data Base (see [16.5](#)) into the project directory.

4.11.1 Configuring Startup Accelerator

The Startup Accelerator (see [14.2.2](#)) improves the startup time of executables compiled with Excelsior JET. Thanks to its seamless integration with the Control Panel, it is very easy to use. Basically, the Control Panel automatically runs the compiled application immediately after build, collects the necessary profile information and hard-wires it into the executable just created. The JET Runtime will then use the information to reduce the application startup time.

You configure the Startup Accelerator in the respective pane at the top of this page. The **Run and profile after build** checkbox enables Startup Accelerator. Clear the box if you do not want the Control Panel to run the application and collect its profile after build. Otherwise, you may specify the duration of the profiling session in seconds in the field **Profiling timeout**. As the specified period has elapsed, the profiling stops and the running application is automatically terminated, so ensure the timeout value is large enough to capture the application startup. On the other hand, you may close the application manually if the profiling timeout proves to be excessively long.

Your application may require command line arguments to run. If this is the case, type them in the **Application arguments** field. For example, if your application opens files of specific types (associated with certain file extensions), you may specify a path to some input file as the application argument.

Finally, the **Enable profiling in automated builds** box (not set by default) controls if the application will be run and profiled automatically when building this project in the unattended mode with the command line JET compiler. For more information about this option, consult section [14.3.2](#).

4.11.2 Building the executable

To start building the executable, click **Build**. If you need to do a full rebuild (see [16.6](#)) of the project, select **Build/Rebuild All** menu item.

The Control Panel first checks the project for consistency (see [4.11.6](#)). If there are no consistency issues, it invokes the compiler. The compilation dialog appears where the status area and the progress indicator are displayed to inform you about the build progress. If you need to view more detailed info about compilation, click **Show log** on this dialog and the compiler's output will be displayed under the **Filtered log** and **Full log** tabs⁹.

4.11.3 Running compiled application

If everything goes well, the JET Control Panel runs the compiled application for a fixed time interval immediately after build to collect a startup profile¹⁰ and automatically terminates it.

Then, a message box appears to inform you that the build process is over and you may now run the compiled application to test how it works. You may do that by clicking **Run** on this box or by directly launching the executable from the application directory.

If the build process is not completed due to compile-time issues, such as low memory error, the Control Panel assists you in fixing them as described in section [4.11.7](#).

4.11.4 Creating build script

If you want to run the JET Optimizer outside the Control Panel (e.g. to make more memory available for compilation or to create the executable using a makefile or a build script), click **Create build script**. The Control Panel first checks the project for consistency (see [4.11.6](#)). If there are no consistency flaws, it will create shell script `build_target-name` in the project directory you specified on Step 5 (see [4.9.1](#)). Use the script to build the application's executable without invoking JET Control Panel.

Additionally, `run_target-name` script will be created in the project directory. The script defines the `JETVMPROP` environment variable that you may use to experiment with the

⁹Also, the full compilation log will be saved in the project directory with the name of the project and the extension `.log`.

¹⁰ Unless you disabled the Startup Accelerator (see [4.11.1](#)).

JET Runtime settings (see 10.5.3) without rebuilding the executable. For example, you may tweak parameters of the memory manager (see 10.6) to check how that affects performance and memory usage of your application.

4.11.5 Packaging your application

This is actually the final step, on which you prepare an installation that includes the compiled executable and any files it may need for execution. The resulting package can be then distributed and installed on other systems.

This functionality is not integrated into JET Control Panel. Clicking on **Package**, you close the Control Panel and invokes the JetPackII deployment wizard (see Chapter 6). If this is the first time you create an installation package for the executable, it will be automatically added to the list of package files. Otherwise, the JetPackII project previously created for the executable will be automatically opened.

4.11.6 Fixing project inconsistencies

When you click **Build** or **Create build script**, the JET Control Panel first checks the project for consistency. If it detects a problem, one of the following messages is displayed:

Save as dialog

Reason: You did not save a newly created project and did not specify the project directory.

Resolution: Do one of the following:

- Select the desired project directory, type project file name and click **Save**. This will set the **Project Directory** and **Executable Name** fields on the **Target** page (see 4.9.1).
- Click **Cancel**, return to the **Target** page and specify project directory and executable name as described in 4.9.1.

Main class is not specified

Reason: You chose to build an executable, but did not specify the main class.

Resolution: Click **OK** button on the error dialog, the **Main class selection** dialog will appear. If you indeed want to build an executable, select the main class from the dialog and click **OK**. Otherwise, click **Cancel** button and select **shared library** from the **Application** menu.

Compiler heap is too large

Reason: Value of the compiler heap setting is above the size of RAM installed on your machine. Starting the build may lead to excessive memory paging and significant slowdown of the compilation process.

Resolution:

1. Close all applications you do not need at the moment.

2. Set compiler heap size to adaptive (zero value) or to 70–80% of the RAM size (displayed to the left in this dialog.)
3. Click **Proceed**

Low physical memory

Reason: The size of RAM installed on your machine does not meet the Excelsior JET system requirements (see 2.1). As a result, the compiler is unable to work with the fixed heap size used by default.

Resolution: Choose one of the following options:

- Click **Proceed** to force the compiler to work in the Adaptive heap size mode.
- Click **Cancel**, save the project, exit the Control Panel and manually correct the **COMPILERHEAP** setting in the project file. The recommended value is 70–80% of the RAM size.
- Save the project and build it on another machine.

4.11.7 Correcting build errors

Errors detected during build, if any, are displayed in red in the **Filtered Log** area. The JET Control Panel may assist you in fixing some typical errors caused by misconfiguration or bytecode inconsistencies (see 4.8.4).

To automatically fix such an error, select it in the **Filtered Log** area, and click the **Fix it!** button displayed below. If the button does not appear when you select an error, it cannot be fixed automatically. In this case, the error message contains some tips on finding out what caused the error and how to fix it.

Chapter 5

Intellectual property protection

It is known that Java class files are easy to reverse-engineer. The same properties that enable Java applications to be run anywhere make reverse translation straightforward. There are a number of Java decompilers on the market that produce amazingly readable source code. As a result, distribution of the application classes makes code theft and tampering easy, which in turn may cause stealing of *user* data and provoke user identity theft. Reverse translation has been a common headache for the Java application vendors who, for any reason, do not plan to open their source code.

On the contrary, the optimized native code produced by the JET Optimizer is as hard to reverse engineer as the code generated by highly optimizing C++ compilers. The understanding of your code will become a much more sophisticated and time-consuming task, so stealing your algorithms and know-how as well as tampering with your code will be much more difficult and expensive.

Note, however, that you may still need to employ a third party solution, such as a license manager or USB dongles, to prevent *illegal use* of your application.

5.1 Code protection

Excelsior JET compiles your class files down to optimized native code and creates a monolithic binary executable. As a result, you no longer need to ship the application classes to your end users.

As compared to bytecode obfuscators, Excelsior JET provides a higher level of IP protection, because the reverse translation of optimized native code is a very hard task and it cannot be automated like Java bytecode decompilation. Here are the reasons why:

1. Decompiling a native executable, also known as disassembling, produces a large listing full of mnemonics of low-level CPU instructions. By simply browsing such a listing, there is no way to determine where the code for a particular method is located.
2. Depending on other adjacent statements, a Java source code statement may appear in the listing in the form of a dozen different native code patterns.

3. During compilation, the JET Optimizer relocates pieces of code within a method. Furthermore, the compiler often inserts copies of methods into the places from where they are called and then modifies the inserted code.

Although an optimized method is functionally equivalent to the original one, the resulting native code is quite distant from the initial source code which therefore cannot be easily restored.

5.1.1 Enabling code protection

When compiling a Java application, the JET Optimizer processes each jar file and directory listed in the classpath and, depending on the project settings, either compiles class files to native code or leaves them in the bytecode form.¹

For each classpath entry, the Optimizer supports the following protection modes:

1. **All classes:** this option guarantees that, independently on other project settings, *all methods of all classes* from this classpath entry will be compiled to native code and, therefore, protected from reverse engineering.
2. **Not required:** in this mode, the Optimizer *may* leave some classes in the bytecode form depending on other project settings.

If code protection for certain classpath entries is not important, you may set the "not required" mode for them. For example, there is no need to protect classes from open source libraries, such as Apache Commons.

You may select the desired code protection mode for any classpath entry as follows:

Open your project with the JET Control Panel and go to the **Classpath** page. Each row of the classpath grid (see 4.7.2) corresponds to a single classpath entry — directory or jar file. The column named **Protect** contains the protection mode for each entry. Click the cell, and select the appropriate value from the drop-down list.

If you edit project files manually, use the **PROTECT** equation to control code protection. Refer to the JC reference chapter (see 16.9.4) for more details.

5.2 Data protection

Not available in the Standard Edition:

One difference between the executables created by the JET Optimizer and C/C++ compilers is that the former must contain Java reflection information for all classes. Although inspecting the names of classes, method, and fields does *not* help a malicious programmer to find the code of the respective methods, such exposure may be undesirable.

Moreover, while the JET Optimizer transforms Java bytecode to native code, *constant data*, such as Java string literals, appear verbatim in the resulting executable. This may be a potential weak point for IP protection.

¹The classes will be handled by the JIT compiler if the application tries to load them at run time.

Finally, the jar files may contain not only classes but also application data, for example, images, media files, various text and property files, XML configuration files, etc.

To protect the reflection information, strings, and program data, you may enable *data encryption* when compiling your application. As a result, the optimized executable will not include the resource files and strings in the original form. Decryption is performed on demand at application run time.

Unlike name obfuscation, the technique does *not* compromise the correctness of execution if the application heavily uses Java Reflection API and/or JNI. This is a unique feature that cannot be found in Java bytecode obfuscators because Excelsior JET provides data protection at the JVM level.

5.2.1 Enabling data protection

You enable data protection on the page **Target** of the JET Control Panel: check the **Encrypt strings and resources** box in the **Executable settings** pane. This setting affects all compiled classes and classpath entries.

If you edit project files manually, use the **CRYPTSEED** equation to enable data protection. Refer to the JC reference chapter (see [16.9.4](#)) for more details.

Note: resource files are protected only if they are placed in a jar file and the jar file is packed into the executable in the auto-detect mode (default for jar files). For more details, consult section [10.13](#). If the resource files you need to protect are located in a directory listed in the classpath, you have to place the files in a jar file and add it to the classpath before compilation.

Enabling data protection increases neither compilation time nor the size of the resulting executable. It may slightly degrade application performance and start up time, though in our tests the impact was minimal.

Chapter 6

Deployment automation

6.1 Intro

Excelsior JET comes with an Installation Toolkit, a complete solution for the deployment of applications which contain binary components (executables, shared libraries) produced by the JET Optimizer.

The toolkit includes:

- JetPackII, a graphical tool that helps you create installation packages
- A specialized version of Excelsior Installer, a fairly simple command-line installer
- A set of utilities that automate the process of rebuilding the installation packages created in JetPackII

It empowers you to create new installations and updates, and deploy them using either Excelsior Installer or a third-party setup authoring tool. You may also create a self-contained directory that includes your application's files and can be simply copied to the target systems without any setup actions, burned to CD, or transferred to a USB stick or flash memory card.

Using the toolkit, you can create *complete* installations without deployment dependencies. In particular, installation of the Java Runtime Environment (JRE) on target systems is not required.

6.2 Excelsior deployment technology

6.2.1 JetPackII

The JetPackII graphical tool, designed in the spirit of the JET Control Panel, is responsible for all stages of creating new installations and updates. Its wizard-like interface guides you through the process of preparing the installation package step by step.

When creating a new package, you define the set of JET-compiled executables and auxiliary files (resources, documentation, samples, etc.) which your application needs to work. For each executable, you may configure the classpath and Java system properties (see e.g. 6.17) to be set during installation. As a result, your end users receive self-installing archives that always create clickable executables on target machines. This way, the need to use shell scripts for application startup is greatly reduced.

You can also create updates to previously created installations. This feature allows you to issue upgrades and patches for your products. JetPackII assists you in tracking the differences between the original and new packages for the update package to include only added and modified files.

Then, you may perform a trial run before packaging your application. That is a testing procedure that ensures the completeness of your package. It simulates execution of your application deployed to a target system. The package contents are copied to a trial run directory, and the applications are run as if Excelsior JET was not installed on your computer.

In the end, you get either a self-extracting Excelsior Installer archive, a self-contained directory, or a set of files and instructions to create an installation package using a third-party tool.

6.2.2 Excelsior Installation Toolkit

Excelsior Installation Toolkit is a part of Excelsior JET that enables you to create self-extracting installation packages without using any third-party tools.

Excelsior Installation Toolkit is a set of utilities seamlessly integrated with the JetPackII wizard. Using it, you immediately get installation packages ready for deployment on the final step of JetPackII.

Here is the list of features available in Excelsior Installation Toolkit:

- Excelsior Installer archives ready for deployment (new installation packages and updates);
- interactive installation wizard (command line version only);
- installation settings under vendor's control:
 - default installation directory;
 - license to be shown and accepted by end users;
- cumulative uninstallation (uninstalling the initial package along with the updates possibly installed later);
- customization:
 - pre- and post-install callback shared libraries, supplied by you to add extra logic to the install/uninstall process through a well-defined functional interface;

- after-installation runnable, supplied by you to perform post-install actions on end-user systems.

Note: Not all features are available in the Standard Edition.

6.3 Before You Begin

Suppose you have compiled your application with Excelsior JET and the compiled version has successfully passed the QA on your system. Now you need to prepare it for distribution to end users.

Ideally, you should write down the answers to the following questions before launching JetPackII:

What resource files your application needs?

Resource files are image files, text property files, HTML files, etc., that applications load from the classpath by means of `java.lang.Class.getResource()` and similar methods.

Are the resources included in the jar files you compiled?

Resources from the jar files were automatically packed into the executable at compilation time so you need to do nothing to make them available to the applications deployed to end user systems. "Standalone" resources located in directories in the local file system should be added to the package manually when creating the installation as described in [6.9](#).

What other files have to be distributed along with your application?

These may be native method libraries, configuration files, samples, manuals, non-Java executables, etc.

What is the desired directory structure of your installation package?

6.4 Starting JetPackII

To launch JetPackII, type

```
JetPackII
```

at the command prompt and press **Enter**. A splash screen with a "Loading..." progress indicator shall appear. When loading is over, the Welcome Screen is displayed.

In the middle column, click:

- **New Installation** if you want to create a new installation package (see [6.8.1](#));
- **New Eclipse RCP Installation** if you want to prepare an Eclipse RCP application (see [7](#)) for deployment;

- **New Tomcat Installation** if you want to prepare a Tomcat Web applications (see 8) for deployment;
- **Update Package** if you want to create an update (see 6.8.2) to existing installations

6.5 JetPackII Overview

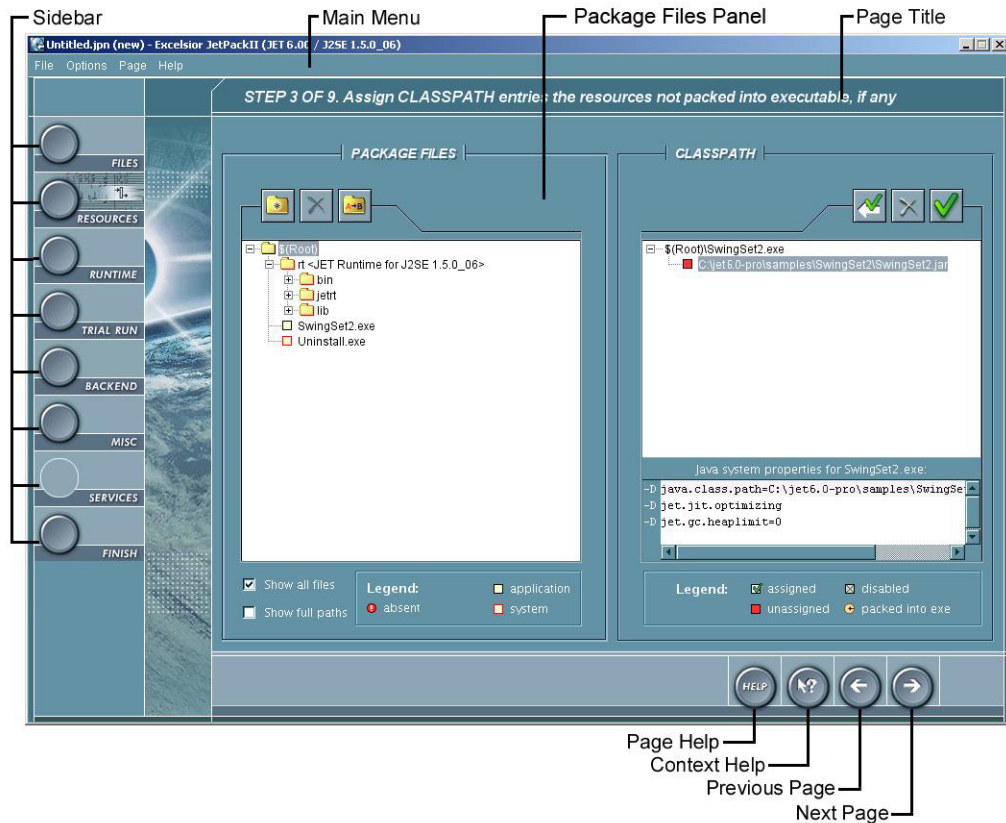


Figure 6.1: Excelsior JetPackII

Like the JET Control Panel (see Chapter 4), JetPackII user interface is built as a series of “pages”, each representing one logical step in the process of your installation package creation. As can be seen on the Figure 6.1, most of page space is occupied by page-specific controls; but there are also some common controls: the sidebar on the left, the main menu at the top, and help/navigation buttons in the lower right corner.

You can navigate through the pages either sequentially, by using the **Next Page / Previous Page** buttons, or directly, by clicking the respective sidebar button or by selecting a page in the **Page** menu. Depending on the context, some pages may be unavailable, with their respective sidebar buttons and main menu items disabled.

Online help is available to assist you at any point. To display help information for the current page, click the **Page Help** button. To get a help on a particular control, click the **Context Help** button. The mouse cursor will change its shape to an arrow with a

question-mark — click the control you wish to learn about, and the respective portion of the online help will be displayed.

6.6 Working With Projects

A JetPackII project consists of a project configuration and a set of files, which together fully determine the installation. A project defines:

- Files to package - the set of directories to be created on a target machine and the set of files to be copied.
- JET Runtime - the exact set of JET Runtime components to be included in the installation package, and their locations within the directory structure.
- Properties of each JET-compiled binary component to be deployed. That includes CLASSPATH settings, and Java system properties, including JET-specific ones.
Note: The pathnames of the actual installation directories may be substituted into the property values during installation. See e.g. 6.17 for the possible use of this feature.
- Installation settings (if you use Excelsior Installer) - registry keys, system Start Menu entries, Desktop shortcuts, file associations to be created, and also other actions to be performed on the target machine.

This information is stored in a *JetPackII project file* that you create as described later in this chapter. JetPackII project files have extensions `.jpn`.

6.6.1 Creating a new project

You may create a new JetPackII project by clicking **New Installation (New Tomcat Installation or New Eclipse RCP Installation)** on the Welcome Screen or by selecting those options in the **File/New Project** menu. If there is already an opened project, JetPackII will prompt you to save it first.

6.6.2 Opening a project

You may open a previously created project by clicking the **Open** icon on the Welcome Screen or by selecting the **File/Open** menu item. A conventional file dialog will appear; use it to browse for the JetPackII project file you want to open or select the desired project in the **File name** combo box.

Later you may quickly open an existing project by selecting it in the **Recent Projects** column of the Welcome Screen or using the **File/Reopen** menu item.

6.6.3 Saving a project

You may save a project by selecting the **File/Save** item from the main menu. When you save a just created project for the first time, a conventional file dialog is displayed; use it to specify the desired pathname for your project file. The default extension `.jpn` will be added to the file name automatically.

6.6.4 Saving a project under a different name

You may save the current project with a different name or in another directory by selecting the **File/Save As** item from the main menu. A conventional file dialog is displayed; use it to specify a new directory and/or a new name for your project file.

6.6.5 Closing a project

You may close the current project by selecting the **File/Close Project** item from the main menu. This brings up the JetPackII Welcome Screen. If the current project has been modified, JetPackII will prompt you to save it first.

6.7 Package Files Panel




The left part of JetPackII pages **Files** and **Resources** is occupied by the *package files panel*¹. The panel displays the installation package you are working on as a tree. Leaf nodes represent the files included in the package. Non-leaf nodes represent the directory structure of the installation. Top-level nodes represent abstract *root directories*. Each root directory is a destination directory for the installation (typically installations will have a single root, but you may need to have multiple root directories in certain circumstances.) The actual location of root directories in the end user's file system becomes known only at install time. Each root has a special symbolic name that starts with `$`, e.g. `$(ROOT)`. A root directory may be referenced by its name from various places, such as command line arguments for trial runs and shortcuts, values of Java system properties, etc.

An installation package contains one or more *system* nodes (see 6.7.1), added automatically by JetPackII. Only a restricted set of operations can be performed on system nodes.


In a complete installation package, directories and files have the respective icons displayed before their names. See also 6.7.1.

In an update package (see 6.8.2), the set of icons is richer:

¹If you are preparing a Tomcat or Eclipse RCP installation, the panel is displayed in the cenral area of the page **Files**

-  - “*new*”: directories and files that were not present in the original package; they will be created and copied during installation of the update;
-  - “*updated*”: directories and files that were present in the original package and will be changed during installation of the update, that is, directories containing new or changed files, and files that were changed and will be replaced during update installation.
-  - “*identical*”: directories and files that were parts of the original package, but were not altered since the original package was created; they are not included into the update package;

The following icon may also appear next to the name of a non-root node:

-  - “*absent*”: marks files that no longer exist or may not be used, and directories that contain such files

6.7.1 System files

- A thin red border surrounding a node icon indicates that the respective file or directory is “*system*”, i.e. was automatically added to the package by JetPackII.

There are the following types of system files:

- JET Runtime files:
 - JIT compilers
 - shared libraries with precompiled Java SE API classes (omitted if the applications being packaged were compiled using the Global Optimizer (see Chapter 13))
 - shared libraries with native methods for the Java SE API
 - Various resource files used by the Java SE runtime: fonts, property files, color profiles, MIDI soundbank, etc.
- JIT Cache files (see 6.9).

You may not remove any of the system files from the package, except for JIT Cache files. Possible operations on system files are described in sections 6.11 and 6.13.3.

6.7.2 Controls

The following buttons appear above the tree view of the installation package files²:

²The buttons are not displayed if you are packaging a Tomcat or Eclipse RCP application. In this case, all the necessary files are added to the package automatically and the content cannot be modified from within JetPackII



Create a new directory in the installation package.



Remove the selected node from the installation.



Rename the selected node.



Create a new root in the installation. This button is available only if the **Multiple roots** option is turned on via the **Options** item from the main menu.

Note: Multiple root installations (see 6.13.3) can be created only using a third-party installer and are not supported in the Standard Edition.

You may also control the view using the following checkboxes:

The **Show all files** check box can be used to toggle the filtering of installation elements unnecessary for the current step.

The **Show full path** check box can be used to show the locations of the files included into installation on your system.

6.8 Selecting installation type

On this step you choose the type of the installation package that you want to prepare — either a new installation or an update to already existing one.

6.8.1 Creating a new installation

Click **New Installation** on the Welcome Screen to start preparing a complete installation package for your application. If you package a Tomcat or Eclipse RCP application³ compiled with Excelsior JET, click **New Tomcat Installation** or **New Eclipse RCP Installation**, respectively.

The **Files** (see 6.9) page will display immediately.

6.8.2 Creating an update

Using JetPackII, you may create updates or patches for the existing installations of your product.

To create such an update, you need the project file for the original package that was saved as "updatable" (see 6.16). The project file has the extension `.jpu`. Click **Update Package** on the Welcome Screen of JetPackII and select the updatable project file using a conventional file dialog. You may also quickly select the desired updatable project among those displayed in the middle column of the Welcome Screen.

JetPackII then assists you in tracking differences between the original installation package and the files currently available on your system to include only added and modified files in the update. See 6.9.2 for details.

³For more information on deployment of Tomcat and Eclipse RCP applications, refer to Chapters 8 and 7, respectively.

See also [6.20](#).

6.9 Step 1: Adding files to package


On this step you can add application files to your installation package.

Note: if you are preparing a Tomcat or Eclipse RCP installation, refer to sections [8.3.1](#) or [7.3.1](#), respectively.

The page displays the package files panel on the left and the embedded file chooser on the right. The package files panel (see [6.7](#)) represents the structure of the installation on a target machine. The embedded file chooser allows you to select the files to be added to the installation. You must add at least one JET-compiled component (executable or shared library) to create an installation.


6.9.1 Adding files using filechooser

To add file(s) to the installation, do the following:

1. Select file(s) in the embedded file chooser on the right.
2. Drag selected file(s) to the package files panel or click  button.

Note that files are not actually copied — the project being created contains just links to file locations. You can check that all referenced files do actually exist on disk by selecting the **File/Refresh** menu item. In general, you should avoid physical deletion of files already added to the installation.

Adding a directory results in inclusion of its entire contents with subdirectories and respective (sub)directories are created in the package files panel. Note that adding large directories may take a while.

To create an empty directory in the installation, use  button.

When you add a JET-compiled executable to the package, JetPackII recognizes it and automatically adds a system folder named `rt`. The folder contains the JET Runtime components required by the added executable⁴. You may rename the JET Runtime folder or move it within the tree using drag-n-drop.

If the added executable has an associated JIT cache, it can be automatically added to the installation as well(see [6.9.3](#).)

The restrictions on what files are allowed within a single installation are as follows:

- One package may not include JET-compiled executables that use different J2SE versions.
- One package may not include executables created by different versions of Excelsior JET.

⁴You may also add some optional components of the JET Runtime on Step 3 (see [6.11](#)).

When creating an update package (see 6.8.2), you can replace files included in the previous installation by dropping a newer file version over the older one. Such files will be added to the package to replace the previous copies of the files when updating a client installation of the application.

6.9.2 Adding files using the Difference Tracker




Another way to include files in an update package (see 6.8.2) is using the Difference Tracker, which enables you to:

- check the differences between the content of the original package and respective files/directories on your system
- quickly add the necessary files/directories to the update package

The Difference Tracker dialog is invoked automatically when you start preparing an update package. You may also invoke it manually later by selecting the **Options/View and commit changes** item of the main menu.

The dialog contains the **Summary of changes** panel that displays files and directories similarly to the Packages Files panel (see 6.7).

The tree highlights the differences between the content of the original package and files/directories located on the disk. To the left of each node element, one of the following icons appears:

-  - “*new*”: files and directories not included in the original package;
-  - “*changed*”: modified files and directories that contain new and/or modified files.
-  - “*identical*”: files and directories identical to those included in the original package;

Note that the Difference Tracker treats files as identical if they have the same name and size, and meet either comparison criteria:

- *by date* - files have the same time of the last modification
- *by content* - files have identical binary content

Comparison by date works faster but it is less accurate. You may select the desired comparison method by selecting the radio button in the **Comparison method** panel on the right.

The tree nodes displayed in the **Summary of changes** panel have checkboxes. Select the checkbox to mark the respective the file or directory (including all its content) as "to be added" to the update package.

Once you have selected the desired files and/or directories, click **Ok** at the bottom of the dialog to commit the selection and include the items in the update. For your convenience,

the checkboxes of the changed files and directories have the selected state by default so you may add them at once by simply clicking **Ok**.

The checkboxes may have one of the following states:

- not selected and not grayed — the node is not included in the package. You may select it.
- not selected and grayed (disabled) — the node cannot be added to the update because it was not changed.
- selected and not grayed — the (new or changed) component will be added to the update package after you click **Ok** in the dialog.
- selected and grayed (disabled) — the (new or changed) component has already been added to the update package.

6.9.3 Packaging JIT cache

If JetPackII detects that the added executable has an associated JIT cache, a dialog is displayed asking whether you want to add that cache to the package. Normally, you would only add optimized caches (see [11.6](#)).

An important example of such executable is a custom launcher created by the `xjava` utility (see [11.7](#)) in the `-Xcompile` operation mode.

Note: Packaging JIT cache only allows you to automatically add the cache descriptor and shared libraries that contain the cached code. You still have to ensure that the original class files will be available to the JIT compiler on the target system.

If the dynamically compiled classes are contained in jar files and/or directories not listed in the classpath, you simply add them to the installation. Note that you should place the class containers at appropriate location to enable the used custom classloaders to find the classes.

If the jar files and/or directories are classpath entries, you can do one of the following:

- Pack entire jars that contain dynamically loaded classes into the executable. See [10.13](#) for details.
- Add those jars and/or directories to the package and assign their classpath entries as described in [6.10](#).

You can enable or disable automatic inclusion of JIT cache for any executable later by selecting **Options/Packaging JIT Cache** menu item.

6.9.4 Caching of JITed classes on end user system

When deployed application runs on end user system, it may occasionally require JIT compilation to handle dynamically loaded classes.

If you added a JIT cache to the project, the JET Runtime automatically enables the caching mode — each time a new class is JITed *on end user system*, its compiled code is saved to the JIT cache directory, and reused on the next run, thus improving overall application performance.

But for certain types of applications, JIT caching makes no sense. Consider an application that constantly generates new classes, e.g. Hibernate thinks. Caching these classes is useless, because storing classes in the cache takes time and disk space but they are never reused.

There is an easy way to detect if your application belongs to this type. Run the application several times, and check the JIT cache directory size after each run. If the directory size keeps growing and does not stabilize after several run sessions, then new classes are continuously generated, and disabling the caching of JITed classes may improve application performance.

You can disable the JIT caching *on end user system* by removing the runtime property `jet.jit.cache` on Step 2 (see 6.10.2).

6.10 Step 2: Configuring Resources And System Properties

Note: if you are preparing a Tomcat or Eclipse RCP installation, this step is not required and the respective page of JetPackII is not available.

Each JET-compiled executable added to the installation has an associated classpath, that is, a list of directories and jar/zip files where the running application will look for resources, such as images, media, property files, etc. On this step you set up the classpath for each executable so that the classpath entries refer to directories and jar/zip files *from the installation package*, not from the local file system. Additionally, you may edit Java system properties for the executable(s)⁵.

The left side of this page contains the **Package files** panel (see 6.7) that shows the contents of the installation being prepared. The **Classpath** panel to the right displays the list of the executables and their classpath entries that are shown as subitems of the executable names.

Each classpath entry name appears with a small icon displayed to the left. The icons indicate *if the entry needs to be assigned*, that is, if you should assign a directory or jar/zip file from the package to the entry. The following icons are used:

⁵Note that when creating an update package, you can set up the classpath and properties only for the files that are either “new” or “updated”. These setting cannot be edited for files marked as “identical” (see 6.7).

- - the entry needs to be assigned. The full path to the respective directory or jar/zip file *on your system* is displayed as the entry's name.
- ☑ - the entry has been assigned. The path relative to a package root is displayed as the entry's name.
- ⊕ - the entry was packed into the executable (see 4.7.2) during compilation. Such entries are also considered assigned and displayed just for reference.
- ☒ - the entry is disabled (by the user) because it does not contain any resource files

If all classpath entries appear to be assigned or disabled (in other words, no entry has the icon ■), Once there are no more unassigned classpath entries in the installation package, you may proceed to the next step (see 6.11). You may also wish to define additional Java system properties (see 6.10.2) before leaving the **Resources** page.

6.10.1 Setting the classpath entries

First of all, the list of classpath entries specified when compiling the executable (see 4.7) cannot be changed in JetPackII. That is, you cannot reorder the existing entries or add a new entry to the list. What you can do, (in fact, what you must do), is specify a particular directory or jar/zip file from the installation package for each entry marked as unassigned. Another option is to disable such an entry if you are sure that it does not contain any resources.

Selecting an item from the package

To assign a classpath entry the respective jar/zip file or directory from the package, do the following:

1. Right-click the entry name and choose **Assign by selecting from package** from the pop-up menu; or, alternatively, select the entry name and click the button



Assign by selecting from package

A dialog displaying a package files tree will appear. Note that it contains only items of the appropriate type. For example, if the classpath entry is a directory, only directories from the package will be displayed.

2. Select the desired item from the package and click **Ok**

Adding an item to the package

It may happen that the package does not contain the required item because you forgot to add it on Step 1. If so, you can add such an item to the package and assign its classpath entry at once without going back to the previous page. For that, do one of the following:

- Right-click the entry name and choose **Assign by adding to package** from the pop-up menu; or

- Select the entry name and click the button



Assign by adding to package

Note: If the classpath entry is a directory, it will be added to the package *as whole*, including all contained files and subdirectories. That is why JetPackII displays a message box with a request for confirmation when adding a directory. After you have added a directory, it is recommended to inspect its contents using the **Package files** panel (see 6.7) to the left and remove unnecessary files, if any.

Disabling a classpath entry


Finally, if you are sure that certain classpath entries do not contain any resources, you can disable such entries. That is the case, for example, if the entry contains only class files that were all compiled into the executable.

To disable a classpath entry, do one of the following:

- Right-click the entry name and choose **Disable** from the pop-up menu; or
- Select the entry name and click the button



Disable classpath entry

If you want to correct the classpath settings, you can re-assign or disable any classpath entry except for those packed into the executable (displayed with the icon )

6.10.2 Editing Java system properties

In the lower right panel you can edit the system properties for each Java application added to the installation package. The properties will receive their values upon application startup⁶.

To set a Java system, property for an executable added to the package, do the following:

1. Select the desired executable from the list of executables above.
2. Type in a name and, optionally, a value of a property in the lower right panel as *property-name* [= *property-value*]

Property names and values may contain references to root directories from the package files in the form $\$(root-name)$. For example, suppose the package contains a root directory called $\$(Root)$ that includes a subdirectory `AppFiles`. You may define the following property

⁶As you know, a Java application can use the `java.lang.System.getProperty()` method to access Java system properties.

```
my.app.files.dir=$(Root)/AppFiles
```

Then you prepare a package and install it into a directory on the target system. Upon application startup, the JET Runtime replaces `$(Root)` with the absolute path of the installation directory. Thus, when the installed application inquires the value of the `my.app.files.dir` property, it gets the full path to the `AppFiles` directory on the target system.

Note: The `java.class.path` system property is set by JetPackII automatically. Its value cannot be edited manually and is displayed at the top of the property list for reference.

6.11 Step 3: Configuring the JET Runtime

On this page, you may:

- add/remove optional components of the JET Runtime to/from the installation package being created
- select the Java SE API components that are not used by your application and can be excluded from the installation package
- choose a compression method to reduce disk footprint of the installed application⁷

6.11.1 Selecting optional components

The upper left panel displays a tree of the JET Runtime components that may be optionally excluded from the installation package without compromising compatibility.

The tree branches and certain components have checkboxes displayed to the left of their names. The checkbox on a tree branch allows you to include all its components at once, while the checkbox on a particular component controls inclusion of that component only. A checkbox has one of the following states:

- not selected and not grayed — the component is not included in the package. You can add this component to the installation by selecting the checkbox.
- not selected and grayed (disabled) — the component cannot be added to the installation.
- selected and not grayed — the component has been already added to the package. You can remove this component from the installation by unselecting the checkbox.
- selected and grayed (disabled) — JetPackII has automatically detected that this component is in use. You cannot remove it from the installation.

⁷Distinguish disk footprint from download size. Disk footprint is the amount of disk space occupied by the installed application whereas download size is the size of the resulting installer prepared for deployment to end user systems.

The number that appears next to a component's name is the amount of disk space (in KBytes or MBytes) occupied by the component's files on end user systems.

Inclusion of optional components also increases the size of the installation bundle though the download size overhead will be typically smaller due to compression implemented in the used installer.

Optional components

The list of optional components displayed in the left panel and their meaning are as follows:

- **JIT** — the Optimizing JIT compiler and JIT Caching engine. By removing these components you force using the non-optimizing (fast) JIT for dynamic compilation and disable JIT caching. For more details, refer to Chapter Mixed Compilation Model (see [11](#)).
- **Additional locales and charsets** — the locales and extended character encoding sets that you may optionally include into the package. By selecting particular checkboxes, e.g. **Japanese Locale** and **Extended Japanese Charsets**, you add to the package the locales and encoding sets that may potentially be in use in the regions where you wish to distribute your application.

By selecting the **Additional locales and charsets** checkbox, you add all locales at once. Note that inclusion of all locales increases the size of the installation.

- **AWT native libraries for Motif and X11** — AWT native method libraries. They can be excluded if your application operates in so called *headless mode*, that is, it does not use the AWT/Swing API to draw on the screen.

Note that a violation of this requirement will result in throwing `java.awt.HeadlessException` at run time.

- **Additional fonts** — all `.ttf` font files from the `lib/fonts` subdirectory of the JET Runtime except the `LucidaSansRegular.ttf` font file which is always included in the package. If your application does not make use of those additional fonts, you can exclude them from the installation bundle.
- **Audio files** — the `lib/audio/soundbank.gm` file. This MIDI soundbank is necessary for MIDI playback, so it may be added to the installation package at your discretion.
- **Runtime utilities** — the standard Java utilities that your application may require at run time. For example, the `rmiregistry` utility (Java Remote Object Registry) may be necessary for RMI applications. Selecting the checkbox, you add all the utilities to the package. By technical reasons, it is impossible to include the utilities if all executables from the package are compiled using the Global Optimizer (see Chapter [13](#)). In that case the checkbox is disabled (grayed). This limitation is to be removed in future versions of Excelsior JET.

- **Java SE API class files** — the jar files that contain the implementing classes of the Java SE platform API in bytecode form, such as `rt.jar`.

The JET Runtime already includes all those classes in the precompiled form so adding them to the package is required in extremely rare cases.

Even if your application uses the Java technologies that rely on compilation of Java *source* files at run time, such as Java Server Pages (JSP), no need to add the files to the package.

Note that adding the platform classes to the package substantially increases its size.

6.11.2 Selecting detached components

Excelsior JET features *Java Runtime Slim-Down*, a new deployment model that enables you to significantly reduce the download size of your Java applications.

The key idea is to select the components of the Java SE API that are not used by the application, and exclude them from the installation. Such components are called *detached*. For example, if your application does not use any of Swing, AWT, CORBA or, say, JNDI, you can exclude them from the installation package.

The detached components should be placed on a Web server so that the JET Runtime could download them if the deployed application attempts to use any of them at run time. You may find a detailed description of Java Runtime Slim-Down deployment in Chapter 9.

Note that detaching Java SE components is possible only if the executable(s) added to the installation have been compiled with the Global Optimizer enabled (see Chapter 13).

The **Detached components** pane of the **Runtime** page displays the names of the components that you have excluded from the installation. By default, however, all Java SE components are included. Click **Configure** to open a dialog for selecting the components you wish to exclude (detach).

Detached components configuration

The left pane of the dialog displays a tree of all Java SE components that can be detached. The tree branches have checkboxes displayed to the left of the components' names. Enable the checkboxes of those components you want to *exclude* from the installation package.

When building an installation that has detached components, JetPackII creates two packages:

- installer to be shipped to end users
- detached package (single file)

The detached package should be placed on a Web server so that the JET Runtime could download and unpack it if the deployed application tries to use some of the excluded components during execution.

You should specify the desired base URL for the detached package in the **Base URL** pane on the right. You will choose the filename of the detached package when creating the installation on the **Final** page (see 6.16).

Note: download from the server will occur if and only if the application accesses any of the detached components at run time.

How to identify unused components

Each detachable Java SE component comprises one or more Java packages and, in some cases, native method libraries and resource files. You may inspect the contents of a detachable component by expanding the respective tree node.

JetPackII helps you identify the Java SE components which are not used by your application. Each component name appears with a color tag displayed to the right. The tags indicate *if classes from the component are referenced* by your application or by third-party libraries it uses. The tags mean the following:

- - component is unused. Such components can be detached.
- - component is (partially) used. In general, it is not recommended to detach such components.

Under certain circumstances, you may detach a component marked with the red tag. When you try to exclude a used component, JetPackII opens a warning dialog. Click **Details** to see the list of classes and/or native method libraries which your application references. As JetPackII *always* places the referenced items into the main installation package, only the remaining part is actually detached.

However, care must be taken. If the application references lots of classes from a component, the remaining (unreferenced) classes may also be used implicitly, e.g. via the Reflection API. Placing them to the detached package may result in its downloading at run time. For more details, see Chapter 9.

To ensure that the downloading of the detached package will not occur at run time, simply run and test your application on the page **Trial run** (see 6.12). Upon application exit, JetPackII will display a warning if the detached package has been accessed during execution.

6.11.3 Enabling disk footprint reduction

You enable disk footprint reduction if you need the client installation of your application to occupy less disk space. It is important, for example, when the application files should be burned onto a CD full of other applications and/or data or the application will run off a USB flash drive.

Note that the reduction of disk footprint is possible only if the executable(s) added to the installation have been compiled with the Global Optimizer enabled (see Chapter 13). That is necessary to detect the parts of the JET Runtime that will unlikely be used during execution and compress, them thus minimizing the required disk space.

To enable disk footprint reduction, select "MEDIUM REDUCTION" or "HIGH

REDUCTION" radio button in the **Disk footprint** panel. If you opt for the high reduction, you can also select an on-demand decompression method. Two options are available:

- **decompression to disk** using temporary files that will be automatically deleted at application exit
- **decompression into memory** storing the uncompressed data on the heap. The created structures will be garbage collected as they are no longer needed

Note that on-demand decompression does *not necessarily occur* at application run time. For a detailed description of the reduction technique and available options, see [10.8](#).

Using the disk footprint reduction methods may negatively impact application start-up time and performance. If the disk space occupied by the application files is not an issue, you can disable the reduction by selecting the "NO REDUCTION" radio button in the **Disk footprint** panel.

Note: disk footprint reduction is not available in Standard Edition of Excelsior JET.

6.12 Step 4: Performing a Trial Run

On this step you can perform a trial run of the application to be deployed.

A *trial run* is a testing procedure that ensures completeness of your package. It simulates execution of your application deployed to a target system. The package contents are copied to the specified trial run directory exactly if it was an installation to an end-user system. The fact that Excelsior JET is installed on your computer does **not** affect the correctness of trial run results.

The main purpose of the trial run is to ensure that the application would behave correctly if installed to a typical target system. At this step you can check if all files required by the application are actually included into the package.

Performing trial runs is optional and may be skipped. However, it is strongly recommended to perform a trial run of each JET-compiled executable at least once after you have added it to the installation package.

During a trial run, all files are copied into target directories corresponding to the installation roots, the post-installation processing is performed and the selected executable is started. Note that the system environment is reduced for the running executable, simulating installation on a “clean” target machine.

When you feel testing is over, close your program in a usual way (of course, you should do nothing special if your program stops automatically, e.g. if it is a number-crunching application).

After the program execution has finished, all JET Runtime components that were involved during the trial run are automatically added to the project. You may inspect them on the Runtime page (see [6.11](#)).

6.12.1 Selecting a trial run directory

Trial run directory is the directory in which JetPackII copies all files from the package before a trial run is executed. You may consider it as a test "installation directory" for the package. It may be selected by clicking the **Browse** button. Note that the directory should be empty (you will be warned if it is not), and the respective drive must have enough free space to accommodate the installed version of your application.

6.12.2 Working with trial run configurations

If your installation package contains more than one JET-compiled executable, each of them should be run at least once. Moreover, if your application behaves essentially differently depending on command line arguments and/or user input, you may wish to run the application several times using different scenarios.

Trial run configuration is a "scenario" for a trial run. It specifies what executable is to be run, its working directory and command line arguments. By default, JetPackII creates one trial run configuration for each executable that you add to the package. You can create more configurations, e.g. for running your application with different command line arguments or with a shell script that defines additional settings.

Trial run configurations are enumerated in a list. The color of a configuration reflects its status. A trial run colored red is not performed yet. A trial run colored green is already performed. Certain changes in the project (such as adding/removing files, including a component into the JET Runtime) switch trial run status to "red".

You can add/remove trial run configurations using buttons on the left of the trial runs list:



Add a trial run configuration. After clicking this button, a dialog is displayed showing the list of executables and script files, suitable for trial run. Select a file to be run and click the **OK** button.



Remove the selected trial run configuration.

6.12.3 Configuring a trial run

The selected trial run configuration can be edited using the options panel to the right of the trial run list.

The **Executable** text field shows the executable or script file name in target form, that is, with its name relative to the root directory. It cannot be changed after the trial run configuration is created.

The **Arguments** text field shows command-line parameters that will be passed to this executable or script.

The **Start in** field shows the working directory of the executable or script.

To select a working directory from the package, do the following:

1. Click the **Select** button. A dialog that shows the directory structure of your installation, will appear.

2. Select a directory and click the **OK** button.

The selected trial run can be performed using the **Run** button. After you click it, JetPackII may prompt you to save the current project unless you haven't saved it earlier. Also, if the trial run directory is not empty, you may be prompted to clear its contents before simulating installation. After copying files, JetPackII starts the executable or script specified in the selected trial run configuration. If it is a console application, its console will remain open after the execution is over and you may safely close it then.

6.12.4 Performing all trial runs

After certain changes in the project you may wish to perform all trial runs. The **Run All** button can be used to perform all trial runs, one by one.

6.12.5 Troubleshooting

If the tested executable exits with a non-zero return code, the trial run fails. JetPackII displays the respective error message that includes the executable or script name and return code. You may cope with it as follows:

1. Check that your application always exits with zero return code in the case of normal termination.
2. Make sure the package files are complete, that is, all auxiliary files necessary for your application were added to the package. Often, applications throw an uncaught exception when some files are missing. You may check it in the application's standard output.
3. If your application requires a special environment to run, e.g. some environment variables should be set, you have to prepare a script file that makes the necessary settings and invokes the JET-compiled executable. Then, add the script to the package files and create a trial run configuration specifying the script in the **Executable** field.

6.12.6 Skipping trial runs

If you do not wish to perform trial runs, you may just skip them by checking the **Skip Trial Run** box. Note, however, that it is not recommended. You cannot proceed to any of the subsequent steps until you either perform all trial runs or check the box.

6.13 Step 5: Selecting a Back-End

On this step you choose the *back-end*, or, in other words, the type of the installation you wish to create.

You choose a back-end using the radio buttons displayed on the right side of the page. Note that when creating an update package, you cannot change the back-end because it should be identical to that selected for the original package.

The left side of the page contains the package files panel (see 6.7). It can be used to review the package and to rename or move the *uninstaller executable* that is automatically added if you select Excelsior Installer as the back-end.

The next sections describe the types of installations that you can create using JetPackII.

6.13.1 Excelsior Installer setup

Use the Excelsior Installer setup option to rapidly create an installer executable ready for distribution. The installer executable displays your end-user license agreement, lets user choose the installation folder, and then unpacks the software and JET runtime to the specified folder. It can optionally run your application right after the installation.

The functionality of the installer is extensible so you can add extra actions (see 6.14.4) to customize both installation and uninstallation procedures.

Note: Excelsior Installer setups have much smaller size as compared with other back-end options, if you select the best compressing level on the final Step (see 6.16.)

By default, the installer runs as a command-line tool that guides the user through the installation process step by step. Unattended (batch mode) installation/uninstallation are also supported. The Excelsior Installer back-end is suitable for most packages⁸.

Note that the uninstaller executable is automatically added to the package on selection of the Excelsior Installer back-end. However, the executable can be renamed or removed, if necessary.

6.13.2 Self-contained directory

Select this option to have JetPackII simply copy the installation package files into a newly created directory on your system on the last step. Before copying the JET-compiled executables, JetPackII “binds” them to the JET Runtime included in the package, so they will run without any extra settings, such as PATH, and from any location, provided that the directory is deployed "as is" and in whole.

In this mode, the basic installation procedure for new packages and updates to client installations is reduced to simply copying the directory to the target system. It effectively means that you may burn the resulting directory to a CD or copy it to a removable drive, package it using simple utilities such as tar+gzip or supply it as input to any third-party installation tool.

⁸If you opt for Excelsior Installer, the client installation will be placed into one directory (or its subdirectories.) You therefore may not use multiple roots placement with Excelsior Installer.

6.13.3 Installation into multiple directories

Note: information in this section is not applicable to the Standard Edition of Excelsior JET.

Choose this back-end if you need the application files to be placed into multiple installation directories on the target system⁹. For instance, suppose you distribute several products sharing some common files. In such case, you may wish to divide each installation into two parts: common files and files specific to a particular product. In the Windows environment, common files are typically placed in a subdirectory of

system-drive: \Program Files\Common Files\

directory whereas the location of product-specific files may be selected by the user during installation. Therefore, the application files will be actually installed in two directories on the target system.

If you choose this back-end, the button



Create new root

will appear at the top of the package files panel. Use it to create a new root directory that corresponds to a separate installation directory on the target machine. Then, you can reorganize package files by placing them under different roots to compose a package that consists of several parts.

On the last step, JetPackII will create a separate directory for each part of the installation. Note that after the directories have been created, locations of the files inside them must not be changed and each directory must be copied to the target system as a whole.

Excelsior Installer does not support installation into multiple directories so you need to use a third-party setup generator to create the resulting installer. In addition to copying the application files, the installer will have to copy and invoke the `xbind` utility (see 6.21.2) from the Excelsior Installation Toolkit. It is necessary to hard-wire the pathnames of the actual installation directories into the compiled executables. On the last step, JetPackII will display an instruction describing the required post-install action in details.

6.14 Step 6: Specifying Installation Settings

This step is intended to describe the appearance and behavior of the resulting installer. Note that this page is accessible only if you opted for using Excelsior Installer on the previous step.

6.14.1 Vendor information

First of all, you should specify a *company name*, a *product name* and a *product version*. For that, use the respective entry fields from the **Vendor information** pane at the top.

⁹That is, the directories which have not a common root directory. For example, they may be placed on different disk partitions.

As you type the values, JetPackII automatically fills the default values for the installation directory which you may change later, if necessary.

In addition, the values entered in the fields **Product Name** and **Product Version** are used to compose the product identifier that will be displayed in the resulting installer.

6.14.2 Installation directory

The installation wizard of Excelsior Installer displays a command prompt where the user selects an installation directory for the application. In JetPackII, you can specify a default value for the installation directory or force installation into a particular location, not allowing the end user to change it.

Specifying default installation directory

If you need to set the default installation directory, select the desired option from the **Installation directory** radiobutton group. The following options are available:

- **Relative path:** the default directory pathname will start from the current directory followed by the value you type in the entry field to the right. For example, if you select this option and type `MyApp`, the default installation directory will be `./MyApp`.
- **Absolute path:** the installation directory is specified as an absolute path. Opting for it, you enter a full pathname starting from the root directory, for example `/opt/MyApp/MyFiles`. If the entered path is not valid¹⁰, JetPackII displays it in red until you correct the value.
- **Relative to home:** the default directory pathname will start from the current user's home directory followed by the value you type in the entry field to the right. For example, if you select this option and type `MyApp`, the default installation directory will be `/usr/home/MyApp`.

Fixing the installation directory

If you do not want the user to change the default directory when installing the application, uncheck the respective checkbox displayed below the entry field.

Pathnames with international characters

If the pathname of the installation directory contains non-ASCII characters, your program may function improperly due to defects in the internationalization facilities existing in certain versions of the Java platform. If you enter such a character in the installation directory field, JetPackII shall display a warning message when you attempt to leave this page.

¹⁰That is, it does not look like an absolute path.

Click **Check** to change the Trial Run directory pathname so that it matches the default installation directory pathname and repeat the trial run (see 6.12) procedure.

Click **Ignore** if you are sure the international characters in the directory name do not affect your software behavior or if you normally skip trial runs.

Click **Cancel** to return to the **Installation Settings** page and correct the pathname so that it contains only ASCII characters.

6.14.3 Installer appearance

On the **Installer appearance** tab you can set the language of the installer interface and the end-user license agreement that the installer will display.

Installation language

To set the language of the installation wizard, select the desired option from the **Language** combobox. Currently, English, French, German, Spanish, Polish, Russian, Japanese, Italian, and Brazilian Portuguese are supported. The default value "Auto-detect" means automatic selection of the language at install time depending on the system locale settings.

License agreement

To add the license agreement dialog to the resulting installer, type the full path to the desired text file in the **License agreement** field, or click **Browse** to select it using a conventional file dialog.

The license agreement must be a plain text file in either Unicode or ANSI encoding. You can set the encoding by selecting the respective radiobutton below the field.

6.14.4 Extra facilities

On the **Extra facilities** tab you can specify a runnable file that will be executed just after the end of the installation process. Another available facility is adding your own *callback shared libraries* that export functions which the installer will invoke before and after installation and uninstallation processes. Both techniques enables you to add extra functionality to the resulting installer/uninstaller.

After-install runnable

To specify the runnable file, click **Select** in the **After-install runnable** pane and select the desired file from the package. If you need to specify command line arguments to the runnable, type them in the **Arguments** field. Values of the arguments may contain references to root directories from the package in the form $\$(root-name)$. For example, if you specify an argument $\$(Root)/MyFiles/InstallStuff$, the installer will expand $\$(Root)$ to the absolute path of the installation directory on the target system.

The selected runnable file will be started after all installation procedures are completed.

Note: You should not rely on that the runnable starts in a particular working directory. If it needs to access the package files, you can specify the necessary arguments for the runnable as described above.

Callback shared libraries

If your product needs to accommodate to a specific target environment, you can use one of the following customization facilities:

- **Installation callback routines:** specify a custom callback shared library that will be invoked once just before copying files and once after the copying process is finished. This library can adjust package files or settings to a specific machine environment and/or perform extra installation actions.

To do that, click **Browse** next to the **Install callback shared library** and select the library using a conventional file dialog.

- **Uninstallation callback routines:** specify a custom callback shared libraries that will be invoked once just before uninstallation is started, so it can perform custom uninstallation actions.

To do that, click **Select** next to the **Uninstall callback shared library** and select the library from the package files.

For more information on the programming of the callback libraries see [6.22.3](#).

Deleting all files from installation folders

By default, Excelsior unInstaller removes only the files copied to the installation folder(s) during installation.

You can configure the uninstaller to remove all files from the installation folders if, for example, your application creates some temporary files in the folders.

To enable this option, check the box at the bottom of the **Extra facilities** tab.

6.15 Step 7: Specifying Installation Settings Of Windows Services

This step is needed only if the installation includes Windows services, so it is always skipped on Linux.

6.16 Step 8: Creating The Installation Package

This is the final step of creating the installation. This page contains the following fields.

Installation package — specify the location of the installation package here. If you use the Excelsior Installer back-end, it must be a single file, while for other back-ends the installation package is created as a directory. If you have already saved the project, this field contains the default value which is made of the project name and location.

Detached package — specify an absolute path and filename of the detached package. Note that this field appears only if you use Java Runtime Slim-Down model and your installation has detached Java SE components (see 6.11.2).

Back-end — a text label that shows the back-end type you chose on Step 7 (see 6.13). If you opted for Excelsior Installer, this field is a combobox that displays the back-end name and a desired strength of the compression algorithm to be used by the installer. The available options are trade-off between the size of the resulting installer and the time to build it.

Create! button — click it after specifying the package location. If your project is not saved yet, you are prompted to save it.

Then a progress window is displayed and, as the process is over, the creation of your installation package is completed. If you have chosen "Excelsior Installer" or "Self-contained directory", you now have a package ready for installation or CD burn. Otherwise (for "multiple roots" installations), detailed instructions are displayed. They explain how to integrate the generated image into a third-party installer.

You are recommended to test your newly created package by installing it to several machines with different operation systems, hardware configurations, etc. before starting distribution.

Save as updatable button — click to save the project file as updatable. It may be used later to create update packages, from minor version updates and hot-fixes to major upgrades.

To prepare an update package, create a new project and click **Update package** on the Welcome Screen (see 6.6.3).

After the successful creation of an update package, you can save the modified project as updatable, thus building a chain of updates.

Note: If you plan to issue minor updates to your application, it makes sense to write down the version of Excelsior JET, the current profile (see 2.3) and the exact configuration of JET updates (see 2.4) you used to compile and deploy your application, so as to avoid JET Runtime updates (see 6.20).

6.17 Packaging native method libraries

In most cases, Java native methods are placed in *native methods libraries* (shared libraries) which the application loads on demand at run time. If your application's classes contain native methods or the application uses a third-party component that includes native libraries, you need to add them to the installation.

Problems may arise when the installed application loads such a library unless you spec-

ify the appropriate search path¹¹. Fortunately, the Java platform provides an OS-neutral way of lookup of the native libraries through setting the `java.library.path` system property.

You can set the search path using JetPackII as follows. Suppose your installation contains a native library placed in the `$(Root)/myLibs` directory. Specify, the property

```
java.library.path=$(Root)/myLibs
```

on Step 2 (see 6.10.2.) As a result, the appropriate search path for the application will be automatically set on the target system.

6.18 Installations protected by license managers

Excelsior JET Optimizer protects your proprietary code from reverse engineering thus improving IP protection. However, it does not provide *copy protection*, that is, preventing unauthorized copying or use of the program. To resolve the issue, you can employ complementary protection mechanisms such as software license managers or hardware keys.

License managers typically encrypt the executable file and then add a wrapper around it. However, that does not allow JetPackII to work with such executables because it needs to read from the executable some information necessary for creating the installation. Therefore, you should use such copy protection tools *after* the installation is created.

That is not a problem if your installation type is self-contained directory (see 6.13.2): you may encrypt the executable(s) after JetPackII has creates the directory.

However, if you use Excelsior Installer, JetPackII directly creates a self-extracting setup so there is no intermediate step when the encrypting tools could work. The problem can be easily resolved using the `xpack` utility from Excelsior Installation Toolkit (see 6.21.)

The procedure of creating an Excelsior Installer setup that includes encrypted executables is as follows.

1. Create the Excelsior Installer setup in JetPackII and install the package on a test machine to make sure that the application works as expected if encryption is not used. On this step you have a project file which you saved on the last page of JetPackII. Let us call it `MyPackage.jpjn`
2. Create an empty directory for temporary use.
3. Execute the command

```
xpack project -make-image temp-directory
```

supplying the JetPackII project name and the temporary directory as arguments. In this example, the exact command line would be

```
xpack MyPackage.jpjn -make-image /home/user/encrypt
```

In the specified directory, the `xpack` utility will create the installation which is *not yet wrapped* into the Excelsior Installer setup.

¹¹A Java runtime throws `UnsatisfiedLinkError` exception if it fails to load a native library.

4. Encrypt the executable(s) in the temporary directory
5. Run the encrypted executable(s) to make sure that the application works as expected
6. Execute the command

```
xpack project -from-image temp-directory -target target-file
```

supplying the same arguments for *project* and *temp-directory*. As a result, the `xpack` utility will create the Excelsior Installer setup that includes the encrypted executables. The resulting installer package name is *target-file*.

Now you can safely deploy the installation package to target systems.

Note: Copy protection mechanisms that involve encrypting executables cannot be used for multiple root installations (see 6.13.3). This limitation will be removed in future versions of Excelsior Installation Toolkit.


6.19 Relocation of JetPackII projects

In some cases, you may need to create the resulting package in an environment different from where the project was created, for example, on another machine¹² or on the same computer with the placement of package files changed. In such scenario, it would be reasonable to reuse the existing project rather than to create a new one from scratch. JetPackII can help you update the existing project so that all settings and properties you specified in it are retained.

Suppose that the package files were copied to (or built in) a new environment. Ideal case for packaging would be just copying the project file (`.jpn`) along with other files to the new location and rebuilding the package in the batch mode.

Note that a JetPackII project uses paths *relative* to its own location to reference the files included in the package. Therefore, if the overall file structure remains unchanged and the files are still available through the same relative paths, you have to do nothing to enable building the project in the new environment.

If some of the package files are not found this way, JetPackII automatically detects that the project has been relocated and lets you either supply the files, or remove them from the project. The following procedure helps you relocate an existing JetPackII project.

1. Start JetPackII in the new environment and open the project file copied. JetPackII detects that the project was relocated and issues a warning. Click **OK** to display the **Package files** panel and check the package for completeness.
2. If any package files are marked as absent (displayed with the icon ) , fix them as follows:

Check the **Show full path** box on the bottom pane to see where JetPackII expects to find the missing files. Then, either copy or move the files to the appropriate location or remove them from the project and add such files again from their present

¹² For that, you have to install JET on that machine and create an identical profile.

locations using the **Filechooser** pane on the right. Note that in the latter case, file-specific settings, for example user-defined system properties (see 6.10) referencing the removed files, will be lost.

3. Perform the Trial Run for safety.
4. Go to the **Finish** page and click the **Create!** button. As the creation of the package finishes, the displayed information will contain instructions on how to build the package in the batch mode.

6.20 Updating JET Runtime

As you know, an installation package created with JetPackII includes JET Runtime files (see 6.7.1) When you include a new version of a certain executable into an update package (see 6.8.2), it may turn out that it was compiled for a different version of the JET Runtime. JetPackII is capable of updating the JET Runtime files automatically in such situations.

6.20.1 When a JET Runtime update is necessary

A JET Runtime update is required if you have done one of the following since the creation of the original installation package:

- You have upgraded to a newer version of Excelsior JET;
- You have switched to a different Java microversion, e.g. from Java SE 6 Update 12 to Java SE 6 Update 20;
- You have applied an update to Excelsior JET itself (see 2.4) that fixed some problems in the JET Runtime; or
- You have changed some of the endorsed standards (see 2.3)

Note: Even if you only intend to update files other than JET-compiled executables, JetPackII forces the JET Runtime into the update package under the following circumstances:

- The version of Excelsior JET you are using is newer than the version used to create the original package; or
- The executables included in the original package were compiled using a profile (see 2.3) that is not present on your system

Tip: To avoid inclusion of the JET Runtime into the minor updates of your applications, make sure to enable reproduction of the respective build environments (see 2.6).

6.20.2 JET Runtime Update dialog

When you initiate the creation of an update package, JetPackII automatically determines whether the JET Runtime included in the original package also needs an update, as described in 6.20.1.

If there is a need to force a JET Runtime update, the respective dialog is displayed immediately after you click **Update Package** on the Welcome Screen. Otherwise, the Difference Tracker is displayed as usual and you may select the files that you want to include in the update package as described in 6.9.2. Then, if you have included in the update any executables created by JET Optimizer, JetPackII checks them for compatibility with the JET Runtime included in the original package. If JetPackII has determined that a JET Runtime update is necessary, it displays the **JET Runtime Update** dialog.

The dialog prompts you to choose a new profile for the update package. The profile used when building the original package is marked as <ORIGINAL>, and the recommended profile is marked as <RECOMMENDED>. Select the desired profile and click **OK**. The respective version of the JET Runtime will be included in the update package.

Notes:

1. The current version of JetPackII includes all JET Runtime files into an update package, even if some of them were not changed. This will be improved in future versions.
2. JetPackII further ensures that there is only one version of the JET Runtime in any actual installation. So if your update package includes a new version of the JET Runtime for any of the reasons listed in 6.20.1, you must recompile *all* executables that were part of the original package, so as to make them work with that new version of the JET Runtime, and include them in the update package.

6.21 Deployment Utilities

6.21.1 xpack

The **xpack** utility is used to re-build an installation package in the batch mode (from the command line or build scripts). After you have created an installation package using JetPackII and saved its project file, you can build the package with the command:

```
xpack JetPackII-project-file [-target target-file] [-version version]
```

The `-target` option specifies the name of the target file (for Excelsior Installer setups) or the target directory (for self-contained directory or multiple roots package). If the option is omitted, the name shown on the last page of JetPackII is used.

The `-version` option may be used for changing the product version displayed by the installation wizard of Excelsior Installer. It takes no effect for other back-ends.

The `-quiet` option disables printing the log to the standard output.

xpack is useful for minor updates, e.g. when only particular files are modified, provided that the overall structure of the project and installation settings were not changed.

The `xpack` utility can also help if you want to use a third-party license manager to protect your installation packages. The following command line prepares intermediate installation image for post-processing by a license manager or by other binary encryption utilities:

```
xpack JetPackII-project-file -make-image image-dir
```

After the post-processing is done, the following command line creates an installation from the post-processed image:

```
xpack JetPackII-project-file -from-image image-dir [-target target-file]
```

Refer to 6.18 for details.

6.21.2 xbind

The **xbind** utility is used to perform post-installation processing of JET-compiled executables. You would normally use it in multiple root installations created with third-party installers.

xbind takes two files as input: a *script file* and a *redirection file*.

JetPackII prepares an xbind script file when creating the installation image. You do not need to edit that file.

An xbind redirection file should be prepared by the installer after the destination directory for each package root is defined. A redirection file is a plain text file, with each line specifying a root directory location on the target machine, in the following format:

```
$ (root-name)=root-location
```

For example,

```
$ (Root) = /usr/home/MyApp
```

Post-installation processing can then be performed via the command:

```
xbind xbind-script xbind-redirection [-unicode-redirection]
```

xbind recognizes redirection files in the Unicode character set. The default encoding is ANSI. Use the `-unicode-redirection` option to specify that the redirection file is a Unicode text file.

6.22 Excelsior Installer

Excelsior JET includes an installation tool which helps you to compose a setup that contains JET-compiled applications along with other files.

6.22.1 Installer

The Installer automatically *binds* installed JET-compiled executables to JET Runtime components. This technique guarantees that installed applications will use exactly the same versions of files that were used during testing on the developer's machine. As a

result, the behavior of deployed applications will be identical to that on the developer's machine during a Trial Run. Besides, the location of standalone resources will be automatically added to the classpath making all executables clickable (no need in launching scripts).

Binding also helps avoid so called "DLL Hell" because shared libraries are no longer looked for along `LD_LIBRARY_PATH` so all import between components is unambiguously resolved independently on the target system environment variables.

There are two possible ways of installation:

- *Interactive installation* – following on-line prompts and specifying setup parameters.

The interactive installation consists of several steps:

1. Installer displays the License Agreement and prompts the user to accept it;
2. Installer prompts user to enter the destination folder or to accept the default one;
3. All files are copied to the destination location;

- *Automatic installation* — in this mode a package can be installed without user interaction. This is useful for a quick product installation to multiple machines, for testing the installer package in different environments, etc.

To launch an installation in such unattended mode, specify the `/batch` option to the installer executable. Other command-line options specify installation settings, such as destination folder, etc. The installation process then runs on its own, no user input is required.

The following command-line options are recognized:

`/dest "full-path"` - defines the destination folder pathname. If it is not specified, the application will be installed into the default folder. The default folder value could be specified with JetPackII (see 6.14.2.)

If the latter was not specified, the setup will ask user for proper path.

`/force` - forces the setup to override an already installed package instance. If it is not specified and if another instance is detected, the setup will not proceed.

`/no-afttrun` - suppresses invocation of after-install runnables

`/http-port` - sets the HTTP port for Tomcat-based applications (see 8.3.2 for details).

`/?` - shows a brief help page.

Note: Both `-` and `/` are recognized as option starting characters, so `/no-afttrun` is equal to `-no-afttrun`.

If a project contains an installation callback shared library or a post-installation runnable, or both, they are invoked in the following order:

- the installation callback shared library is loaded;

- the PreInstall function from the installation callback shared library is invoked:

```
int PreInstall (char* installation_folder_name);
```
- the installation callback shared library is unloaded;
- application files are copied to the destination location;
- the installation callback shared library is loaded;
- the PostInstall function from the installation callback shared library is invoked:

```
int PostInstall (char* installation_folder_name);
```
- the installation callback shared library is unloaded;
- the installation exits, the post-installation runnable is invoked.

For the functions above, the return value of integer type means: zero — no error, installation may proceed, any non-zero value — there is an error, installation must be aborted.

6.22.2 Uninstaller

An Excelsior Installer package may include an uninstaller executable, which removes installed package files on user request.

The uninstallation acts as follows:

- the confirmation question is displayed;
- if the project contains an uninstallation callback shared library, the

```
int PreUninstall (void);
```

function is invoked; its return value means: '0' - no error, uninstallation may proceed, 'non-zero' - an error occurred, uninstallation must be aborted.
- all files are deleted (including the uninstaller itself);
- uninstallation program exits;

To perform uninstallation in the *unattended mode*, the user should specify the `/batch` option on the command line of the uninstaller executable.

Uninstallation is cumulative, that is, it removes the initially installed package along with all updates that were possibly installed later.

6.22.3 Callback shared libraries

The *install callback* shared library should export the two functions:

```
int PreInstall (char* installation_folder_name)
int PostInstall (char* installation_folder_name)
```

where the `installation_folder_name` parameter is the absolute pathname of the installation directory. The *uninstall callback* shared library should export the function

```
int PreUninstall (void)
```

Return value of the functions is a return code. If they return a non-zero value, the (un)installation process will be terminated and a dialog displaying the respective message and the return code, will appear. The installer/uninstaller invokes the functions in the order described in sections [6.22.1](#) and [6.22.2](#).

Note: when building the shared libraries, enable the static linking mode so that they do not depend on any other shared libraries except those implementing the system API.

Chapter 7

Eclipse RCP

Note: information in this Chapter is applicable only to the Enterprise Edition of Excelsior JET.

Since version 6.5, Excelsior JET provides a protected environment for running Eclipse RCP applications. You may compile your RCP application to a native code executable and distribute it without the original class/jar files. As a result, all those Java decompilers become useless to the malicious programmers and attackers simply because Java classes are not included in the end user installations.

At the same time, compilation to native code substantially improves the security and IP protection of Eclipse RCP applications, as the sensitive code and data get protected from reverse engineering (see Chapter 5), tampering with the OSGi bundles becomes impossible, and the exposure of security vulnerabilities is reduced.

7.1 Overview

At a glance, the usage of Excelsior JET for an RCP application looks as follows:

1. You specify the RCP application directory¹ to the JET Optimizer
2. The Optimizer compiles OSGi bundles (plugins) into a native executable and places it into that directory. The executable looks like the standard executable launcher of RCP applications but it contains the native compiled code.
3. To prepare the application for deployment, you use the JetPackII tool (see Chapter 6) specifying the RCP application directory and an *output* directory. JetPackII replicates the structure of the original directory adding the necessary JET Runtime files and removing the class/jar files which are no longer required.

¹This is a directory to which the application was exported either by the Product Export Wizard of the Eclipse IDE or by an automated build process.

7.1.1 The main functionality

As a result, you create the RCP application directory that can be deployed to end user systems. It has the same structure as the original one and contains the launcher executable. The two major differences are:

1. the plugin classes/jars are not included – they are transformed into native code highly resistant to reverse engineering
2. the compiled RCP application works without the JRE – no need to install (a particular version of) Java on end user systems

Note: If necessary, you or your end users may install other (non-compiled) plugins in the compiled RCP application as usual.

7.1.2 Extra features

In addition, you may author the application installer using the Installation Toolkit (see 6.13.1) bundled with Excelsior JET. Yet another Excelsior JET's capability available to the developers of Eclipse RCP applications is download size reduction for the installation packages. Using the Java Runtime Slim-Down deployment model (see Chapter 9), you may leave the unused parts of the Java SE platform, such as Swing, out of the application installer ².

7.1.3 Automated builds

To enable automated builds, you need to create two project files for each RCP application. One project is for the JET Optimizer to build the native executable and the other – for JetPackII to *package* the application, that is, to prepare it for deployment. The result of packaging is either a self-contained directory containing the application ³ or a complete installation package powered by Excelsior Installer.

Once the projects have been created, you may use the command line interface to the JET Optimizer (see 16.2) and JetPackII (see 6.21.1) to enable native compilation and packaging in an automated build process.

You use the JET Control Panel (see Chapter 4), to quickly set up the project for the JET Optimizer. The created project file has the extension `.prj` and can be used in automated builds with the command

```
jc =p Compilation-project.prj
```

The JetPackII graphical tool (see Chapter 6) helps you create the project for packaging. The project file has the extension `.jpn` and can be used in automated builds with the command

²The reduction can be achieved for small and medium-size applications that include a couple of thousands of classes. For large RCP applications, such as the Eclipse IDE, the removing of a few unused Java SE components will not reduce the total download size substantially.

³It is similar to the directory created when the RCP application is exported but it is truly self-contained because the application no longer needs the JRE to run.

```
xpack Packaging-project.jpjn
```

Note: If you modify the RCP application, for example, add/remove some plugins, you do *not* need to create new projects nor edit the existing ones. When compiling the application, all changes will be automatically recognized and processed with the default settings. You open the projects with the JET graphical tools only if you need to change their settings.

The rest of this chapter contains a detailed description of how to use Excelsior JET for Eclipse RCP applications.

7.2 Compilation

To create a new project for compiling an RCP application, start the JET Control Panel and click **Eclipse RCP Application** on the Welcome Screen. The **Start** page will be displayed.

7.2.1 Step 1. Describing RCP application

On this page, browse for the directory to which your RCP application was exported either by the Product Export Wizard of the Eclipse IDE or by the existing automated build process.

Then, select the launcher executable of your RCP application⁴. Typically, the launcher executable comes with a configuration file that has the same name and the extension `.ini`. If the file exists, the Control Panel parses it and then creates the project for the RCP application.

At this point you may go directly to the page **Finish** (see 7.2.7) and compile the RCP application. It is recommended, however, to proceed to the next step and run the application in a special test mode using the JET Runtime.

7.2.2 Step 2. Test Run for RCP application

On the **Test run** page, you may run the application before compiling it to native executable.

Note: Test Run for large RCP applications may take a while, in particular, the application may start much longer than usual. That is expected behavior because the JET Runtime not only executes the application but also profiles it, which is useful for subsequent optimization. For more details, see 4.6.

After the Test Run is over, you may go to the next page to check/modify the default settings for the application's OSGi bundles (plugins).

⁴The Control Panel automatically detects it if there is the only launcher executable in the specified directory.

7.2.3 Step 3. Settings for OSGi bundles

On the **Classpath** page, you may specify which bundles need to be protected by compiling to native code.

Grid

The central area of the page is occupied with a grid control. In the left column, the grid displays the names of the OSGi bundles marked with gray background. The other columns contain settings for the application's bundles.

Bundles consisting of a single jar are displayed as single rows of the grid.

If a bundle contains several jars, it is displayed as follows:

- the contained jars are displayed on separate rows below the bundle name.
- a special row marked with the "." char ("root directory") is added to denote the (separate) classes and resource files placed into the bundle along with the jar files.

Compilation settings for the bundles

If you need to protect the code from particular bundles, select the "all classes" option in the combo box from the **Protect** column. As a result, the contained classes will be native compiled and, therefore, will be omitted from the end user installation of your RCP application.

If, for some reasons, you want to keep certain OSGi bundles in bytecode form, adjust the settings in the respective grid rows as follows: select "auto-detect" in the **Optimize** column and "not required" in the **Protect** column. Such non-compiled bundles will be handled by the JIT compiler (see Chapter 11) that comes with the JET Runtime.

Other controls available on this page are described in section 4.7.

7.2.4 Step 4. Optimizer and Runtime options

The **Options** page does not contain any controls specific to Eclipse RCP applications. You may find a detailed description of this page in section 4.8.

7.2.5 Step 5. Settings for the RCP executable

In the **Executable settings** pane of the **Target** page, you select the name of the native executable that will be created in the RCP application directory after compilation. By default, the Control Panel offers name *Launcher-name-native* but you may change it as you wish.

Also in this pane, you may enable protection of the *application data* (see 5.2) by checking the **Encrypt strings and resources** box.

Other controls available on this page are described in section 4.9.

7.2.6 Step 6. Application appearance

The page **Appearance** simply displays the splash screen image you selected when creating the RCP application. You cannot change it on this page. If you need to do that, you may replace the splash image in the RCP application before using Excelsior JET and the Control Panel will then automatically recognize the change.

7.2.7 Step 7. Building the executable

On the page **Finish**, click **Build** to compile the application. You will be prompted for saving the project. Using the filechooser select the name of the project file and the directory to which it will be placed. Note that the Optimizer will create temporary files (see 16.5) in that directory during compilation.

If you compile a large RCP application, it is not recommended to build it from within the Control Panel because less memory will be available to the JET Optimizer. Instead, click **Create build script**, select the project file name and directory and exit the Control Panel. Then launch the `build_Project-name` script created in the project directory.

As compilation is over, the native executable will appear in the RCP application directory you specified when creating the project.

Note: By default, the JET Control Panel automatically launches the resulting executable after compilation. It is done to collect profile data useful for reducing the application startup time. If you do not wish to enable the startup optimization, you may turn the profiling run off as described in section 4.11.1.

7.3 Packaging for deployment

To create a new project for packaging an RCP application, start JetPackII (see Chapter 6) and click **New Eclipse RCP Installation** on the Welcome Screen. The **Files** page will be displayed.

7.3.1 Files to package

On this page, browse for the directory that contains the compiled RCP application. JetPackII will automatically add the necessary files to the package.

The structure and contents of the package are displayed in the **Files to package** pane. Notice that the plugin jars compiled to native code are *not included* in the installation package.

Later, if you need to add more files to the package or remove some previously added files, you may copy/remove them to/from the RCP application directory and JetPackII will automatically update the list of files to package. If you do that while JetPackII is running, select **File/Refresh** from the main menu to display the changes immediately.

7.3.2 Configuring the project

Other steps of creating the project are not specific to Eclipse RCP and described in Chapter 6.

7.3.3 Building the package

On the page **Finish**, select the full pathname to the installation package and click **Create!**. You will be prompted for saving the JetPackII project. Using the filechooser select the name of the project file and the directory to which it will be placed. Later you may use the project file to automatically build the installation as described in section 6.21.1.

Depending on the install type you chose on the page **Backend** (see 6.13), JetPackII creates either application installer or self-contained directory in the specified location.

7.4 Known limitations

Excelsior JET imposes some limitations on the entire Eclipse RCP functionality and has some known issues:

1. It is assumed that all bundles are placed in

RCP-Directory/plugins subdirectory

If your application is configured to use multiple bundle repositories, you may re-configure it to use only the `/plugins` directory.

2. The Eclipse RCP auto-update facility is not fully supported. Specifically, it is not possible to update pre-compiled bundles. This functionality is currently supported only for the bundles left in bytecode form.
3. Compilation of very large Eclipse RCP applications (2x the size of the Eclipse IDE or larger) takes long or runs out of memory. See the 7.6 for possible resolution.

These limitations will be removed in future versions of Excelsior JET.

7.5 FAQ

What are the supported versions of the Eclipse Runtime?

Excelsior JET 7.6 supports the Equinox OSGi runtime versions 3.1 through 3.6 out-of-the-box. In particular, it has been tested on the Galileo Release of the Eclipse Platform as well as on the previous Eclipse releases. More recent version of the Eclipse RCP will be added through Maintenance Packs available at

<http://www.excelsior-usa.com/jetdlupd.html>

Versions prior to 3.1 are not supported.

Does Excelsior JET support the dynamic loading of plug-ins that were not pre-compiled?

Yes, of course. The JET Runtime includes a JIT compiler to dynamically load Java class files that were not pre-compiled. For more details, refer to Mixed Compilation Model (see Chapter 11).

Can I compile only certain OSGi bundles to native code and leave the rest intact?

Yes, it is possible. Configure the JET project as described in section 7.2.3.

7.6 Troubleshooting

The Control Panel rejects the RCP application directory I specified

Reason: The directory is outdated or does not contain an RCP application launcher

Resolution: Export the latest version of the RCP application to this directory

Compilation takes way too long or the compiler terminates with an insufficient memory error

Reason: Under circumstances, the JET Optimizer does not scale well for very large RCP applications.

Resolution1: If you enabled the Global Optimizer, disable it as described in section 4.7.3

Resolution2: Open the JET project file (`.prj`) with a text editor and modify it as follows:

1. add the option `-compilewithfixedheap+` to the beginning of the project
2. increase the value of the **COMPILERHEAP** equation. For example, specify

```
-compilerheap=1300m
```

Note: your computer should have 2+ GB of RAM to build large applications

Resolution3: Disable native compilation for some of the application's bundles which you do not need to protect as described in section 7.2.3. Note, however, that it may negatively affect the application performance.

Chapter 8

Tomcat Web applications

Note: information in this Chapter is applicable only to the Enterprise Edition of Excelsior JET.

Since version 7.0, Excelsior JET enables you to protect Apache Tomcat Web applications from decompilation.

You may compile the Tomcat Server along with your Web application to a native code executable and distribute it without the original class/WAR files. As a result, all those Java decompilers become useless to the malicious programmers and attackers simply because Java classes are not included in the end user installations.

At the same time, compilation to native code substantially improves the security and IP protection of Tomcat Web applications, as the sensitive code gets protected from reverse engineering (see Chapter 5) and the exposure of security vulnerabilities is reduced.

8.1 Overview

At a glance, the usage of Excelsior JET for Tomcat Web applications looks as follows:

1. You specify *Tomcat home directory* to the JET Optimizer. The directory should contain the Tomcat Server with Web applications deployed into it.
2. The Optimizer compiles Tomcat along with all deployed Web applications into a native executable and places it into `bin-native/` subdirectory of the Tomcat directory. To start the compiled Tomcat Server, you may simply run the created executable.
3. To prepare the Web application(s) for distribution, you use the JetPackII tool (see Chapter 6), specifying the Tomcat home directory and an *output* directory. JetPackII replicates the structure of the original directory adding the necessary JET Runtime files and omitting the class/WAR files which are no longer required.

8.1.1 The main functionality

As a result, you create the Tomcat directory with the deployed Web applications that can be distributed to end users. It has the same structure as the original one, contains the compiled executable and, optionally, standard Tomcat scripts. The two major differences are:

1. the class/WAR files are not included as they were transformed into native code
2. the compiled Tomcat and Web application work without the JDK – no need to install (a particular version of) Java on target systems

Note: If necessary, you may deploy other (non-compiled) Web applications on the compiled Tomcat Server as usual.

8.1.2 Extra features

In addition, you may author the application installer using the Installation Toolkit (see 6.13.1) that comes with Excelsior JET.

Yet another Excelsior JET's capability available to the developers of Web applications is reduction of the install package size. Using the Java Runtime Slim-Down deployment model (see Chapter 9), you may bundle a light version of Java runtime with Tomcat leaving unused parts of the Java SE platform, such as Swing, out of the application installer¹.

8.1.3 Automated builds

To enable automated builds, you need to create two project files. One project is for the JET Optimizer to build the native executable and the other – for JetPackII to *package* the application, that is, to prepare it for distribution. The result of packaging is either a self-contained directory with the application² or a complete installation package powered by Excelsior Installer.

Once the projects have been created, you may use the command line interface to the JET Optimizer (see 16.2) and JetPackII (see 6.21.1) to enable native compilation and packaging in an automated build process.

You use the JET Control Panel (see Chapter 4), to quickly set up the project for the JET Optimizer. The created project file has the extension `.prj` and can be used in automated builds with the command

```
jc =p Compilation-project.prj
```

¹The reduction can be achieved for small and medium-size applications that include a couple of thousands of classes. For large Web applications, the removal of unused Java SE API classes will not make much difference.

²It is similar to the Tomcat directory but is truly self-contained because the compiled Tomcat no longer needs the JDK to run.

The JetPackII graphical tool (see Chapter 6) helps you create the project for packaging. The project file has the extension `.jpn` and can be used in automated builds with the command

```
xpack Packaging-project.jpn
```

Note: If you remove/modify the Web applications located in the Tomcat directory or deploy a new application, you do *not* need to create new projects nor edit the existing ones. When compiling Tomcat, all changes will be automatically recognized and processed with the default settings. You open the projects with the JET graphical tools only if you need to change their settings.

The rest of this chapter contains a detailed description of how to use Excelsior JET for Tomcat Web applications.

8.2 Compilation

To create a new project for compiling Tomcat Web application(s), start the JET Control Panel and click **Tomcat Web application** on the Welcome Screen. The **Start** page will be displayed.

8.2.1 Step 1. Describing Web applications

On this page, select the Tomcat home directory that contains the Tomcat Server and Web application(s) you want to compile. The Control Panel scans the specified directory and displays the detected Tomcat version and the names of found Web applications in the **Tomcat Server and Web applications** pane.

At this point you may go directly to the page **Finish** (see 8.2.7) and compile Tomcat along with the Web applications. It is recommended, however, to proceed to the next step and run the application in a special test mode using the JET Runtime as described in section 8.2.2.

Tomcat customization

When developing Web applications running on Tomcat, you may have changed the default Tomcat bootstrap sequence, used your own main class for the Tomcat Server and so on. If that is the case, you need to describe the modifications when creating the project.

Click **Customize Tomcat** to open the Tomcat customization dialog where you may add more jars files/directories to the classpath, change the default Tomcat entry point – the main class, and specify extra VM options. You may find an example of the Tomcat main class in the subdirectory `samples/TomcatCustomization` of your Excelsior JET installation directory.

Troubleshooting

As a result of the Tomcat directory scanning, the Control Panel may detect severe errors that would block native compilation of Tomcat unless they are corrected. For example, if the directory contains an ancient version of Tomcat not supported by Excelsior JET or the directory structure is corrupted, the creation of a native executable is not possible.

If such errors are displayed in the **Tomcat Server and Web applications** pane, you need to fix them to proceed to the next steps as described in section 8.6. Note that you may do that without exiting the Control Panel - just click the **Refresh** button after the errors are fixed and the scan procedure will be repeated.

8.2.2 Step 2. Test Run for Web applications

On the page **Test run**, you may run the Tomcat Server and your Web application(s) on the JET Runtime *before* native compiling them.

Note: During Test Run, the initial response time of a large Web application may be lower than usual. That is expected behavior because the JET Runtime not only executes the applications but also profiles them, which is useful for subsequent optimization. For more details, see 4.6.

After the Test Run is over, you may go to the next page to check/modify the default settings for the Web applications.

8.2.3 Step 3. Settings for Web applications

On the **Classpath** page, you may specify which classes, jars, or WAR files need to be protected by compiling to native code.

Grid

The central area of the page is occupied with a grid control where the Control Panel shows the Tomcat directory contents organized as a set of *sections*.

Each Web application deployed to the Tomcat Server is displayed as a separate section. The other sections are:

- Tomcat bootstrap classpath
- Tomcat implementation jars and library jars - those located in *Tomcat-home/lib/*

In the left column, the classpath grid displays the names of sections marked with gray background. The other columns contain settings for the jars, WAR files, or directories that belong to the section.

Sections consisting of a single file or directory are displayed as single rows of the grid. If a section contains several jars, e.g. an application WAR file or directory, it is displayed as follows:

- the contained jars are displayed on separate rows below the section name.
- a special row marked with the "." char ("root directory") is added to denote the (separate) classes and resource files found in the WAR file or directory along with the jar files.

Compilation settings

If you need to protect a particular Web application, select the "all classes" option in the combo box from the **Protect** column. As a result, the contained classes will be native compiled and, therefore, omitted from the end user installation of your Tomcat Web application.

If, for some reasons, you want to keep certain applications in bytecode form, adjust the settings in the respective grid rows as follows: select "auto-detect" in the **Optimize** column and "not required" in the **Protect** column. Such non-compiled jars will be handled by the JIT compiler (see Chapter 11) that comes with the JET Runtime.

Other controls available on this page are described in section 4.7.

8.2.4 Step 4. Optimizer and Runtime options

The **Options** page does not contain any controls specific to Tomcat Web applications. You may find a detailed description of this page in section 4.8.

8.2.5 Step 5. Settings for the Tomcat executable

In the **Executable settings** pane of the **Target** page, you select the name of the native executable that will be created after compilation in *Tomcat-home/bin-native/* directory. The Control Panel offers some default name for the executable but you may change it, if necessary. Also in this pane, you may enable protection of the *application data* (see 5.2) by checking the **Encrypt strings and resources** box.

If you do not want your end users to inspect or modify the Tomcat configuration files – those located in *Tomcat-home/conf/* – you may tell the JET Optimizer to place them into the executable so the files will not appear in *conf/* subdirectory of end user installations of your Web application. For that, check the **Hide Tomcat configuration files** box in the **Misc** pane.

If you use standard Tomcat scripts from *Tomcat-home/bin/*, such as `startup`, `shutdown`, etc., and wish to continue using them with the compiled Tomcat, check the **Generate standard Tomcat scripts** box in the **Misc** pane. As a result, the scripts working with the compiled Tomcat will be created in *Tomcat-home/bin-native/* along with the executable.

Other controls available on this page are described in section 4.9.

8.2.6 Step 6. Application appearance

This page is not available under Linux.

8.2.7 Step 7. Building the executable

On the page **Finish**, click **Build** to compile the application. You will be prompted for saving the project. Using the filechooser select the name of the project file and the directory to which it will be placed. Note that the Optimizer will create temporary files (see 16.5) in that directory during compilation.

If you compile a large Web application, it is not recommended to build it from within the Control Panel because less memory will be available to the JET Optimizer. Instead, click **Create build script**, select the project file name and directory and exit the Control Panel. Then launch `build_Project-name` script created in the project directory.

As compilation is over, the native executable will appear in the `bin-native/` subdirectory of the Tomcat directory you specified when creating the project.

Note: By default, the JET Control Panel automatically launches the resulting executable after compilation. It is done to collect profile data useful for reducing the startup time of the Tomcat server. If you do not wish to enable the startup optimization, you may turn the profiling run off as described in section 4.11.1.

8.3 Packaging for distribution

To create a new project for packaging the compiled Web applications, start JetPackII (see Chapter 6) and click **New Tomcat installation** on the Welcome Screen. The **Files** page will be displayed.

8.3.1 Files to package

On this page, browse for the Tomcat directory that you specified when compiling the Tomcat Server and Web applications. It should contain `bin-native/` subdirectory with the compiled executable. JetPackII will then automatically add the necessary files to the package.

The structure and contents of the package are displayed in the **Files to package** pane. Notice that the Web application jars compiled to native code are *not included* in the install package.

Later, if you need to add more files to the package or remove some previously added files, you may copy/remove them to/from the Tomcat directory and JetPackII will automatically update the list of files to package. If you do that while JetPackII is running, select **File/Refresh** from the main menu to display the changes immediately.

8.3.2 Configuring the project

Other steps of creating the project are not specific to packaging Tomcat Web applications and described in Chapter 6. There are some minor differences which are given below.

Trial Run

The Trial Run procedure for Tomcat applications is mostly the same as for applications of other types (see 6.12).

When the Trial Run is over, press Ctrl+C in the console or launch the `bin/shutdown` script from the Trial Run directory to stop the compiled Tomcat Server. Note that JetPackII cannot automatically detect whether a Trial Run session for Tomcat Web applications has been completed successfully. To set the Trial Run status, JetPackII asks you to confirm that everything worked as expected and, just for sure, inspect the Tomcat log files created during Trial Run.

HTTP port

If you use Excelsior Installer, you may allow the user to change the Tomcat's default HTTP port when installing your application. For that, check the **Allow the user to change the default Tomcat port** box on the **Misc** page of JetPackII. This will enable the user to specify a (valid) HTTP port number for the Tomcat Server installation (see 6.22.1).

Note: Before enabling this capability, you may need to correct Tomcat configuration files. Specifically, if the HTTP port number is explicitly set via the `port` attribute of the `<Connector>` tag in the `conf/server.xml` configuration file, you should replace the default value 8080 with a newly introduced VM option of your choice, e.g. `my.app.http.port`. JetPackII displays a detailed instruction on the necessary changes when you try to check the box.

If the user skips the step of changing the HTTP port number when installing your application, the default port will be 8080. You may define another default value by adding the line

```
my.app.http.port=port-number
```

to the `conf/catalina.properties` configuration file.

8.3.3 Building the package

On the page **Finish**, select the full pathname to the installation package and click **Create!**. You will be prompted for saving the JetPackII project. Using the filechooser select the name of the project file and the directory to which it will be placed. Later you may use the project file to automatically build the installation as described in section 6.21.1.

Depending on the install type you chose on the page **Backend** (see 6.13), JetPackII creates either application installer or self-contained directory in the specified location.

8.4 Known limitations

Excelsior JET 7.6 imposes the following limitations on the Apache Tomcat functionality:

1. It is assumed that all Web applications must be deployed into a subdirectory of *Tomcat-home*. It may be `webapps/` or another directory.

If your Web applications are located outside of *Tomcat-home*, you should reconfigure their deployment location to use Excelsior JET.

2. As Web applications are compiled together with Tomcat into one executable, hot re-deployment of a running pre-compiled application is not possible.

You would need to stop Tomcat, replace the executable, and start it again.

These limitations may be removed in future versions of Excelsior JET.

8.5 FAQ

What are the supported versions of Apache Tomcat?

Excelsior JET 7.6 supports Apache Tomcat 5.0.x³, 5.5.x, and 6.0.x

Versions prior to 5.0 will not be supported.

Can I deploy additional applications on the native compiled Tomcat?

Yes, you can do that using the standard Tomcat deployment procedure. Web applications deployed this way will be handled by the JIT compiler that comes with the JET Runtime. For more details, refer to Mixed Compilation Model (see Chapter 11).

Can I compile only certain Web applications to native code and leave the rest intact?

Yes, it is possible. Configure the JET project as described in section 8.2.3.

8.6 Troubleshooting

The Control Panel rejects the Tomcat directory I select

Reason: The directory is outdated or does not contain a Tomcat Server structure

Resolution: Rebuild the Tomcat directory, ensure that your Web applications are deployed to the Tomcat Server

JetPackII rejects the Tomcat directory I select

Reason: The directory does not contain `bin-native/` subdirectory with the compiled Tomcat Server.

³Starting from version 5.0.1

Resolution: Recompile the Tomcat Server with Excelsior JET, ensuring that the right Tomcat directory is specified in the project

Compilation takes way too long or the compiler terminates with an insufficient memory error

Reason: Under circumstances, the JET Optimizer does not scale well for very large Web applications.

Resolution1: If you enabled the Global Optimizer, disable it as described in section [4.7.3](#)

Resolution2: Open the JET project file (.prj) with a text editor and modify it as follows:

1. add the option `-compilewithfixedheap+` to the beginning of the project
2. increase the value of the **COMPILERHEAP** equation. For example, specify
`-compilerheap=1300m`

Note: your computer should have 2+ GB of RAM to build large applications

Resolution3: Disable native compilation for some of the application's jars which you do not need to protect as described in section [8.2.3](#). Note, however, that it may negatively affect the application performance.

Chapter 9

Java Runtime Slim-Down

Note: information in this Chapter is not applicable to the Standard Edition of Excelsior JET.

9.1 Introduction

Since version 5.0, Excelsior JET features *Java Runtime Slim-Down*, a new deployment model aimed at significant reduction of the download size and disk footprint of Java SE applications.

The key idea is to select a few components of the Java SE API, which are not used by the application, and exclude them from the installation. Such components are called *detached*. For example, if your application does not use any of Swing, AWT, CORBA or, say, JNDI, Excelsior JET enables you to easily exclude (detach) the associated files from the installation.

The detached components should be placed on a Web server so that the JET Runtime could download them if the deployed application attempts to use any of the components.

The necessary support for Java Runtime Slim-Down is also added to the JET Installation Toolkit: using JetPackII (see [6.11.2](#)), you may select the detached components and assign the URL of the web server where the detached components will be located. When building an installation that has detached components, JetPackII compresses them into a single file called *detached package*, and creates it alongside of the main installation package.

Note that using Java Runtime Slim-Down is possible only if the executable(s) added to the installation have been compiled with the Global Optimizer enabled (see [Chapter 13](#)). Without that, the detaching of unused Java SE components cannot be implemented in the JET Runtime.

To summarize, using the Java Runtime Slim-Down model includes the following steps:

1. build the executable with the Global Optimizer enabled from within the JET Control Panel (see [4.7](#))
2. select detached Java SE components and assign the URL of the web server using JetPackII (see [6.11.2](#))

3. build the main and detached packages (see 6.16)
4. make the detached package available for download from the URL assigned on step 2
5. ship the main installation package to end users

This chapter contains a detailed description of what is Java Runtime Slim-Down, how it works and how to use it for your Java applications.

9.2 Java SE components

9.2.1 What is a detachable Java SE component

A detachable component is an API which is a part of the Java SE platform. Excelsior JET allows you to detach the following components: AWT/Java2D, CORBA, JDBC, JNDI, JSound, Management, RMI, Swing, XML, and Scripting (Rhino Javascript engine).

A component may comprise:

- public API classes
- implementing classes (one or more packages)
- native method libraries and associated resource files, such as fonts

Using JetPackII, you may inspect what items are included in each component. For that, open the **Detached component configuration** dialog (see 6.11.2) that displays a tree of all detachable components and expand the respective tree nodes.

Note that contents of the detachable components are pre-defined and may *not* be changed. When configuring a detached package, you simply select items from the list but you may *not* add a new detachable component. Excelsior JET imposes the restriction to guarantee flawless execution of Java applications deployed in this mode.

9.2.2 Used and unused Java SE components

JetPackII helps you identify the Java SE components that are not used by your application. When you open the **Detached component configuration** dialog (see 6.11.2), each component's name is displayed with a color tag. The legend of the tags is as follows:

- - component is unused, that is, the application code *does not reference* any items from this component
- - component is (partially) used, that is, some items from this component are referenced

In order to detect referenced items, Excelsior JET uses:

- whole-program static analysis performed by the Global Optimizer (see Chapter 13)

- results of application profiling during Test Run (see 4.6)

If you see that a component is marked as used but you are not sure that your application really uses it, click the red tag on the right to the component's name. JetPackII will open an information dialog that shows all referenced classes from this component. Check your application code including the code of the third-party libraries listed in the application's classpath, if any, to find the places from where the classes are referenced.

9.2.3 How to identify unused components

The Java SE components marked with the green tag are not used by your application and can be detached without a doubt.

Under certain circumstances, you may also detach a component marked with the red tag. When you try to exclude such a component, JetPackII will open a warning dialog. Click **Details** to see the list of component's classes that your Java application references and, if applicable, the native method libraries that it loads at run time. As JetPackII *always* places the referenced items into the main installation package, only the remaining part is actually detached when excluding such a component. Therefore, if only a few classes from a Java SE component are referenced, in most cases, it can be detached just like an unused component.

However, care must be taken. If the application references lots of classes from a component, the remaining (unreferenced) classes may also be used implicitly, e.g. via the Reflection API. Moving such components to the detached package may result in its downloading at run time.

If you encounter an issue with unexpected downloading of the detached package, refer to section 9.5 that describes possible reasons and resolutions.

9.3 Deployment

Before starting distribution of an application that uses Java Runtime Slim-Down, you should make the detached package available for download from the assigned web server.

When creating an installation, you should specify the URL where the detached package will be located. In JetPackII, you set the base URL using the **Detached component configuration** dialog (see 6.11.2) and choose the exact name of the detached package on the **Final** page (see 6.16).

For example, if the base URL is

```
http://www.companyname.com/downloads
```

and the name of the package is

```
detached.pak
```

the file should be available from

```
http://www.companyname.com/downloads/detached.pak
```

Note also that target systems should have a network connection for possible download of the detached package.

9.4 Execution

Of course, if the deployed application does not attempt to use a component from the detached package, it works as if the entire Java SE Runtime was installed. Otherwise, if the application loads a class or native method library located in the detached package, the JET Runtime performs the following actions:

1. The detached package is downloaded from the URL you specified when creating the installation; if the downloading fails due to an unstable connection or absence of the requested file on the server, the JET Runtime throws one of the following VM exceptions:
`NoClassDefFoundError`: *<class name>*, if the thread attempts to load a class
`UnstatisfiedLinkError`: *<library name>*, if the thread attempts to load a native method library
2. Upon successful download, the file is placed into the system temporary directory, the SHA digest of the file is computed and checked against the digest hard-wired into the application executable by JetPackII when building the installation; If the check fails, the JET Runtime throws a VM exception as described above.
3. The downloaded file is unpacked into a subdirectory of the system temporary directory and then deleted
4. The requested item (class or native method library) is loaded into the application process. Note that the detached package contains the classes in the bytecode form (`.class` files) so they are loaded and compiled by the JIT compiler that comes with the JET Runtime (see Chapter 11 for details.)

By default, the unpacked files are not removed at application exit. As a result, they will be reused on next application launch without the downloading of the detached package. If however, the cached files are deleted or corrupted, and the application accesses the detached package, the JET Runtime repeats steps 1 through 4 described above.

If you do not want the application to cache the unpacked files, clear the **Cache downloaded files** checkbox in the **Detached component configuration** dialog (see 6.11.2) dialog or set the system property (see 10.5.1) `jet.remote.cleanup`. In such case, the Runtime will remove the unpacked files (if any) upon application exit.

9.5 Runtime Slim-Down HOWTO

This section covers some issues related to using Java Runtime Slim-Down.

Despite I only excluded the components which JetPackII marked as unused, the JET Runtime downloads the detached package at run time

Reason 1: The components you have excluded were unused at the time the JetPackII project was initially created. Then, you have added new classes or third-party libraries to the application and some of the components may have become used. JetPackII and the `xpack` (see 6.21) utility issue a warning in such case but you may have missed it.

Resolution: Open the project with JetPackII, go to the **Runtime** page, click **Configure** in the **Detached components** pane and check if a detached component is marked as used with the red tag. If this is the case, clear the component's checkbox to place it to the main installation package.

Reason 2: The deployed application loads a plug-in that makes use some of the detached components.

Resolution: Open the project with JetPackII, go to page **Trial Run**, run your application and follow the scenario that causes downloading of the detached package. Upon application exit, JetPackII will issue a warning listing the detached Java SE components that were accessed at run time. Cancel exclusion of the components from the installation as described above.

Note that you can assist JetPackII in automatic detection of such components by performing a thorough test run (see 13.4.2) before compiling the application.

How do I figure out what particular items from the detached components are accessed by the application at run time?

Set the system property (see 10.5.1) `jet.remote.log` to enable logging of the access trials to the console. Run the application and follow the scenario that causes downloading of the detached package. The names of accessed classes and/or native method libraries will be logged.

Target system has a network connection configured with a proxy-server. How to set up the deployed application so that the JET Runtime could access the detached package?

Excelsior JET Runtime uses the standard Java networking API for downloading the detached package. To configure proxies for the installed Java application, set the following Java system properties (see 10.5.1):

```
http.proxyHost, http.proxyPort, or  
ftp.proxyHost, ftp.proxyPort
```

For more details, refer to the Java SE platform documentation available at

<http://java.sun.com/j2se/1.5.0/docs/guide/net/properties.html>

I already use Java Runtime Slim-Down. Is it possible to further reduce the download size of my Java application?

Option 1: Tune the compiler options that control the size of the resulting executable. The major effect can be achieved by setting the "Low" or "Tiny methods only" policy for the inlining optimization (see [4.8.1](#) for details.) Then, re-compile your application and create the installation.

Option 2: If your application uses Swing or AWT/Java2D, consider replacing them with an alternative GUI toolkit such as LWJGL or SWT. Having it done, you will be able to detach the Swing and AWT components from the main installation package that will result in a substantial saving in the download size.

Chapter 10

Application considerations

10.1 Implementation-specific differences between JVMs

Each released version of Excelsior JET passes the Java Compatibility Kit (JCK) test suite. This guarantees that all features of the Java programming language and Java SE platform are implemented in compliance with the specification. However, certain applications exploit Java features not enforced by the specification. Such applications may fail to run on the JET Runtime, and probably on some other JVMs too.

This section lists the known implementation differences between Excelsior JET and the HotSpot JVM, the reference implementation included with the Sun-Oracle JRE. You are encouraged to remove all possible usages of implementation-biased features to make your application compatible with any compliant Java SE VM.

VM vendor

The first obvious distinction is that the `java.vm.vendor` system property has a different value. Surprisingly, certain third-party Java components use it to determine the "capabilities" of the underlying JVM.

How to fix: You may try to find the latest versions of such components, which may have been improved to work with a larger set of Java SE VMs.

Order of reflected methods and constructors

The elements of arrays returned by `Class.getDeclaredConstructors()` and `Class.getDeclaredMethods()` are not sorted and are not in any particular order.

This is exactly what the Java SE API specification says rather than a feature of Excelsior JET. Nevertheless, certain third-party Java components try to exploit a particular order of reflected class members.

How to fix: You may try to find the latest versions of such components, which may have been improved to work with a larger set of Java SE VMs.

Deprecated methods of class Thread

`Thread.stop()` has no effect unless the thread is waiting on an object. `Thread.suspend()` may provoke a deadlock in the JET Runtime.

Note that `Thread.stop()`, `Thread.suspend()`, and `Thread.resume()` are deprecated methods. Specifically for Excelsior JET, it is *strongly* recommended to rewrite the code that uses these deadlock-prone methods.

How to fix: A detailed explanation and recommended replacements are given in the article "*Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?*" available at

<http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html>

Eager class resolution

When compiling a Java application, the JET Optimizer tries to resolve almost all references from each class to other imported classes. On the contrary, HotSpot exploits lazy class resolution, that is, a reference to a class is resolved only when it is first used. Both resolution policies are allowed by the Java specification.

Reliance on lazy class resolution in the application code may cause subtle side-effects and, in general, results in creation of fragile code. It is better to avoid using speculative assumptions on when a particular class will be loaded.

Workaround: If the code exploiting lazy class resolution is not pre-compiled, that is, it is handled by the JIT compiler included with the JET Runtime, you may enable the `jet.jit.disable.resolution` VM option (see 10.5.1) that enforces lazy class-loading in the JITted classes.

No JVMTI

JVM Tool Interface (JVMTI), an optional part of the Java SE platform API, is not supported in Excelsior JET due to security reasons. Its implementation would enable a wide variety of introspection techniques for the compiled applications, which is highly undesirable.

Limited support for Java instrumentation agents

Despite the JET Runtime recognizes the `-javaagent` option, it takes no effect on pre-compiled classes. Thus, instrumentation is done only for the classes loaded by JIT compilers that come with the JET Runtime (see Chapter 11).

No class/jar files

To provide code protection, the original class/jar files are not deployed with the compiled application, and therefore, are not available at run time. If the application tries, for example, to compute a check sum of a class by reading its content with conventional file system operations, it would fail.

Workaround: You may configure the JET Optimizer to make the required jar file available to the compiled application, provided they do not contain the code you wish to protect from decompilation. For more details, see section 10.13.

Note: Some applications, such as the `javac` compiler, need only symbolic information from class files and do not use the contained bytecode instructions. In such cases, everything should work without any workarounds.

Standard Exception Messages

Detailed messages of the exceptions thrown by the JVM, such as `IndexOutOfBoundsException`, may differ from those of HotSpot.

Stack trace

By default, `Throwable.printStackTrace()` methods print a few fake elements, because stack tracing is disabled in the JET Runtime. However, certain third-party APIs may rely on stack trace printing. For example, the Log4J API uses the stack trace information providing logging services.

Workaround: You may enable stack trace printing as described in section 10.11.

Default heap size

If you do not set the maximum heap size, the JET Runtime determines it adaptively depending on the allocation profile and amount of available physical memory as described in section 10.6.

The HotSpot JVM uses different heuristics to set the default maximum heap size. For more details, see

<http://java.sun.com/j2se/1.5.0/docs/guide/vm/gc-ergonomics.html>

Workaround: You may set the maximum heap size explicitly (see 10.6.)

Signed jars

As, by default, the application jar files are not deployed with the compiled executable, checking digests of the signed jars fails.

Workaround: You may configure the JET Optimizer to make the signed jars available to the compiled application as described in section 10.13.

Note: In such case, the jars will probably be used only for checking the signatures which does not help ensure the authenticity. For that purpose, it would make more sense to sign the compiled executable using a third-party utility.

Non-standard VM options

Most `-XX` options are not supported as they are specific to a particular implementation. Moreover, certain `-X` options are not supported too, for example setting `-Xbootclasspath` takes no effect. The list of the supported VM options is given in section [10.5.2](#).

Endorsed jars

If your application uses Endorsed Standards Override, it is not enough to specify the `java.endorsed.dirs` system property as you do when using the HotSpot JVM. To enable the override mechanism, you need to use the JET Setup utility to create a new JET profile including the endorsed jars. For more details, see [2.3](#).

10.2 Runtime Selection

<p>Note: information in this section is not applicable to the Standard Edition of Excelsior JET.</p>

Excelsior JET VM comes with multiple implementations of the runtime system, optimized for different hardware configurations and application types. It enables your application to effectively utilize the computing power of systems that support parallel execution¹, thereby improving performance.

This section describes the available runtime systems and helps you select the proper runtime for your applications and target system configurations.

10.2.1 Available runtimes

Desktop Runtime

The Desktop Runtime is suitable for applications that typically run on conventional desktop and notebook computers. It is optimized for *single-CPU systems*, including those based on multi-core chips. This is the best choice for rich clients, visualization and engineering design tools, and other desktop applications.

¹Examples are multi-processor servers, multi-core chips and CPUs that support Hyper-Threading Technology

Server Runtime

The Server Runtime fits best for highly concurrent server applications. It provides out-of-the-box performance and scalability and takes full advantage of the computing power of multi-processor hardware. Note that Excelsior JET, Enterprise Edition is the only retail version that includes the Server Runtime.

Classic Runtime

The Classic Runtime is designed to use on low-end hardware which does not support parallel execution such as uniprocessor systems equipped with old CPU models of the x86 architecture. It is not recommended for use on HyperThread/multi-core CPU and multi-processor systems.

Note that the Classic Runtime is the only option in the Standard Edition of Excelsior JET.

10.2.2 Which runtime fits best for my application?

Basically, you select an Excelsior JET Runtime depending on the type of your application and hardware configurations of the systems to which it will be deployed. To make the right choice, follow the recommendations below.

Server-side applications

If your server application will be deployed to multi-processor machines, opt for the **Server Runtime**. Other Runtimes are not optimized for N-way systems so choosing them would significantly degrade throughput of server applications.

Client-side applications

The **Desktop Runtime** is the optimal choice for desktop applications running on single-CPU systems.

Note that using the **Classic Runtime** on the systems that support parallel execution, ranging from Intel P4 powered by the Hyper-Threading technology to Intel Core 2 Duo/Quad and up, may cause *significant* performance degradation.

Embedded applications

If your embedded platform is equipped with a CPU that does not support parallel execution², the Classic Runtime fits best. Otherwise, use the Desktop Runtime.

You may experiment with each runtime flavor as described in the next section.

²For instance, a Pentium II-class processor

10.2.3 How to enable a particular runtime

You select the runtime when compiling your application. The JET Control Panel shows the available RT options on the **Target** page (see 4.9.2). If you use the command-line interface to the JET compiler, specify the **JETRT** equation in the project file:

```
-JETRT=DESKTOP
```

Finally, if you run your application via the **xjava** launcher, use the `-Xrt` option to select the desired runtime (see 11.7.2):

```
xjava -Xrt=server -jar MyServer.jar
```

You may also select the runtime at application launch time specifying the `jet.rt` property (see 10.5.1). For example, to enable the Classic Runtime, specify

```
-Djet.rt=classic
```

The valid values are `classic`, `desktop`, and `server`.

Note, however, that the `jet.rt` property can only force lower-end runtimes³ than the runtime selected at compile-time. For instance, if you chose the Desktop Runtime when compiling your application, you may force the Classic Runtime during its launch, but not the Server Runtime. If you wish to test your application against all runtimes, enable the Server Runtime when compiling and then select other runtimes using the `jet.rt` property as desired.

10.3 Code performance

If you used any optimizing compilers before, you must have experienced a situation when a particular combination of the many available optimization control options results in incorrect code being generated by the compiler for your program, or when the program only works properly when compiled with debug information generation enabled. The reason is that a compiler with dozens of optimization control options is extremely difficult and expensive to test, as the compiler manufacturer has to check all possible combinations of options, so no vendor does that.

Fortunately, this is not the case with JET. Unlike many other compilers, JET always optimizes code to the maximum extent possible. Even enabling generation of debug information to ease field engineering does not affect the output code. The only two optimizations that may be disabled or adjusted are stack allocation of objects and inline expansion of methods, because they can negatively affect scalability (see 10.4.2) and code size of the resulting program.

To get the maximum performance, do the following:

1. Enable stack allocation of objects:
 - Using the Control Panel: On the **Options** page, check the **Allocate objects on the stack** box.

³In the sense of this order: Classic < Desktop < Server

- In the project file: turn the **GENSTACKALLOC** option ON:
+GENSTACKALLOC

Stack allocation is enabled by default. See [4.8.1](#) and [10.4.2](#) for more information.

2. Enable inline expansion of methods:

- Using the Control Panel: On the **Options** page, select an item other than “OFF” from the **Inline expansion** list.
- In the project file: turn the **INLINE** option ON and optionally adjust **INLINELIMIT** and **INLINETOLIMIT** equations:

```
+INLINE
-INLINELIMIT      = 200    % default is 100
-INLINETOLIMIT    = 2000   % default is 1000
```

See [4.8.1](#) and also comments below for details.

Inline expansion should be used with care. It allows the compiler to substitute a method in-line, if that method is declared as final or if compile time analysis confirmed safety of such substitution. As a rule, enabling inline expansion results in faster but bigger code. However, the effect of this optimization depends on the programming style (sizes of methods, etc). You may control inlining aggressiveness using equations **INLINELIMIT** and **INLINETOLIMIT** in the project file, or by selecting one of the presets in the **Inline expansion** list in the Control Panel.

10.4 Scalability

10.4.1 Multiprocessor systems

Excelsior JET is fully SMP-aware, so your parallel applications will scale to multiple processors. However, you should select the Desktop or Server Runtime to enable the most effective utilization of several CPUs (see [10.2](#)). You may wish to fine tune the garbage collector as described in [10.6.5](#).

10.4.2 Number of threads

Excelsior JET runtime maps Java threads directly onto native operating system threads.

The default size of stack reserve is about one megabyte, which basically means that if your program attempts to run 1,000 threads simultaneously (which is not unusual for server-side applications), their stacks will claim *one gigabyte* of the process’ virtual address space, of the total of four gigabytes. Given that some part of the virtual address space is reserved by the operating system, one process cannot run more than a few thousand threads simultaneously with that default setting.

An attempt to run too many threads simultaneously in a Java application results in `OutOfMemoryError`. Waste of address space for thread stack reserves also effectively

limits the amount of memory that the memory manager may allocate from the OS and thus may also cause `OutOfMemoryError`.

To reduce the default stack size and thus improve the scalability of your application with respect to the number of threads:

- Using the Control Panel: On the **Options** page (under the **Advanced** tab), drag the **Stack limit** slider or type the desired value into the respective entry field.
- In the project file: Specify the equation **STACKLIMIT**, which sets the size of the thread stack in bytes (rounded up to the next 4KB boundary):

```
-STACKLIMIT=65536
```

This is equivalent to setting the `-Xss` option of the standard Java launcher.

Another thing to consider is that enabling stack allocation of objects increases stack size requirements of the compiled applications, which may negatively affect scalability. Therefore, you may wish to disable this optimization by unchecking the **Allocate objects on the stack** box on the **Options** page (see 4.8.1) or turning the **GENSTACKALLOC** option off in the project file. This would enable you to further reduce the thread stack size as described above, thus letting your application run more threads simultaneously at the cost of minor performance loss.

10.5 Java system properties

A Java launcher allows you to set a system property using the `-D` command-line switch, for instance:

```
java -Dp1name -Dp2name=p2value MyApp arg1 arg2
```

A JET-compiled application is a conventional executable file that is run by itself:

```
MyApp arg1 arg2
```

so there is no place to set a property on the application's command line.

10.5.1 How to set system properties

JET provides three ways to set system properties for your compiled application:

At compile time:

You may hardwire system properties into the resulting executable as follows:

- Using the Control Panel: On the **Start** page (see 4.5.2), type the desired system properties into the respective field.
- In the project file: Use the **JETVMPROP** equation:

```
-JETVMPROP=-Dp1name -Dp2name=p2value
```

At install time:

You can override properties hardwired into the deployed executable, if any, taking into account the target system parameters. See 6.10.2 and 6.21.2 for details.

At launch time:

If you build a multi-app executable (see 10.7), you may specify the system properties and JET Runtime options directly on the application command line.

Otherwise, use the `JETVMPROP` environment variable. For the above example, you would use the following commands:

(bash/Bourne shell)

```
export JETVMPROP="-Dp1name -Dp2name=p2value"
./MyApp arg1 arg2
```

You may omit the double quotes if you need to set exactly one property:

```
export JETVMPROP=-Dp2name=p2value
./MyApp arg1 arg2
```

Warning: The `JETVMPROP` environment variable is intended to be used during development and field engineering. Normally, you should hardwire the necessary system properties into the executable at compile time or install time as described above. If you need to override the hardwired properties at launch time, use a batch file for launching your application. Never set the `JETVMPROP` environment variable globally (i.e. in your login script or the system startup script) on enduser systems as it may cause other JET-compiled applications to misbehave.

If the `xjava` utility (see 11.7) is used to run your application, you can either use the `JETVMPROP` environment variable or follow the standard `java` command line syntax for setting the properties:

```
xjava -Dp1name -Dp2name=p2value MyApp arg1 arg2
```

Note: On application startup, properties set through the `JETVMPROP` environment variable are *merged* with hardwired properties. If a particular property is set using both methods, the value retrieved from the environment variable shall prevail.

10.5.2 Standard runtime options

The JET Runtime recognizes the following standard options:

- Assertion directives:
`-ea`, `-da`, `-enableassertions`, `-disableassertions`, `-esa`,
`-dsa`, `-enablesystemassertions`, `-disablesystemassertions`
- Maximum stack and heap size:
`-Xss`, `-Xmx`
- Enabling strict verifier:
`-Xverify:all`

- Setting maximum memory size for direct buffers:
-XX:MaxDirectMemorySize
- Specifying a Java Agent (for non-compiled classes):
-javaagent:

Note: currently, JET Control Panel does not recognize this option when parsing Java command line but you may set it manually as described in [10.5.1](#). See also [10.1](#).

- -version, -verbose:gc

10.5.3 JET Runtime-specific properties

In addition, the JET Runtime recognizes the following properties:

`jet.gc.heaplimit=size[k|m]`

Sets the maximum heap *size* to the given amount of bytes (kilobytes or megabytes respectively if either of the suffixes *k* or *m* is present), overriding the compile-time heap size setting. Setting *size* to zero enables adaptive heap size. See [10.6](#) for more information.

`jet.gc.ratio=ratio`

Specifies the maximum proportion of CPU time to be allocated to the garbage collector at runtime, in 1/1000ths. See [10.6.1](#) for details.

`jet.rt=flavor-name`

Select a specific JET Runtime. Valid values of *flavor-name* are `classic`, `desktop`, `workstation`, and `server`.

See [10.2](#) for details.

`jet.gc.threads=N`

Sets the maximum number of concurrent threads that may be used for garbage collection. By default, *N* equals the number of processors/cores on the system. See [10.6.5](#) for more information.

`jet.jit.cache`

Enables caching of JIT compilation results. See [11.4](#) for more information.

`jet.jit.cache.dir=path`

Designates *path* as the JIT compiler cache directory. See [11.4](#) for more information. This property is effective only if caching of JIT compilation results is already enabled.

`jet.jit.heap.size=value`

Sets maximum amount of memory the JIT compiler is allowed to use. Default value is 128 megabytes. Use this option to increase amount of JIT memory if an `OutOfMemoryError` is thrown as a result of JIT compilation.

`jet.log.dlls`

Logs names of shared libraries loaded by the application at run time into a file.

`jet.stack.trace`

Enables stack backtracing facility. See [10.11](#) for more information.

`jet.jit.optimizing`

Forces the optimizing JIT compiler. See [11.4.1](#) for more information.

`jet.jit.fast`

Forces the fast JIT compiler. See [11.4.1](#) for more information.

`jet.jit.disable.resolution`

Forces the JET Runtime to postpone compilation of dynamically loaded classes to the latest possible moment (i.e. a *lazy resolution* technique is used.) Specifying this option reduces JIT pauses at the cost of possible performance degradation for JIT-compiled classes.

`jet.runtime.error.message=message-text`

Allows customizing the error messages printed after a JET Runtime crash. For more details, refer to section [10.17](#).

10.6 Memory management

Excelsior JET lets you define the policy of memory management to be used for your Java application at run time. This section describes a few simple mechanisms that you can employ to set an optimal balance between performance of the application and its memory footprint.

10.6.1 GC ratio

Ideally, an application should work fast enough and consume as few memory as possible. In practice, there are always trade-offs between application performance (or throughput) and its memory footprint. What's especially important is that performance and memory requirements are application-specific. For one program, performance is critical, whereas low memory footprint has more value for the other. Put simply, one size does not fit all.

Fortunately, the JET Runtime lets you fine tune the performance/memory balance for your Java application. For that, you just set a ratio in which the Runtime will portion out execution time between the two tasks: *execution of the application's code* and *garbage collection*. This technique is simple and effective:

- By assigning GC a greater time share, you increase the frequency of GC invocations which does not allow the Java heap to excessively grow in size.

- And vice versa, by lowering the ratio of GC in execution time, you reduce the GC overhead thereby improving performance at the cost of higher memory consumption.

You can do that by specifying the *GC ratio* setting to the JET Runtime. Its value is the tenths of percent of total CPU time that may be spent for garbage collection. For example, the default GC ratio is 1.1% which means that GC will normally take the percentage in total execution time. You can vary the GC ratio as follows.

- Using the JET Control Panel: adjust the **GC ratio** control on the **Options** page (see 4.8.2); or
- Set the system property (see 10.5.1) `jet.gc.ratio`. For example, to set the ratio to 5.0%, specify

```
-Djet.gc.ratio=50
```

The maximum GC ratio you can set is 40.0%.

Note: changing the default GC ratio is not available in the Standard Edition of Excelsior JET.

The default setting (1.1%) is biased to performance and, therefore, may cause excessive memory consumption in some cases. You can experiment with your Java application by fine tuning the GC ratio. Often, increasing the value by several units may noticeably reduce memory footprint. That's why the unit of measure is 1/10th of percent not a whole percent.

One might question: what reduction in memory footprint can I expect when increasing the GC ratio by X%? In general, it depends on two factors: *how much garbage* your application produces and *how fast* it does that. Thus, the answer is specific to a particular application.

Note that under certain circumstances, the JET Runtime is unable to meet the GC ratio requirement. For example, if a Java application allocates objects very intensively and amount of available physical memory is low, more CPU time may be spent in the garbage collector regardless of the ratio specified.

Finally, you may set the GC ratio to 0 to entirely disable the mechanism (it's not recommended, though.) In such case, GC is automatically invoked only if the JET Runtime is unable to serve a regular request for object allocation due to memory constraints as described in the following sections.

10.6.2 Maximum heap size

You may specify the maximum amount of heap memory that your application is allowed to allocate at run time⁴:

- Using the Control Panel: adjust the **Maximum Heap Size** control on the **Options** page (see 4.8.2)

⁴The respective command line parameter of the standard Java launcher is `-Xmx`

- In the project file: specify the value of the `jet.gc.heaplimit` property in the **JETVMPROP** equation:

```
% maximum heap size is 30MB
-JETVMPROP="-Djet.gc.heaplimit=30m "
```

To override the compile-time setting, you may set the system property at install time or even at launch time (see [10.5.1](#)).

Note that you need to specify the maximum heap size only if you do not want the Java heap to exceed certain upper bound in any case. On the one hand, it may be useful to reduce application memory footprint. On the other hand, it may compromise performance of your application. For example, it might work more effectively when using large Java heaps on the target machines that have high RAM capacity.

The default value of the maximum heap size is 0 which means *Adaptive*. In this mode, the Java heap is only bounded by the size of Virtual Address Space that varies from 2 to 4GB, depending on the underlying operating system. But do not be afraid. It does not mean that your application will take gigabytes of memory at run time. First, the JET Runtime checks the GC ratio (see [10.6.1](#)) which effectively damps inflation of the heap. Moreover, the runtime constantly monitors the amount of *available physical memory* as described in the next section.

10.6.3 Handling low memory

As a rule, it's hard to determine the "right" application memory settings because they are specific to a particular target system. For instance, you consider a box equipped with 256MB of RAM as a "typical target system" but some customers are proved to be still using 128MB machines. Another example is a system under heavy workload: imagine that your application runs simultaneously with a couple of other programs hungry for memory. Anyway, your application must work, more or less efficiently.

To address these problems, the JET Runtime considers the amount of *available physical memory* in the system to schedule garbage collection. If low memory is detected when allocating an object, GC is activated to avoid unnecessary swapping RAM pages to the hard disk. Of course, it leads to more frequent GC invocations and penalizes application performance. But it's a good alternative to tremendous overheads of swapping memory to and from.

However, if the runtime detects that application execution has degraded to sole garbage collection or it's unable to allocate an object even after GC invocation, the physical memory boundary is crossed. The JET Runtime augments the Java heap with the amount of memory enough to serve the allocation request plus some decent gap necessary for further operation. Often, this move helps to clear the air. The operating system reacts to the event and fetches memory, e.g. from the system cache, so that the application continues running smoothly.

You have control over the JET Runtime with respect to crossing the physical memory threshold. To override the default behavior

- Using the Control Panel: On **Options** page (see [4.8.2](#)), select **throw OutOfMem-**

oryError from the **Handling low memory** combo box; or

- Specify the system property (see 10.5.1) `-Djet.gc.no.swap`.

As a result, `OutOfMemoryError` will be thrown instead of using virtual memory to increase the heap. This mode is useful if you suspect that your application has performance issues due to not enough memory in the system. You may easily prove or disprove that by disabling the default policy and handling `OutOfMemoryError` exceptions, if any.

10.6.4 Large heaps

The theoretical limit of the heap size for 32-bit applications is 4 gigabytes. In practice, however, a part of the virtual address space is occupied by the operating system, thread stacks, program executable and dynamic libraries, etc. As a result, the application cannot allocate a multi-gigabyte heap *even if the system has enough physical memory*.

The upper bound of the maximum possible heap size is *O/S-dependent*. If you run a JET-compiled application on 64-bit Windows or Linux, it is able to allocate up to 3GB of heap memory without hassles. However, care must be taken if the system requirements of your Java application include 32-bit Windows and/or Linux flavours. Typically, all the systems allow allocation of a 1.5 GB heap out-of-the-box provided the machine has enough RAM. This can be considered as the “largest common denominator”.

Certain 32-bit operating systems, e.g. Windows 2003 Server, can be configured to enable allocation up to 3 GB heap as well. For more details, refer to

<http://www.microsoft.com/whdc/system/platform/server/PAE/PAEmem.msp>

In addition, some Linux distros provide an option to build the kernel in a special mode that makes approx. 4GB address space available to the process. For example, RedHat Enterprise Linux 3 and higher include a kernel known as the *hugemem* kernel. Note that this incurs some overhead when transferring from user to kernel space, e.g. in the case of system calls and interrupts.

10.6.5 Parallel hardware

If your target systems have multiple CPUs, or support parallel execution via multi-core or HyperThread technologies, select the appropriate version of the JET Runtime as described in section 10.2. The right choice of Runtime has a great impact on performance of applications that use the Java heap intensively.

Moreover, on the parallel hardware, the garbage collector runs in multiple threads to reduce collection pauses. By default, the total number of concurrently working GC threads equals the number of CPUs/cores reported by the operating system. However, you may reduce this number for optimum system performance, e.g. if you want one or more CPUs to be available to some native threads or processes running at the same time.

For that, set (see 10.5.1) the value of the `jet.gc.threads` property to the maximum number of GC threads. For instance, to make the garbage collector use at most two threads, specify the following setting:

```
-Djet.gc.threads=2
```

Note: parallel garbage collector is not available in the Standard Edition of Excelsior JET.

10.7 Multi-app executables

Note: information in this section is not applicable to the Standard Edition of Excelsior JET.

New in JET 7.0:

Excelsior JET may compile more than one application into a single executable and let you select a particular application at launch time via command line arguments. This enables you to:

- easily compile and package several applications that share common libraries
- use the Global Optimizer (see [13](#)) for more than one application at once
- specify Java system properties and JET Runtime options on the command line

You enable the multi-app mode:

- Using the Control Panel: check the **Multi-app executable** box on the **Target** page (see [4.9.5](#)).
- In the project file: turn on the **MULTIAPP** option:

```
-MULTIAPP+
```

10.7.1 Command-line syntax

The command line syntax of multi-app executables is an extension of the `java` launcher command line syntax that allows specifying the main class, VM options, Java system properties, and the arguments of the application:

```
Exe-name [Properties-and-options] Main-classname [App-arguments]
```

For example,

```
MyApp com.mycompany.app.Main
```

Runs the multi-app executable with `com.mycompany.app.Main` as the main class.

```
MyApp -Xmx256m -Dmy.prop=val com.mycompany.app.Main
```

Runs the multi-app executable with the specified main class and sets a runtime option (maximum heap size) and a Java system property.

```
MyApp -Xmx256m -Dmy.prop=val com.mycompany.app.Main 10 1
```

The same as above but also specifies two application arguments.

Note: If the specified main class does not exist, `NoClassDefFoundError` is thrown and the application terminates.

10.7.2 Default main class

When you simply run a multi-app executable without arguments, the main class and Java system properties set at compile-time will be used.

If you need to specify command line arguments or system properties for the application without changing the default main class, use the following syntax:

```
Exe-name [Properties-and-options] [-args App-arguments]
```

Here go a few examples of the well-formed command lines:

```
MyApp -args 10 1
```

Runs the multi-app executable with two application arguments.

```
MyApp -Xmx256m -Dmy.property=value
```

Runs the multi-app executable with no arguments and sets a runtime option (maximum heap size) and a Java system property

```
MyApp -Xmx256m -Dmy.property=value -args 10 1
```

The same as above but also specifies two application arguments.

Notes:

1. Certain options supported by the standard `java` launcher may be also specified on the command line of multi-app executables. You may find the list of such options recognized by multi-app executables in section 10.5.2. Note that some irrelevant options such as `-cp`, `-jar`, `-Xbootclasspath` are not supported and will be ignored.
2. The `JETVMPROP` environment variable is ignored for multi-app executables.
3. On application startup, the properties set on the command line are *merged* with the hard-wired properties specified at compile-time (see 10.5.1). If a particular property is set using both methods, the value retrieved from the command line prevails.

10.7.3 Splash screen

If the application compiled as a multi-app executable has a splash screen specified via the compile-time settings (see 10.10.1), the splash will be displayed as usual.

If you need to suppress the splash screen, e.g. when running the executable with a main class other than the default, specify the `-Xnosplash` option in the *Properties-and-options* section of the command line.

You may also wish to replace the pre-configured splash screen image with another one. To do that, use the `-splash` option specifying the pathname of the desired image, for example

```
MyApp -splash:another-splash.gif com.mycompany.app.Main
```

10.8 Disk footprint

Note: information in this section is not applicable to the Standard Edition of Excelsior JET.

With Excelsior JET, you can optimize *disk footprint* of your Java application, that is, reduce the amount of disk space required by the application installed on target systems. This can be useful if the application files should be placed into flash memory, which is a typical requirement for contemporary embedded systems. Another use case is running a Java application off a USB flash drive without installation.

Excelsior JET includes the Global Optimizer (see Chapter 13) that performs static analysis and uses the profile information collected during Test Run (see 4.6) to figure out what application and library classes are rarely used or not used at all during execution.

The idea behind the saving of disk space is not to compile the classes that will unlikely be loaded at run time: such class files are compressed when creating the installation package and then *installed in a compressed form* on the target system. This technique helps reduce disk footprint dramatically with a minimal impact on application start-up time and performance in most cases.

If, however, the application tries to load a class placed into the compressed bundle, the JET Runtime inflates (a part of) the bundle to access the requested class transparently to the running application. This technique is called *on-demand decompression*.

There is a trade-off between the effectiveness of disk footprint reduction and possible overheads that on-demand decompression may impose. You have two options to balance the overheads as described below.

10.8.1 Medium reduction

The medium reduction method saves up to 15 MB of disk space. If on-demand decompression occurs, the JET Runtime inflates only *a part* of the compressed bundle to get the requested class file. As a result, this reduction method has little impact on start-up time/performance and does not increase memory usage.

10.8.2 High reduction

The high reduction method saves up to 30 MB of disk space using so called solid compression. That is, if on-demand decompression occurs, the *entire* bundle is inflated and temporarily stored to make its contents available to the Runtime.

Note: inflating the highly compressed bundle takes up to several seconds and thus pauses application execution until the data are decompressed. Refer to section 10.8.4 that describes how to achieve non-disruptive execution.

Two techniques of storing the uncompressed data are implemented:

- *into memory* — the necessary structures are created on the heap. Note, however, that the memory is reclaimed by garbage collector as the uncompressed data are no longer required. Therefore, on-demand decompression into memory may increase peak memory usage but not application memory usage on average.
- *on disk* — the inflated bundle is stored in temporary files that will be automatically deleted by the JET Runtime at application exit. The advantage of this method is that on-demand decompression occurs at most once per application run session and peak memory usage does not increase. The uncompressed data files are stored in the system temporary directory. Note, however, that this technique cannot be used on the embedded systems not equipped with a hard drive.

You can choose the method of disk footprint reduction appropriate to your application using the JetPackII tool. When creating the installation, select the desired options in the **Disk footprint** panel on the **Runtime** page (see 6.11.3).

10.8.3 Complementary methods

In addition to enabling the compression of unused classes, you may exclude optional components of the JET Runtime from the resulting installation (see 6.11.)

Another possibility is tuning the compiler options that control the executable size. The major effect can be achieved by setting the "Low" or "Tiny methods only" policy for the inlining optimization (see 4.8.1 for details.)

These methods help you further reduce disk footprint of your optimized Java applications.

10.8.4 On-demand decompression

If you opt for the high reduction of disk footprint, on-demand decompression of the entire bundle *may* occur during application execution. Note that it happens only if the Global Optimizer could not detect all classes that are used at run time and some of them were placed into the compressed bundle when creating the installation.

Before deployment of your application to target systems, you can check whether the decompression occurs at run time. For that, test your application launched from within JetPackII on the page **Trial Run** (see 6.12). If the decompression has occurred, JetPackII displays a warning after completion of the trial run.

If you want to eliminate the decompression overheads, you should assist the Global Optimizer by providing a more complete profile information. To do that, open the project in the JET Control Panel and perform the test run step (see 4.6) to repeat the same testing scenarios you used when running the application from within JetPackII.

Moreover, if there are automatic tests for the application, e.g. code coverage or stress tests, you can perform the test run without invoking the JET Control Panel as described in section 13.4.2.

As a result, the profile information will be automatically completed and you can then recompile your application with the Global Optimizer enabled. To learn more about why the test run is required, refer to section 13.4.1.

If, due to some reasons, the performing of a exhaustive test run is not feasible for your application, you can select the medium reduction method to avoid possible overheads of on-demand decompression.

10.9 Creating trial versions

Note: information in this section is not applicable to the Standard Edition of Excelsior JET.

You can create a trial version of your Java application that will expire in a specified number of days *after the build date of the executable* or on a *fixed date*⁵. It means that you will need to update the download file of your application from time to time. In return, the end user will not be able to bypass the license check by changing system date at install time only⁶. As the evaluation period is over, the application no longer starts and shows the expiration message you specified when compiling it.

You enable creation of the trial version as follows:

- Using the Control Panel: on **Target** page (see 4.9.1), select the **Trial** radiobutton from the **Trial version** pane and specify the number of days for the evaluation period.
- In the project file: use the **EXPIRE** equation

-EXPIRE=*number-of-days*

or

-EXPIRE=*DayMonthYear*

For example, you may add to the project file equation

-EXPIRE=60

to create a two month trial version; or equation

-EXPIRE=15Oct2012

⁵The setting of fixed date expiration is available only through the command line compiler.

⁶To bypass the check, one should keep a back date in the system as long as your application is active. This is very inconvenient because it may provoke issues in your and/or other applications running at the same time.

to enable expiration of the trial version on the date October 15th 2012.

To specify the expiration message do the following:

- Using the Control Panel: on **Target** page (see 4.9.1), type a text of the expiration message to be shown as the evaluation period is over, in the **Expiration message** editbox from from the **Trial** pane.
- In the project file: use the **EXPIREMSG** equation:

-EXPIREMSG=*expiration-message-text*

You may use the control sequence `\n` in the message to display the following text on a new line.

10.10 Application appearance

To customize appearance of your application, you can assign a splash screen to be displayed at the very startup of the executable.

You may control this feature at compile time using either JET Control Panel or command line interface to the JET compiler as described below.

10.10.1 Splash screen

If startup of your application takes longer than you would like it to do, the thumb rule is to show a splash screen. It definitely relieves the loading process from the end user's perception, moreover, the splash may contain information about your product and company.

The splash screen functionality appeared in Java API since Java SE 6. For more details, see

<http://java.sun.com/developer/technicalArticles/J2SE/Desktop/javase6/splashscreen>

Excelsior JET extends that functionality as follows:

- A limited support of splash screen functionality for Java 5 applications;
- Advanced options to control splash behavior;
- Ease of adding a splash screen to your application.

All you need to do is to specify the necessary splash settings to the JET Runtime.

Displaying a splash

If the splash image has been specified in the manifest of the application's JAR file, the respective image will be obtained automatically:

- Using the Control Panel: make sure that a checkbox **Get image from the manifest of the JAR file containing the main method** on **Appearance** page (see 4.10.1) is selected. This checkbox is enabled if the Control Panel detects that a splash screen image is specified in the manifest the application's JAR file.

- In the project file: use the **SPLASHGETFROMMANIFEST** option:

```
-splashgetfrommanifest+
```

Otherwise, you need to assign the application a splash screen image manually:

- Using the Control Panel: type a path⁷ of the desired image file in the **Specify splash image** field on **Appearance** page (see 4.10.1) or click **Browse** to select the image using a conventional file dialog.

- In the project file: use the **SPLASH** equation:

```
-splash=path-to-the-image-file/MySplash.gif
```

The supported image formats are PNG, GIF and JPEG, including images with transparency, translucency, and animation.

You may convert images to required format using a third-party image processing tool.

Once assigned, the splash will *quickly* appear at the very startup of the application (hence the name InstantSplash.)

Suppressing a splash

There are various options to control when the splash screen image shall disappear.

An obvious behavior of the splash screen is to disappear when the main application's window is opened. However, it is less obvious how to determine what window is the main one (consider an application that opens auxiliary windows such as console, list of warnings, or some dialogs before the main window.) You may choose one of the following "auto-hide" policies:

1. Hide the splash screen when application opens *any* window, irrespective of what particular API (JFC/Swing, SWT or something else) is used. This is default.
2. Hide the splash screen when application opens a *particular* window, depending on the window title.
3. Hide the splash screen as the application opens any AWT frame. This policy is not available for Java 5 applications.

Note: By default, the splash screen disappears as the application opens any AWT frame. If you choose the other option, the following limitations apply: translucency in the splash image is not supported .

If you want to change the default policy, do the following:

⁷You may use an absolute path or path relative to the project directory

- Using the Control Panel: choose the appropriate policy in the **Auto hide** combo box on **Appearance** page (see 4.10.1). If the **When specified window opens** policy is chosen, you need to additionally specify a prefix of the window's title in the **Window's title prefix** field.
- In the project file: use the **SPLASHCLOSEONAWTWINDOW** option and **SPLASHCLOSEONTITLE** equation to select the corresponding policy:
 1. To hide the splash screen when any window opens:


```
-splashcloseonawtwindow-
```
 2. To hide the splash screen when the window with title started with a string "My main window" opens:


```
-splashcloseontitle="My main window"
```
 3. To hide the splash screen when any AWT window opens:


```
-splashcloseonawtwindow+
```

Another option that controls the splash screen behavior is a *splash delay*. You may wish to delay splash disappearance, e.g. to let the user enjoy your artwork. For that, specify the number of seconds you want it to stay on the screen:

- Using the Control Panel: check the **Show splash for** box and type the number of seconds into the next entry field on **Appearance** page (see 4.10.1)
- In the project file: use the **SPLASHMINTIME** equation:


```
-splashmintime=3
```

Note: The splash delay does not work if you opted to hide the splash screen when any AWT window opens. In that case, the **Show splash for** box is disabled and the **SPLASHMINTIME** equation is ignored.

If a delay is set, the splash works as follows. It appears at application startup and stays on the screen for the specified time. After that, the selected "auto hide" policy has effect. If the window has been already opened, the splash disappears; if not, it remains on the screen until the respective window will be opened.

Finally, you may choose whether to hide the splash screen if user clicks on it. By default, user click hides the splash screen for Java 5 applications, but does nothing for Java 6 applications. If you wish the splash screen to disappear on user clicks for Java 6 applications as well, do the following:

- Using the Control Panel: select the **Close splash with a mouse click** checkbox on **Appearance** page (see 4.10.1).
- In the project file: use the **SPLASHCLOSEONCLICK** option:


```
-SPLASHCLOSEONCLICK+
```

10.11 Stack trace

By default, `Throwable.printStackTrace()` methods print a few fake elements, because stack tracing is disabled in the JET Runtime. However, some third-party APIs may rely on stack trace printing. One example is the Log4J API that provides logging services.

The JET Runtime supports two modes of stack trace printing: *minimal* and *full*. In the minimal mode, line numbers and names of some methods are omitted in call stack entries, but class names are exact. In some cases, it is enough for stack trace-dependent APIs to work, e.g. the Log4J API operates flawlessly in this mode.

Note: Enabling stack trace may negatively impact performance, if exceptions are thrown and caught repeatedly.

In the full mode, the stack trace info includes all line numbers and method names. However, enabling the full stack trace has a side effect of substantial growth of the resulting executable size by approximately 30%.

10.11.1 How to enable stack trace

In the JET Control Panel, you enable stack trace on the **Target** page (see [4.9.4](#)).

You may also select the stack trace mode via compiler options and run-time properties. To enable the minimal stack trace, set the property (see [10.5.1](#)) `jet.stack.trace`:

```
-Djet.stack.trace
```

To enable the full mode, in addition to setting the property, toggle the **GENSTACKTRACE** option ON in your project file

```
+GENSTACKTRACE
```

and re-compile your application.

10.11.2 How to disable stack trace

For security reasons, you may wish to completely disable stack trace output in the end-user version of your application. To do that, set the option **DISABLESTACKTRACE** ON in your project file:

```
+DISABLESTACKTRACE
```

10.12 Assertions

In a JVM, assertions are disabled by default, but may be enabled for the entire application or for particular classes at launch time using the `java` command's switches `-ea`, `-da`, `-esa`, and `-dsa`. With JET, if you enable assertions using the same switches in the **JETVMPROP** equation in your project file:

```
-JETVMPROP=-ea:com.MyCompany.MyPackage
```

or control them at launch time using the JETVMPROP environment variable:

```
export JETVMPROP=-ea:com.MyCompany.MyPackage  
./MyApp
```

JET also supports enabling and disabling assertions programmatically at run time.

10.13 Resource packing

A typical Java application consists of classes that contain the application's code, and resources such as images, icons, property files, which the application loads at run time.

If the resource files are placed into one or more jar files, the compiler packs the resources into the resulting executable or shared library, almost like that is done for native applications. The JET Runtime will read the resource files during execution and use them as if they were separate files. No changes in your application's code are required to make this technique work.

Using the JET Control Panel, you may enable/disable resource packing for particular jars as described below.

10.13.1 Resource packing modes

When compiling a Java application, the JET Optimizer processes each jar file listed in the classpath and, depending on the project settings, may pack the contained resource files into the resulting executable.

For jar files, the Optimizer supports the following resource packing modes:

1. **Auto-detect:** pack resource files and non-compiled classes⁸, if any. This way, all jar contents is either compiled or packed, so the original jar file is no longer required for running the application.

This is the default mode for the jar files included in the project.

2. **None:** do not apply resource packing to this classpath entry. In this mode, neither class files, nor resource files are packed into the executable. The jar file should be distributed along with the application executable.

For example, suppose you develop a Java application extensible with plug-ins. You may disable packing the plug-in jars and distribute them along with the executable. As a result, client installation of your application can be updated by replacing the jar files, for example, with newer versions of the plug-ins. Note that the JET Runtime will JIT-compile the classes from the jar files.

⁸Non-compiled classes may appear if you enable selective optimization for this classpath entry. For more details, see [4.7.2](#)

3. **Original jar/zip:** pack the entire jar/zip file, including *all* resources and classes.

For example, consider a Java component whose implementation requires presence of the original class files at run time. For instance, Java Crypto Extensions also known as security providers check the sizes of their class files during execution. In such cases, the class files serve as *both* program code and resources. Therefore, despite all the classes are pre-compiled, you have to make them available to the running application. Packing such jar files as a whole resolves the problem.

Note: Excelsior JET does **not** support the packing of resource files that reside in directories. The reason is that directories are *writable*, that is, the application may overwrite resource files at run time, so packing them into the executable may change behavior of the program.

If your application really overwrites resource files at run time, you should deploy that directory along with your executable and ensure the directory is included in the application classpath. For more details, see [6.10](#).

Otherwise, you may place all classes and resources from the directory in a jar file and add it to the classpath. As a result, the JET Optimizer will compile the jar file in the default mode of resource packing as described above.

10.13.2 Setting resource packing options

To change the default settings, open your project file with the JET Control Panel, and go to the **Classpath** page (see [4.7](#)). The central area of the page is occupied by the classpath grid (see [4.7.2](#)). Each row of the grid corresponds to a single classpath entry — a directory or jar file. The last column named **Pack into exe**, specifies the resource packing mode for each entry. Click the cell, and select the appropriate value from the drop-down list.

If you edit project files manually, use the **PACK** equation to control resource packing. Refer to the JC reference chapter (see [16.9.4](#)) for more details.

10.14 Character encodings

When creating a profile (see [2.3](#)), JET precompiles the Java SE classes that are needed for input and output of characters in national encodings into a set of shared libraries. The idea is to place support for rarely used encodings and encodings that require large conversion tables (such as Chinese) into separate shared libraries, so as to reduce the disk footprint of compiled applications that do not use some or all of those encodings.

If your application may need to use national encodings, make sure to include the respective locales into the installation package as described in [6.11.1](#).

10.15 Bytecode consistency checks

On a virtual machine, if a certain class is imported but never used during program execution, i.e. it is never instantiated, none of its static methods are called, and none of its

static fields are accessed, then no attempt to load that class would occur and therefore the program would execute normally even if the respective class file is not present on the system or cannot be reached by the VM.

Similarly, if a class does not contain a particular method or field that is referenced in another class, execution on a virtual machine would fail only when and if control reaches the code referencing that absent field or method.

Being an ahead-of-time compiler, Excelsior JET analyzes the entire compilation set and thus is able to detect such inconsistencies. By default, they result in compilation errors. However, when an inconsistency is detected in bytecode that is not under your control, such as a third-party class library, you usually do not have a chance to remedy it quickly. If you encounter such problem, contact the vendor who supplied the inconsistent bytecode, and toggle the respective JET options to make the compiler treat bytecode inconsistencies as warnings:

- If the compiler issues error F955 "class/sym conflict", it means that a class that is part of the JDK is found in the classpath (see 4.7). Remove that class or adjust the project:
 - Using the Control Panel: On the **Options** page, check the **Ignore conflicting classes** checkbox.
 - In the project file: turn the option **IGNORECLASSDUPLICATION ON**:
+IGNORECLASSDUPLICATION
- If the compiler issues error F402 "Class C not found", and the file `C.class` is indeed not present at compile time:
 - Using the Control Panel: On the **Options** page, select **dynamic compilation** from the **Class absence results in** box.
 - In the project file: set the equation **CLASSABSENCE** to "HANDLE":
-CLASSABSENCE=HANDLE
- If the compiler issues error F020 "Undeclared identifier", make sure that you have the latest version of the respective class file. If recompiling its Java source does not help or is not possible:
 - Using the Control Panel: On the **Options** page, check the **Ignore member absence** checkbox.
 - In the project file: turn the option **IGNOREMEMBERABSENCE ON**:
+IGNOREMEMBERABSENCE

The compiler will report warnings and will insert the code throwing `java.lang.NoSuchMethodError` or `java.lang.NoSuchFieldError` for all locations where absent members are used.

Note: In some applications, imported classes unavailable at compile time become available at run time. JET may handle this situation properly if you enable the reflective shield (see 11.5) facility of the Mixed Compilation Model.

10.16 Baseline compilation

In some cases, the main JET compiler may fail to compile a particular method due to lack of memory or inability to restore the method's high-level structure required for optimizing compilation, which may be caused by bytecode obfuscation. To cope with it, the main compiler invokes the so called *baseline compiler* for such methods. The baseline compiler implements only a few simple optimizations, and thus is able to compile any verifiable bytecode.

If the baseline compiler was invoked at least once during compilation of your project, the main compiler issues a message about the number of methods processed by the baseline compiler. Additionally, it logs the information about the methods compiled with a lower optimization level. The log file, having the executable's name and the ".vaz" extension, is created in the current directory.

If baseline compilation occurs during compilation of your project, we would appreciate if you e-mail us the resulting ".vaz" log file along with the .class files listed in it. Doing that, you help us improve the quality of our product.

10.17 Customizing Runtime error messages

A JET Runtime crash, that is, an abnormal termination of application execution may be caused by bugs in the Runtime or in the application itself. The latter typically occurs due to misuses of JNI functions from within native method libraries that come with certain third-party components or APIs. Such errors are hard to diagnose because a poorly coded native method can corrupt application memory structures in any way.

If this happens on the developer system, the Runtime prints some error information followed by the message

```
Please contact Excelsior Support Dept. at java@excelsior-usa.com
```

If, however, the application terminates due to a Runtime crash on end user systems, the tail message will be replaced with

```
Please contact the vendor of this application
```

You can customize this message to print, for example, a contact information of your company's customer service. To do that, set the system property

```
jet.runtime.error.message=message-text
```

when compiling or packaging your application (see 10.5.1). You may use the \n control sequence in the message to display the following text on a new line.

Chapter 11

Mixed Compilation Model

11.1 Overview

Static compilation alone could not work for the Java applications that load classes unknown at compile time. To cope with it, the JET Runtime includes a just-in-time (JIT) compiler to support dynamic compilation of the Java bytecode. That enables you to statically compile a part of the application and remain the other part "as is" - in the form of `.class` or `.jar` files. This technique is called *Mixed Compilation Model* (MCM.) Though MCM is *always* enabled in the JET Runtime, dynamic compilation is demand-driven, that is, it occurs when the application attempts to load a class that was not pre-compiled.

A typical use case for MCM would be a Java program whose client installation is extensible with plug-ins. The core of such an application can be pre-compiled and then deployed to target systems as the native executable. If the user installs a plug-in distributed as a `.jar` file, the JET Runtime loads it using the dynamic compiler. This way, an application backed by MCM is able to correctly load the classes that were not or could not be compiled before execution.

Note, however, that dynamic compilation occurs at run time and competes with the executing program for resources. As a result, it may worsen start-up time and/or performance of the application. If this is the case, you address the problem by configuring the JET Runtime in one of three ways:

- selecting one of two JIT compilers available in the JET Runtime;
- enabling the caching of dynamically compiled code; and
- optimizing dynamically loaded classes on the developer's system, if possible

This chapter describes each technique in details and also points out certain Java technologies that definitely require Mixed Compilation Model.

11.2 Quick start

Let us consider an example that does not work without dynamic compilation. The Java SE platform includes Dynamic Proxy API, a mechanism that allows programs to dynamically create bytecode that represents classes and create new instances of those classes. A dynamic proxy class is a class that implements a list of interfaces *specified at run time*.

Obviously, Dynamic Proxy API cannot operate without dynamic compilation because bytecodes are created on-the-fly, literally as arrays of bytes, which are then fed to the virtual machine. You may get convinced of that it works flawlessly with the JET Runtime.

11.3 When the mixed model operates

11.3.1 When JET Runtime uses MCM

For certain types of application, the JET Runtime definitely employs the mixed model. That is the case, if some classes to be loaded are not known at the moment of static compilation. In general, there are two reasons for that:

- the classes appear later, i.e. when running the application
- the classes are loaded by custom classloaders that may define specific policies for lookup of class files

For example, dynamic compilation occurs, if your application features the following:

Feature	Used by
loading classes from remote host	RMI Distributed Object Model
generation of bytecode at run time	Dynamic Proxy API
generation of class files at run time	JasperReports
custom classloading and third-party plug-ins	NetBeans
dynamic deployment of components	Java application servers, such as JBoss

11.3.2 When you may use MCM

Even though certain classes of your application can be compiled statically, you may enable the mixed model for them explicitly. It is useful if your application contains a lot of similar classes but loads just a few. A good example is country-specific or language-specific classes in internationalized applications. Instead of precompiling all of them, you may supply those classes in a `.jar` file. You may optionally pack that `.jar` to the executable as described in [10.13](#).

However, it is not recommended to use MCM for "hot" classes that take a significant fraction in total execution time. It would be better to pre-compile such classes because the JIT compiler incurs run time overheads and employs less efficient optimizations as compared to the static compiler.

11.4 Using the mixed model

11.4.1 JIT compiler selection

The JET Runtime comes with two JIT compilers:

optimizing

This JIT compiler is essentially the main static compiler, scaled down and adapted for runtime use.

fast

This compiler implements a few low-cost optimizations, and thus is an order of magnitude faster than the optimizing JIT compiler.

Note: the Standard Edition of Excelsior JET includes fast JIT compiler only.

The differences between the two compilers are summarized in the following table:

Characteristic	Optimizing JIT	Fast JIT
Compilation speed	Low (order of magnitude slower than HotSpot)	High (comparable to HotSpot)
Code quality	High (somewhat below the static compiler)	Moderate
Memory requirements	High	Moderate

Note: If the optimizing JIT compiler fails to compile a method due to lack of memory or inability to restore the method's structure suitable for optimizations, which may be caused e.g. by obfuscation, it always invokes the fast JIT compiler for such a method.

You select either optimizing or fast JIT compiler on **Options** page (see 4.8.3) of the JET Control Panel. Another way to force using a particular JIT is setting the system property (see 10.5.1) `jet.jit.optimizing` or `jet.jit.fast`, respectively.

If you do not select the JIT compiler explicitly, the JET Runtime will use the optimizing JIT compiler.¹

If you want to have the shortest possible JIT compilation pauses, use the **Fast JIT** compiler, and specify the `jet.jit.disable.resolution` property at runtime (see 10.5.3). Note that setting this property may negatively affect JITed code performance.

¹The fast JIT may be enabled by default if you use JIT Cache Optimizer (see 11.6.3) or the `xjava` launcher (see 11.7).

11.4.2 Caching JIT'ed code

Note: information in this section is not applicable to the Standard Edition of Excelsior JET.

To reduce the overhead of dynamic compilation, the JET Runtime is capable of caching the code produced by the JIT compiler during execution. Once a JIT cache is created, it may be reused by the application on next launches so that JIT compiler will not be invoked for the cached classes. To provide correctness, the JET Runtime performs consistency checks to ensure that the bytecode being loaded has not been changed since the previous run. If the checks fail, JIT cache is (partially) rejected, that is, the outdated code is not used. In place of it, the JIT compiler produces an up-to-date version of the cache.

You enable the caching mechanism for your application on **Options** page (see 4.8.3) of the JET Control Panel or by setting the system property (see 10.5.1) `jet.jit.cache`.

Technically, JIT cache is organized as a directory that contains a set of shared libraries with the cached code and a *cache descriptor* file. The JET Runtime consults the cache descriptor when it performs consistency checks or looks for a piece of the cached code.

By default, the JIT cache directory has the name `jittemp` and is located in:

For a conventional executable:

the directory which is set current when the executable is launched (if applicable, the Working Directory property of the respective shortcut, otherwise the directory containing the executable file)

For a shared library accessed via the Invocation API:

the current directory of the calling process at the moment of the first call to `JNI_CreateJavaVM()`

You may assign a different location for the cache directory on **Options** page (see 4.8.3) of the JET Control Panel or by setting the system property (see 10.5.1) `jet.jit.cache.dir`, for instance:

```
-Djet.jit.cache.dir:/tmp/jitcache
```

Note: In general, using an absolute pathname for the cache directory is not recommended as it may cause troubles on deployment. If you want to rename the cache directory or change its location on end user systems, specify the `jet.jit.cache.dir` property when packaging your application for deployment (see 6.10.2 for details.)

11.5 Reflective shield

Many JVM implementations, e.g. Sun HotSpot, load classes on the first use (class instantiation, static method call or static field access), not on the first reference. This is called *lazy class loading*. Therefore, if a referenced class is not available at application launch

time but becomes available at run time before it is actually used by a method, the program executes normally.

By default, the JET static compiler resolves all references to classes at compile time so if you create a project that includes classes with unresolved import dependencies, JET Control Panel displays detailed warnings on the Class view dialog (see 4.7.5).

Sometimes class absence is just an inconsistency that may be easily fixed. For example, it may appear if you forget to add a `.jar` to the compilation set. However, if you have double checked your project and not found the missing classes in a jar file or directory, still there is a chance that the classes in question are made available at run time.

To cope with it, the compiler will generate special stubs at all places where an unavailable class is used. On the first execution, such a stub checks if the referenced class has already been loaded. If not, the stub attempts to load it using the JIT compiler and then uses reflection information (hence the term *reflective shield*) to resolve the field or method that was referenced in the code. On subsequent executions, the stub will determine that the resolution has already occurred and proceed to the actual use of the field or method.

If a shielded class has not become available at run time, the respective stubs will throw `java.lang.NoClassDefFoundError`.

You may also use this reflective shield facility to pick out the classes and packages that you do not want to precompile by simply making them unavailable to the JET static compiler. However, take into account that reflective stubs create overhead that may lead to considerable performance degradation if methods of the shielded classes are frequently executed.

Note: The JIT compiler is also capable of generating reflective stubs for imported classes that are not available yet.

11.6 JIT cache optimization

<p>Note: information in this section is not applicable to the Standard Edition of Excelsior JET.</p>

This section describes JIT Cache Optimization, an innovative Excelsior's technology that enables you to use the JET static compiler to optimize applications that rely on custom classloaders.

11.6.1 Challenges

Many Java applications use custom classloading to dynamically load classes at run-time. This technique provides unique name spaces for applications that support third-party plug-ins. Each plug-in is loaded by a separate custom classloader thus avoiding a possible class names conflict.

The JET Caching JIT compiler is capable of compiling and caching classes loaded by custom classloaders. The problem, however, is that the dynamic compiler is a scaled down version of the main, ahead-of-time JET compiler and, therefore, performs weaker

optimizations in exchange for substantially lower resource demands during on-the-fly compilation. Therefore, the quality of dynamically compiled code is lower.

If an application heavily uses custom classloaders, yet another problem arises. Classloading, as it is defined in the Java Language Specification, imposes strict rules on the order of class lookup and resolution of references to other classes. This makes ahead-of-time compilation of such application impossible, because in a general case not all references may be resolved statically. A simple counterexample is a custom classloader defining its own classpath at run-time. As a result, it may occur that only a small part of the application can be **statically** compiled even if the majority of its classes **is known** at compile-time. So the advantages of static compilation for applications of such kind are significantly reduced. The challenge was to work around this problem.

Finally, a JIT cache for a particular application may contain dozens of shared libraries created during multiple dynamic loading sessions.

11.6.2 The solution

The JIT Cache Optimization feature addresses all the above problems through “fusing” the contents of a JIT cache into a single shared library using the main JET ahead-of-time compiler.

11.6.3 How to optimize a JIT cache

JIT cache optimization is a three-step process:

1. Create a JIT cache by running your application in a special mode, when the classes processed by the JIT compiler are flushed to disk.
2. Run the main JET compiler against the classes collected on Step 1. The result is one fully optimized shared library and the new cache descriptor.
3. Replace the cache contents with the files created on Step 2.

The `samples/Classloaders/JITCacheOptimizer` subdirectory of your Excelsior JET installation contains some example programs.

Alternatively, you may use the `xjava` launcher (see 11.7) to run your entire application through the Caching JIT compiler and then optimize it.

STEP 1. Create a JIT cache with flushed classes

Suppose you have an application that uses the JET Caching JIT compiler and you set a cache directory, for example, `MyTemp`:

```
export JETVMPROP=-Djet.jit.cache -Djet.jit.cache.dir:MyTemp
./MyApp
```

First, set the `jet.jit.save.classes` property to flush all dynamically compiled bytecode to disk for subsequent recompilation. Class flushing has a positive

side effect: it allows you to pre-compile even those classes which do not exist on your system in the form of class files (e.g. if they are dynamically created at run time).

Note: You may use `jet.jit.save.classes` on the developer's machine only and disable it in the compiled application if you find that it may cause security violation. Turn **DISABLECLASSSAVING** option ON in your project file to prohibit class flushing on end users' systems.

After that, you run your application using the JET Caching JIT compiler to create a JIT cache. Along with compiled code, the dynamic compiler creates a cache descriptor in your JIT cache directory. Cache descriptor is a plain text files named

```
jitJET-version-code J2SE-version-code.cache
```

(for instance, `jit37023.cache`).

STEP 2. Fusing JIT cache

Create a separate project specifying the cache descriptor as a source:

```
----- fuseCache.prj -----
-lookup=*.cache=MyTemp
+gendll
-outputname=fusedCache
!uses path-to-application-project-file
% JETVER/VCODE hold JET/JRE version codes
!module jit$(jetver)$(vcode).cache
-----
```

and build it as

```
jc =p fuseCache.prj
```

As a result, you will get a new cache descriptor (e.g. `jit31523.cache`) created in the `MyTemp` directory and `fusedCache.so` created *in the current directory*.

STEP 3. Using the Newly Fused Cache

At this point, you may remove all files from `MyJITTemp` except the cache descriptor and copy the newly created `fusedCache.so` to that directory. Also, remove `jet.jit.save.classes` property setting.

Now, if you run your executable, it will automatically use the fused cache.

11.6.4 Incremental fusion

You may fuse JIT Cache incrementally. For that, follow the steps described in 11.6.3, but do not delete flushed classes and leave the `jet.jit.save.classes` setting intact on

Step 3. If your application creates new cache entries that you wish to fuse with the shared library created at previous steps, just perform the procedure again. This saves time of dynamic compilation because the cached code is reused and compilation occurs only for newly loaded classes. To save time of static recompilation of JIT cache, you may specify `=s` switch to enable the incremental build mode as

```
jc =p =s fuseCache.prj
```

When you decide to stop the process, remove `jet.jit.save.classes` setting and run your application without flushing classes.

11.7 xjava launcher

Note: information in this section is not applicable to the Standard Edition of Excelsior JET.

If the application you want to optimize with Excelsior JET takes full advantage of Java dynamic class loading facilities, e.g. it uses custom classloaders intensively or comprises third-party libraries that do so, creating a complete *and correct* project file becomes challenging. To make the optimization process more straightforward, Excelsior JET provides the `xjava` application launcher.

`xjava` is a command-line tool with a syntax and semantics very similar to the conventional `java/javaw` launchers included with the Sun JRE. `xjava` loads the JET Runtime and feeds your application's classes to the Caching JIT compiler. The resulting JIT cache can later be optimized into a single shared library (see 11.6) by invoking `xjava` again with the `-Xcompile` option added. This will also produce a custom launcher for your application that you will be able to distribute.

The disadvantage of using `xjava` is that only the classes that were actually loaded during your test runs will be optimized. Moreover, when a different classloader loads a class that was previously loaded, the class may have to be compiled again. In a general case, you will have to deploy all the original jar files together with the optimized JIT cache shared library, not to mention the JET runtime shared libraries and the custom launcher. This limits the usage of this feature to situations when the distribution size is not an issue, which is typical for server-side applications.

11.7.1 Synopsis

```
xjava [ options ] main-class arguments
xjava [ options ] -jar jar-file arguments
```

options

Command-line options of the launcher.

main-class

Name of the main class

jar-file

Name of the jar file to be invoked.

arguments

Arguments passed to the `main()` method, if any.

11.7.2 Options

For the sake of compatibility, the `xjava` launcher supports some standard and non-standard options of the Sun JRE launcher.

`-classpath` | `-cp` *classpath*

classpath is a list of directories, jars and ZIP archives to be searched for class files and resources, separated by colons (“:”). Overrides the `CLASSPATH` environment variable. If that variable is not set and the `-classpath` / `-cp` option is not used, the current directory (.) is assumed to be the only entry of the user class path.

`-Dproperty=value`

Set *value* of the given system *property*.

`-ea` [: [*package*] . . . | *class*]

Enable assertions. Assertions are disabled by default.

If *package* is specified, assertions are enabled in that package and all its subpackages. If only “. . .” is specified, the unnamed package is assumed. If *class* is specified, assertions are enabled in that class only.

`-da` [: [*package*] . . . | *class*]

Disable assertions. Assertions are disabled by default.

If *package* is specified, assertions are disabled in that package and all its subpackages. If only “. . .” is specified, the unnamed package is assumed. If *class* is specified, assertions are disabled in that class only.

`-esa`

Enable assertions in all system classes.

Note: This option is recognized but ignored, because `xjava` uses system classes precompiled by JET Setup (see Chapter 2). In the current version, assertions in system classes are always *disabled*.

`-dsa`

Disable assertions in all system classes.

Note: This option is recognized but ignored, because `xjava` uses system classes precompiled by JET Setup (see Chapter 2). In the current version, assertions in system classes are always *disabled*.

`-version`

Print product version to the standard error and exit.

- showversion
Print product version to the standard error and continue.
- ? | -help
Print a brief help message and exit
- X
Print help on non-standard options and exit.
- Xmxsize
Set maximum Java heap size. Append “K” or “M” to indicate that the value is in kilobytes or megabytes respectively.
- Xsssize
Set Java thread stack size.
- Xverify:all
Verify that all class files obey language constraints. Without this option, all classes are still verified, but less strictly — without the checks that are disabled in the Sun HotSpot VM for non-remote classes by default.

The following options are specific to `xjava`:

- Xenduser
Do not cache class files.
- Xcachedir:directory
Set the JIT cache directory. The default is the “jittemp” subdirectory of the current directory.
- Xcompile[:launcher-name]
Compile the JIT cache into a single shared library using the main JET compiler and generate a custom launcher *launcher-name*. See [11.7.3](#) for details.

If *launcher-name* is not specified, the launcher will be named after the main class specified on the `xjava` command line (or after the jar file if the `-jar` option is used.)
- Xrt:runtime
Select a specific JET run-time. Valid values of *runtime* are `classic`, `desktop`, `workstation`, and `server`. If *runtime* is not specified, `desktop` is assumed.

See [10.2](#) for details.
- Xuseprj:prj-file
Retrieve options for JIT cache compilation from *prj-file*. This option has no effect if `-Xcompile` is not specified.

Note: Only the options that affect code generation will be retrieved from *prj-file*.
- Xgui
This option is ignored on Linux.

11.7.3 JIT cache optimization with xjava

The `-Xcompile` option changes the behavior of `xjava` completely. Instead of running your application, it does the following:

1. recompiles the classes cached on previous runs into a single shared library, essentially automating Step 2 of the JIT cache fusion process (see 11.6.3) for you.
2. creates a *custom launcher* — a small executable that loads that shared library and runs your application.

Any command-line options other than `-Xcompile` are hard-wired into the custom launcher, so the correct usage is to *add* `-Xcompile` to the options you previously specified:

```
xjava -cp MyLib.jar -Dfoo=bar -Xmx256M foo.bar.Main
xjava -cp MyLib.jar -Dfoo=bar -Xmx256M -Xcompile foo.bar.Main
```

If you want to change the default settings for JIT cache compilation, use the `-Xuseprj:prj-file` option, where *prj-file* is a regular JET project file (see 16.8). `xjava` ignores *prj-file* directives that do not affect code generation.

11.7.4 Usage example

To begin using the `xjava` launcher, you normally just need to replace the “java” / “javaw” command with “xjava”. For instance, the following command:

```
xjava -cp MyLib.jar foo.bar.MainClass
```

will run your application with the JIT compiler caching files in the ‘`jittemp`’ subdirectory of the current working directory. Then, you can optimize that cache by adding the `-Xcompile` option:

```
xjava -Xcompile -cp MyLib.jar foo.bar.MainClass
```

The result will be the custom launcher `foo.bar.MainClass`, optimized cache shared library, and new cache descriptor. The `-cp MyLib.jar` option will be hardwired into the launcher. Now you may run the optimized application by typing just

```
foo.bar.MainClass
```

To deploy the optimized application, create a new package in JetPackII and add the launcher executable to it. JetPackII will recognize the launcher and prompt you for adding the optimized cache to the package. See 6.9.1 for details.

11.8 JIT compilation and code protection

Code protection is an important benefit of the Excelsior JET JVM. The JET Optimizer converts your classes to optimized native code, which is very hard to reverse engineer,

so understanding your algorithms becomes a headache even for the most experienced hackers.

However, Excelsior JET has some limitations with respect to code protection:

- Class files that will be JITed on end user systems cannot be protected.

When creating a JET project file for your application, make sure to set `protect` to `all classes` for all jar files that contain the proprietary code (refer to [4.7.2](#) for details).

- Executables created by the `xjava` utility do not provide code protection.

The `xjava` launcher is intended to help you optimize applications that heavily rely on custom classloaders. Even if you compiled such an application to native code, the current implementation of the JET Runtime requires the application class files to be available at run time². So, all jar files must be deployed along with the application binary.

Excelsior engineers are currently investigating how to enable code protection for this kind of applications. If this topic is of your interest, please contact us and describe your case.

²This limitation applies only to the applications generated by `xjava`.

Chapter 12

Dynamic linking

12.1 Intro

Operating system support for dynamic linking

In a traditional developer toolkit that includes a static compiler and linker, e.g. a C++ toolkit, static libraries are linked together with the application code and all references between them are resolved at link time. Contemporary operating systems provide support for dynamic link (or shared) libraries. Unlike static libraries, they are linked with executables at run time. If an executable contains direct references to methods and/or fields from a shared library, linking is performed by the dynamic linker when the executable starts. This is called *load time linking*. Some applications, however, postpone the linking process to a later phase so that the application's code loads shared libraries and resolves references using the system API. This is called *run time linking*. The executable and a set of loaded shared libraries form a so called *working set* of an operating system process.

Typically, working sets of different processes have common components, such as shared libraries implementing the system API functions. These components are *shared* between processes, that is, only one copy of code and constant data from a shared library is loaded into memory. This implies the following important benefits for native applications employing the dynamic link model:

- applications use less memory thanks to sharing
- application start-up is much faster if common components are already loaded by another process
- application can control its working set by loading certain shared libraries on demand
- multi-lingual programming is easier because shared libraries and the main executable may be implemented in different languages

What about Java applications?

The question is whether you the Java developer can benefit from dynamic linking supported by the target operating systems? If you are reading this manual, the answer is YES because you most probably have a copy of Excelsior JET JVM installed on your system.

First of all, the Java 2 platform classes are pre-compiled into a set of shared libraries when you create a JET profile (see 2.3). Therefore, they will be shared between the processes established by JET-compiled executables, just like the shared libraries providing the OS API are shared between conventional native applications. This effectively means that you already have a JVM capable of sharing constant data (such as reflection information) and *code* (e.g. methods of Java 2 platform classes) between several running Java applications.

As for your application's code and third-party Java APIs not included in the Java 2 platform, you have two possibilities of using dynamic libraries:

- Create *Invocation shared libraries* callable from a foreign environment, e.g. from C/C++
- Build *multi-component applications* consisting of several executables and one or more common shared libraries

The next Sections describe both techniques in details.

12.2 Invocation shared libraries

To create a shared library callable from applications written in another language, you can use the JET Control Panel: click **Invocation DLL** on the Welcome Screen (see 4.2) and select the desired jars on the **Start** page (see 4.5).

Using Invocation shared libraries is a bit tricky. Like any other JVM, Excelsior JET executes Java code in a special isolated context to correctly support exception handling, garbage collection, and so on. That is why Java methods cannot be directly invoked from a foreign environment. Instead, you have to use the standard Java 2 platform APIs, specifically the Invocation API and Java Native Interface (JNI). See `samples/Invocation` in your JET installation directory for detailed examples.

12.3 Multi-component applications

Note: Before you begin, learn about *multi-app executables* (see 10.7) to decide if you really need to create a multi-component application. For example, if you plan to compile two executables sharing common libraries and both executables are always deployed together, it would be much simpler to build a single multi-app executable rather than two executables and shared library.

Currently, Excelsior JET Control Panel does not support creating multi-component applications so you have to use the command line interface to the JET compiler. However, the

Control Panel can assist you at the first steps of creating a multi-component application. Please, **read this entire section carefully** so as to correctly setup project files for shared libraries/executables and avoid problems with compilation and running multi-component applications.

12.3.1 Building multi-component applications

This section contains step-by-step instructions for creating multi-component applications. Please, note that all the steps described below are obligatory.

It is supposed that you have a set of jar files containing all class files and *resources* necessary to your application. If your application includes some standalone classes/resources residing in a directory, you have to create a jar file from this directory's contents. This is important because the JET compiler can automatically pack all resources from jar files to the resulting shared libraries and executables. Moreover, it is much easier to write a JET project file if all classes and resources are packed to jars.

STEP 1. Consistency check

First of all, create a project to compile all the jars into a single executable using the JET Control Panel, then proceed to the Step 3 (see 4.7).

If there are consistency warnings, click the **Class View** buttons and inspect the **Warnings** tab below. If it says there are *duplicated classes*, fix the project by removing duplicated classes from your jar files, and repeat the project creation process until there are no more class duplications in the compilation set.

If warnings about unresolved import dependencies are displayed, you can ignore them at this step, provided the application works under the HotSpot JVM flawlessly.

Finally, go through the remaining pages to set up the project, compile it and make sure that your application works properly.

Note: The created project file **cannot** be used for subsequent compilation of shared libraries and executables.

STEP 2. Defining multiple compilation sets

On this step, you define the contents of shared libraries and executable(s) being created. Specifically, you break down the compilation set, checked for consistency as decried above, into several compilation sets, each to be compiled into a separate component (executable or shared library). The breakdown is **not arbitrary** as it should obey the following rules:

- A jar may not belong to more than one compilation set.
- Cyclic import between compilation sets is not allowed. For example, if `A.jar` uses `B.jar` and, at the same time, `B.jar` uses `A.jar`, they should be put into the same compilation set.

So be ready that the model "one jar — one shared library" cannot be used if the jars have cyclic interdependencies.

The JET Control Panel can assist you in defining multiple compilation sets as follows:

1. Open the JET Control Panel and click **Invocation library** on the Welcome Screen.
2. On the page **Start**, check the "**Enable Manual setting**" box and four new buttons will appear on the screen. Using the upper button, add one or more jars to the compilation set.
3. Go to the page **Classpath**.
4. Check that NO warnings about unresolved import dependencies are displayed except those persisted since the **Consistency Check** step.
5. If there are such warnings, add the respective jars until all imported classes are resolved.
6. At this point, you have a compilation set (let us denote it CS_1) consisting of one or more jars that will be compiled into the first shared library.
7. Go to the **New** page and add few more jars. Again, check that no unresolved import dependencies are displayed. The files you added constitute the compilation set for the second shared library, let us name it CS_2 .
8. Repeat the process for the remaining jars, thus defining compilation sets $CS_3, \dots, CS_k, CS_{exe}$ for other components.

The compilation set CS_{exe} , including the jar with the main class of your application has to be defined last as it will be compiled to the executable. The other compilation sets will be compiled to shared libraries.

Actually, you can define several compilation sets that reference CS_1, \dots, CS_k to create several executables with common shared libraries.

Let us consider a typical example of multiple compilation sets. Suppose you have two applications using a Java Optional Package, such as JavaHelp. You may define three compilation sets: the first one contains the jars of JavaHelp and the others — jar files of your applications. This breakdown implies that you can create a shared library for the JavaHelp API and two executables using that shared library.

STEP 3. Creating the build environment¹

Create an empty directory called, for instance, `MultiCompBuild`. In the directory, create JET projects (plain text files with the extension `.prj`) - one project per each component. For all but the last compilation sets defined on STEP 2, create projects named, say, `CS_1.prj, . . . , CS_k.prj` after the following template:

¹Example project files cited in this Section can be found in `samples/MultiComponent` in your JET installation directory.

```

-OUTPUTNAME=shared-library-name      (without extension)
-GENDLL+
-CLASSABSENCE=HANDLE
-IGNOREMEMBERABSENCE+

-LOOKUP=*.jar=path-to-jars

!uses path-to-project-file-1
!uses path-to-project-file-2

%jar files from this compilation set
!CLASSPATHENTRY jar-1
    -PACK=NONCOMPILED
    -OPTIMIZE=ALL
    -PROTECT=ALL
!END

```

You have to edit only the `-OUTPUTNAME=`, `-LOOKUP=*.jar=`, `!CLASSPATHENTRY` and `!uses` settings. Note that you have to add `!uses` directives (see 16.8.5) only to the projects that depend on other projects.

For the compilation set corresponding to the main executable, create a project file called, for instance, `CS_EXE.prj`, after the following template:

```

-OUTPUTNAME=executable-name      (without extension)
-CLASSABSENCE=HANDLE
-IGNOREMEMBERABSENCE+

-LOOKUP=*.jar=path-to-jars
-MAIN=main-class

!uses CS_1.prj
!uses CS_2.prj
    . . .
!uses CS_k.prj

%jar files from this compilation set
!CLASSPATHENTRY jar-1-for-exe
    -PACK=NONCOMPILED
    -OPTIMIZE=ALL
    -PROTECT=ALL
!END

!CLASSPATHENTRY jar-2-for-exe
    -PACK=NONCOMPILED
    -OPTIMIZE=ALL
    -PROTECT=ALL
!END
    . . .

```

In this template, you have to edit the `-OUTPUTNAME=`, `-LOOKUP=* .jar=`, `-MAIN=` and `!CLASSPATHENTRY` settings.

STEP 4. Compilation

Set `MultiCompBuild` as the current directory and compile these projects using the commands

```
jc =p CS_1.prj
...
jc =p CS_k.prj

jc =p CS_EXE.prj
```

Now you can run the resulting executable.

Note that this particular order of compilation is important to transparently resolve references between the components being compiled. This is exactly the order in which the compilation sets were defined on STEP 2.

STEP 5. Re-compilation

If you change the jars from `CSexe` compilation set, it is enough to recompile only the executable with the command

```
jc =p CS_EXE.prj
```

However, if jars compiled into one of the shared libraries were changed, it is necessary to recompile that shared library as well as the components that were built after it, including the executable. Finally, if modifications of shared library's jars change the import relationships (e.g. new classes are referenced), you may have to repeat the whole process starting from STEP 1. Fortunately, that is a rare case if you create shared libraries from third-party jars which do not change.

12.3.2 Using multiple components

As it was said above, shared libraries can be mapped to the process via load time and run time linking. This Section describes how JET-compiled applications employ both ways of linking.

Load time linking

If one component directly uses classes from another component, this *always* implies load time linking². Examples of direct uses are creating a class instance via the `new` operator, extending a class from another component, accessing static members, etc. However,

²!uses directive (see 16.8.5) also forces load-time linking for the imported component.

if one component employs *only* the Reflection API to use classes from another components, load time linking does not occur because there are no direct references between the components. A typical example is loading classes via the `Class.forName()` method and passing the obtained class objects to the methods from the `java.lang.reflect` package.

Note that `Class.forName()` and the Reflection API also work for all classes from the components linked to the process at load time.

Run time linking

JET-compiled applications can load shared libraries on demand in a portable way, that is, without using the operating system API. The `Class.forName()` method serves for this purpose so that looking for a class from a pre-compiled but not yet loaded shared library results in loading that shared library at run time.

Suppose component B has to load component A at run time. To enable run-time linking, two pre-conditions should be met:

1. B does not reference A directly as described in the previous Section
2. *Class-to-shared library* mapping for A should be specified when compiling B

The mapping allows the `Class.forName()` method to succeed for classes precompiled into shared libraries loaded at run time. You need to tell the JET runtime in which shared library to look for each class or package. You may do that by adding the following options to the **JETVMPROP** equation or the `JETVMPROP` environment variable:

```
-dll : class-name : dll-name      or
-dll : class-prefix* : dll-name
```

where

class-name

is a fully qualified name of a class, such as `com.MyCompany.MyClass`

class-prefix

is a class name prefix followed by the “*” character:

```
-dll : com.MyCompany.* : MySharedLib.so
```

This syntax is useful when you compile an entire package into a shared library.

dll-name

is the name of the shared library containing the class *class-name* or classes whose fully qualified names begin with *class-prefix*. If *dll-name* contains spaces, enclose it in double quotes:

```
-dll : com.MyCompany.* : "/usr/local/My App/MySO.so"
```

Note that if you do not specify a pathname, the shared library will be sought along the `PATH`, but that may slow down application execution. If you specify an incomplete pathname, such as “`My App/MySO.so`”, that pathname will be appended to

the name of the current directory and then to the name of each directory along the PATH.

You can specify more than one `-dll` option. If the name of the class being loaded matches more than one *class-prefix*, the class will be sought sequentially until it is found or there are no more matching prefixes.

Examples:

Project file:

```
% MYAPP.PRJ
-JETVMPROP=-dll:com.My.*:./private.so -dll:com.*:common.so
```

Shell script: #RUNMYAPP.SH

```
JETVMPROP="-dll:com.MyCompany.*:./plugins/myplugin.so"
export JETVMPROP
./MyApp
```

Run time compilation

If some classes cannot be pre-compiled, for example, those of third-party plug-ins to your application, the JET Runtime enables Mixed Compilation Model (see Chapter 11).

In this case, the bytecodes for the classes are sought in accordance with the conventional search algorithm used by the Java 2 platform. Then, the bytecodes are loaded and compiled at run time.

Chapter 13

Global Optimizer

Note: information in this Chapter is not applicable to the Standard Edition of Excelsior JET.

13.1 Introduction

Excelsior JET features the Global Optimizer — a powerful facility that has several important advantages over the default compilation mode:

- **single component linking** makes an executable that does not require JET Runtime shared libraries, thus reducing the download size of the installation package and disk footprint
- **global optimizations** improve performance and reduce startup time and memory usage of the compiled application

The Global Optimizer is integrated with the core JET compiler. By default, JET compiles applications in the dynamic link model, i.e. compiled application classes are linked into an executable that requires JET Runtime shared libraries containing precompiled Java SE platform classes (see 2.3). Quite the contrary, the Global Optimizer detects the platform classes that are actually used by the application and compiles them along with the application's classes into a single executable. Despite the resulting executable occupies more disk space as compared with the default mode, it does NOT require the JET Runtime shared libraries¹. As a result, the download size of the installation package is reduced.

The following table compares the Global Optimizer with the default compilation mode:

	executable size	download size	compilation time	disk footprint
Global Optimizer	larger	smaller	longer	smaller
Default mode	smaller	larger	shorter	larger

¹ only a few native shared libraries containing JIT compiler implementation are used

13.2 Application domain

Though the Global Optimizer provides major improvements mentioned above, it also has certain disadvantages as compared with the default mode.

First of all, you cannot use it to build multi-component applications (see [12.3](#)). The Global Optimizer always produce a single executable for the whole application, so you cannot split it into executable and shared libraries.

The other disadvantages are:

- compilation may take more time and memory
- update installation packages (see [6.8.2](#)) have a larger size
- including several executables into a single installation package may significantly increase the package size

In the default mode, the code of all platform classes is contained in the JET Runtime dynamic libraries shared between all executables in the original installation and update packages. Quite the contrary, every executable generated with the Global Optimizer contains its own copy of precompiled platform classes. So, if an installation package includes several executables or an update package contains a new version of the executable, the total download size increases due to code duplication. However, this is not a strict rule. For example, an installation package with a couple of small console applications compiled with the Global Optimizer may still have smaller download size as compared with the default mode.

To recap, when using the Global Optimizer, pay close attention to the following issues:

- size of the installation package if it includes several compiled executables
- size of update packages
- compilation time

Fortunately, switching between the Global Optimizer and the default compilation mode is very easy so you can try both to check which mode suits best for your case.

13.3 How it works

13.3.1 Global optimizations

Compiling both application and platform classes together, the Global Optimizer analyzes the entire program, thus enabling the compiler to optimize it more effectively.

13.3.2 Code mixing

When compiling an application, the Global Optimizer performs a whole-program type analysis, making a safe approximation of actually used classes and methods. Unused platform classes are left in the bytecode form (`.class` files), which significantly reduces the download size of optimized Java applications. For example, if you develop a console or SWT² application, the classes from the `java.awt` and the `javax.swing` packages are not compiled. This allows JetPackII (see 6) to include those classes into the installation package in a highly compressed form³. As a result, the created package has smaller size as compared with the default mode in which all classes are included into the package in the precompiled form.

13.3.3 Dynamic compilation

Under circumstances, the set of actually used classes may differ on the developer's and target systems. For example, it may depend on a flavour of the operating system, screen resolution, default system locale, etc.

However, this does not mean that the compiled application fails in such case. If the application loads a non-compiled class, it is compiled on-the-fly by the JIT compiler included in the JET Runtime. Though dynamic compilation imposes some runtime overheads, the overall application performance is almost not affected, because only a few extra classes are typically loaded⁴.

13.3.4 Code protection

Of course, distributing a class in the bytecode form does not provide code protection for that particular class. It is not a problem, however. Typically, the classes that need code protection are developed in your company (*application classes*), while the platform classes and the classes of third-party APIs (*API classes*) do not require protection and can be distributed in the bytecode form.

When you create a project in the JET Control Panel, it automatically separates the classpath in two groups: entries containing application classes and entries containing third-party API classes. If you select the SMART optimization preset on the Classpath page (see 4.7), the Global Optimizer is turned on and the Control Panel automatically enables protection for application classes and disables it for API classes. However, you can correct the default settings using the classpath grid (see 4.7.2).

Note: the more classes for which protection is disabled, the smaller size of the installation package you get.

²SWT is a graphic toolkit developed by IBM as a light-weight alternative to JFC/Swing. It is used, for example, by the Eclipse Platform

³If you opt for Excelsior Installer setup on the Backend page (see 6.13), JetPackII compresses the non-compiled classes using the pack200 method specially designed for Java class files

⁴provided that a thorough test run (see 13.4.2) is performed

13.4 Using the Global Optimizer

Building an executable with the Global Optimizer includes the following steps:

1. create a project for your application, in the same way as for the default compilation mode
2. perform a test run (see [13.4.2](#))
3. enable the Global Optimizer (see [13.4.2](#)) for your project
4. compile the project

The main difference from the default mode is that you are strongly encouraged to perform the test run step (see [4.6](#)) that is otherwise optional.

13.4.1 Why Test Run is necessary

When using the Global Optimizer, the Test Run step is obligatory. Due to the dynamic features of Java, the information about application execution, such as the set of classes loaded at run time, cannot be collected using only a static analysis. During a test run, the necessary information is collected dynamically and saved to `.usg` file which is then used by the Global Optimizer.

For example, consider a typical application that uses plugins and loads them via the Java Reflection API. The information about what plugins are actually loaded is known only at run time, because it may depend on user actions, configuration files, etc. Moreover, the set of loaded plugins can vary between different runs. Without a thorough test run, the Global Optimizer has no actual information about what plugins are executed and thus the quality of optimizations applied to them is reduced. As a result, performance and response time of those plugins will be low.

Note: Even if your application's code does not use plugins, Java Reflection API and JNI, you still have to perform a test run, because these mechanisms are used by the Java platform classes. For instance, Reflection is employed during Swing initialization, so most Java GUI applications implicitly use it.

13.4.2 Performing a Test Run

You can perform a test run in one of three ways:

- from the JET Control Panel (see [4.6](#))
- running the compiled executable in a special test mode
- using the `xjava` utility (see [11.7](#))

The recommended way is to perform a test run from within the JET Control Panel. All you need to do is just follow the instructions on Step 2 (see 4.6).

The information below is intended for users who, for some reasons, wishes to perform a test run manually. For example, it is useful if you want to stress test the application with the different input data or add the test run step to nightly build.

If you had already compiled your application into an executable, you enable the test mode by setting the system property (see 10.5.1) `jet.usage.list=MyApp.usg`.

Otherwise, you can run `xjava` utility (see 11.7) with this property specified on command line, for example

```
xjava -Djet.usage.list=MyApp.usg -Xnocache -cp MyApp.jar MyMain
```

If the `jet.usage.list` property is set, execution of your application *is* qualified as a test run. That is, upon application exit, the collected information will be written into the specified `.usg` file created in the application's working directory. In the examples above, the information will be saved to `MyApp.usg`.

You may re-run your application as many times as you wish, supplying, for instance, different input data. In this case, the previously created `.usg` file is not overwritten, but new information is added to it.

Notes:

1. When performing a test run for a JET-compiled shared library accessed by a native executable via the Invocation API (see 12.2), the `.usg` file is written when `DestroyJavaVM()` is called. If the process that loaded the shared library terminates without calling `DestroyJavaVM()`, the gathered information is *not* saved.

During a test run, stress test your application to achieve the largest possible code coverage. Thorough testing the application helps Excelsior JET collect more information about its execution and better optimize it.

The more detailed you test the application during a test run, the more information is collected and thus the better optimized application you get in result.

Note: a `.usg` file is generated automatically so it should not be edited by hand.

13.4.3 Enabling the Global Optimizer

In JET Control Panel, you turn on the Global Optimizer by selecting the "Smart" optimization preset (see 4.7.1). As a result, the Global Optimizer is enabled and properties of all classpath entries in the classpath grid (see 4.7.2) are set to the values recommended for this mode.

If you are editing your project file manually, add the following to the project file:

```
-global+  
!module MyApp.usg
```

13.5 Deployment

You can add the executables optimized with the Global Optimizer to the installation package in the same way as executables compiled in the default mode. Namely, on Step 2 of the JetPackII tool (see Chapter 6), add the executable to package files and all necessary the JET Runtime files will be added automatically.

However, it is not recommended to mix such executables with those compiled in the default mode in one package, because it significantly increases the size of the resulting package.

Moreover, it is recommended to use the Global Optimizer only if you have exactly one JET-compiled executable in the package. Otherwise the size of the package may increase because some platform classes are included into each executable.

Note: If you need to distribute more than one application in the package, consider using the multi-app executables (see 10.7).

Note: By technical reasons, it is impossible to include runtime utilities (see 6.11) to a package that contain executables compiled with the Global Optimizer. This limitation will be removed in future versions of Excelsior JET.

Chapter 14

Startup time optimization

Note: information in this Chapter is not applicable to the Standard Edition of Excelsior JET.

14.1 Introduction

The memory paging technique, used by contemporary operating systems, creates a considerable overhead on application startup time, because:

- the executable file is not read from the disk sequentially, but in a scattered order dependent on the execution path¹
- the entire memory page² is loaded even if the application only needs a few bytes from that page

Typically, this appears in the following scenario: on the first launch after system reboot (*cold start*), an application starts much longer than on any subsequent launch (*warm start*) when its files are cached in memory by the operating system.

Therefore, it is desirable to optimize application executables so as to minimize the number of pages to be loaded on startup and to ensure that the pages are accessed in an order close to sequential. This can be done with the *Startup Optimization Toolkit* included with Excelsior JET, which can reduce:

- cold startup time of Java applications by a factor of 1.5 – 2x
- application memory usage by 5–10%

14.1.1 Startup Optimization Toolkit

The Startup Optimization Toolkit comprises:

¹This is less of an issue for Solid State Drives.

²At least, 4KB on Intel x86

- the *Startup Optimizer* that works at compile time; and
- the *Startup Accelerator* that works at application launch

Both tools are profile-guided, that is, the application should be run in a profiling mode to collect the run-time information which is then used for the optimization. The good news are that you have to do nothing special to enable the startup optimization because these tools are integrated into the JET Control Panel.

14.1.2 Application considerations

In order to get maximum effect from the startup optimization, follow these recommendations:

1. Use *both* the Startup Optimizer and the Startup Accelerator as they complement each other.
2. For small- and medium-size applications³, enable the Global Optimizer (see Chapter 13) to take the standard Java SE packages used by the application in the optimization scope.
3. If your software product contains several applications sharing common Java libraries, build the applications together into a multi-app executable (see 10.7).

Known limitation: If you create a multi-component application (see 12.3), that is, compile the application's jar files into the executable and dynamic libraries, the effect of using the Startup Optimization Toolkit may be insignificant because the overheads of loading the libraries are not optimized.

14.2 Enabling startup optimizations

14.2.1 How to turn the Startup Optimizer on

Just run your application once on the page **Test Run** (see 4.6) of the JET Control Panel before compilation and then build the executable as usual.

Here is what happens behind the scenes when you do that:

1. During the Test Run, a startup profile of the application is collected and saved into a file. The file is created in the application's working directory and has the `.startup` extension
2. This file is automatically added to the project
3. The compiler reads the profile data from that file and performs the necessary optimizations

³Say, applications whose classpath includes 15,000 classes at most.

Though it requires you to run your application to gather the profile data, you do *not* need to repeat this step each time you modify your application. In most cases, the profile created for the Startup Optimizer is quite stable. You may wish to recollect it before making the final build for pre-release QA or integrate the profiling into the established automated build as described in section [14.3.1](#).

14.2.2 How to turn the Startup Accelerator on

New in JET 7.2:

In the JET Control Panel, the Startup Accelerator is enabled by default. Immediately after build, the Control Panel automatically runs the application just compiled and collects the profile data. You may only need to configure the default settings such as profiling timeout as described in section [4.11.1](#).

Under the covers, it works as follows:

1. The compiled application is automatically run for profiling
2. Profile data is collected and the application is automatically terminated on the specified timeout
3. The collected profile is then *hard-wired* into the application executable
4. Upon application startup, the JET Runtime uses the profile data to reduce the startup time

Note: Any recompilation of the application not followed by the profiling run of the resulting executable *disables* the optimization. To address this problem, the Startup Accelerator is also integrated with the command line JET compiler which is able to automatically run the executable in the profiling mode just after building a project. This and other options of integration of the Startup Accelerator in automated builds are covered in details in [14.3.2](#).

14.3 Automated builds

The next sections describe how to integrate the startup optimization into automated builds.

14.3.1 Startup Optimizer in automated builds

You may automate the collection of startup profile to include this step in your established build process. For that, you run your Java application with the `xjava` utility (see [11.7](#)) specifying the name of the resulting profile file in the `jet.jit.profile.startup` property on command line, for example

```
xjava -Djet.jit.profile.startup=Foo.startup -Xnocache -jar Foo.jar
```

Upon application exit, the collected information will be written into the specified file created in the application's working directory. In the example above, the information will be saved to `Foo.startup`.

Note: If a profile file with the same name already exists, it will not be overwritten, but new information will be added to it.

The profile filename is specified in the project via the **STARTUPPROFILE** equation (see 16.9.4). Provided you performed a Test Run at least once, the JET Control Panel adds this equation to the project. When automating the collection of the startup profile, make sure that the profile filename written in the project matches what you have specified on the `xjava` command line.

14.3.2 Startup Accelerator in automated builds

If the application can be run on the same machine where it is compiled, the integration of the Startup Accelerator into the build is very simple:

- Using the JET Control Panel: On the **Finish** page (see 4.11.1), check the box **Enable profiling in automated builds**.
- In the project file: set the value of **SAPROFMODE** equation to `ALWAYS` (see 16.9.4)

As a result, the JET compiler will automatically run the application in the profiling mode right after building the project and hard-wire the profile data into the resulting executable. Note that the duration of the profiling session is defined by the timeout you specify as follows:

- Using the JET Control Panel: On the **Finish** page (see 4.11.1), type the value in the **Profiling timeout** field.
- In the project file: set the value of **SAPROFTIMEOUT** equation (see 16.9.4)

Make sure that the timeout is long enough to capture the application startup.

XSTARTUP utility

If, for some reasons, you want to profile the application independently of its compilation, you use the **xstartup** utility specifying the application executable, the working directory and the profiling timeout. For example, this command

```
xstartup --work-dir=foo bar/MyApp
```

launches `bar/MyApp` from directory `foo`, profiles the application startup and hard-wires the collected data into executable `MyApp`.

Note: You may run the **xstartup** utility after deploying the application to another machine. However, if you use the Excelsior Installer to power the installation package created with JetPackII, there is no intermediate step when **xstartup** could work. As the

similar problem arises when using license managers, which also need to modify the compiled executable, you may use the packaging technique described in section 6.18.

Synopsis

```
xstartup [ options ] executable-name application-arguments
```

options

Command-line options of `xstartup`.

executable-name

Pathname of the executable file to be profiled. The collected data will be automatically hard-wired into it.

application-arguments

Command-line arguments of the application executable, if any.

Options

`-t` *n*

Sets the profiling timeout to *n* seconds. The application will be automatically terminated on this timeout. The default value is 20 seconds.

`--work-dir=`*directory-pathname*

Specifies the working directory for the running application. This switch is mandatory and must be set for any `xstartup` command.

In most cases, it is enough to set “.” as *directory-pathname*. But if the behavior of your application depends on the directory from which it is launched, specify the directory’s pathname.

Chapter 15

Troubleshooting

15.1 Compilation

When compiling my project, the JET Optimizer crashes

Though we put a lot of efforts in testing, the last bug in Excelsior JET is yet to be found. If you encountered a bug in the JET Optimizer, or if it crashed while compiling your application, please do the following:

- Upgrade to the latest version of Excelsior JET. There is a chance that the problem was fixed after your current version was released.
- Install the latest maintenance pack. We routinely issue MPs to all Excelsior JET versions currently supported, fixing major issues experienced by our customers.
- Contact Excelsior support. If possible, send us a stripped down version of your application, so we can reproduce the problem on our end. Information about your hardware and software environment, as well as the version and edition of Excelsior JET in use will also help.

When compiling my project, the JET Optimizer issues an "out of memory" message and then terminates

Reason: The current setting of memory usage limit in the JET's compiler is too small for compilation of your program.

Resolution: Increase the value of **COMPILERHEAP** in in your project file, for instance:

```
-compilerheap=90000000
```

When compiling my project, intensive memory paging occurs

Reason: The current setting of compiler memory usage limit is too large for your system.

Resolution: Close all applications that you do not need at the moment. If it does not help, reduce the size of compiler heap through decreasing the value of **COMPILERHEAP** in your project file or on the command line, for instance:

```
jc MyApp.jar -compilerheap=80000000
```

The default value of **COMPILERHEAP** may be inspected by running `jc` in the EQUATIONS submode:

```
jc =e
```

15.2 Execution

Diagnostics of unrecoverable errors: TRAP#N

Unrecoverable errors are those preventing the program from further execution, that is, the errors (crashes) which cannot be handled.

A known example is access violation that occurs when reading/writing memory at an address to which no memory page was mapped or transferring control to such a region. If a JET-compiled application runs into the problem, the Runtime prints TRAP#3 message and execution abnormally terminates.

Of course, it should never happen in a pure Java application but, unfortunately, it may happen due to:

- bugs in the native method libraries used by the application
- JNI misuses (correct usage of JNI is tricky for inexperienced programmers)
- operating system malfunction (yes, we found a few examples when debugging customers' applications)
- bugs in the JET Optimizer or Runtime escaped Excelsior QA (despite it's a rare case for the matured versions released last years, it may happen.)

Often, the errors are volatile, that is, hardly reproducible. To address the problem, the JET Runtime includes a special diagnostic facility. If an unrecoverable error occurs, the Runtime collects various information about the application state, logs it into a file, and prints the error message referencing that file.

HOWTO

How do I set Java properties at the very startup of my application?

Properties may be set at compile time using the **JETVMPROP** equation or at launch time using the **JETVMPROP** environment variable. See [10.5](#) for details.

The compiled application cannot find resources (images, etc.)

Resolution: Use resource packing (see [10.13](#)).

OutOfMemoryError is thrown during execution

Reason: Application's heap size is set to adaptive, and there is not enough available physical memory on the system, or the explicit setting is too low.

Resolution: Adjust the application's heap size setting:

- Using the Control Panel: go to the **Options** page and set the **Heap Size** field to the desired value (see [4.8.2](#).)
- At launch time: set the `jet.gc.heaplimit` property through the environment variable `JETVMPROP` in a batch file, e.g.:

```
SET JETVMPROP=-Djet.gc.heaplimit:90m
MyApp
```

More information on memory management settings of the JET Runtime may be found in [10.6](#).

Reason: Your application runs too many threads.

Resolution: Reduce thread stack size:

- Using the Control Panel: go to the **Options** page and decrease the value of **Stack size** (see [4.8.4](#).)
- Using the command line compiler: decrease the value of **STACKLIMIT** in `jc.cfg`, in your project file, or on the command line, for instance:

```
jc MyApp.jar -stacklimit=100000
```

The default value of **STACKLIMIT** may be inspected by running `jc` in the EQUATIONS submode:

```
jc =e
```

15.3 Deployment

My application does not work on another system

See Chapter [6](#) for information on deployment of the compiled applications.

How do I build my application as a single executable file?

Excelsior JET is not able to create a single all-inclusive executable that can be deployed to target machines by simply copying. Instead, JET offers the following deployment options:

- Preparing a self-contained directory that includes the executable(s) and all necessary supplemental files. Such a directory can be deployed to other computers by simply copying. No extra settings are required to run the included executables.
- Creating a self-extracting setup with command-line or GUI interface.

See Chapter 6 for information on JET deployment automation facilities.

If you are interested in creating single executables for reasons other than deployment, please fill out the form at

<http://www.excelsior-usa.com/jetfr.html>

and describe your requirements for that feature.

Chapter 16

JC reference

16.1 Configuring the compiler

16.1.1 System search paths

For your operating system to know where to find the executable binary files which constitute the JET package, the operating system search paths must be set appropriately. This is normally done by the JET installation program. See also the *Read Me First* file from your on-line documentation.

16.1.2 Portable filename notation

Keeping in mind future JET ports to other platforms, we introduced a portable notation for file names that may be used in all compiler control files and on the command line. The portable notation combines DOS-like and Unix-like notations (note that on Unix file names are case sensitive):

```
[ drive_letter ":" ] unix_file_name
```

Examples

```
c:/jet/bin  
/mnt/users/Alex/cur_pro  
MyProject/src/main.class
```

Along with the *base directory* macro "\$!" (See 16.7), this portable notation allows you to write control files in a platform independent and location independent manner.

16.1.3 Working configuration

The core part of JET is the `jc` utility, which combines the compiler with the project subsystem, accompanied with a set of system files¹:

`jc.red` Search path redirection file (see 16.1.5)
`jc.cfg` Configuration file (see 16.1.7)
`jc.msg` Texts of error messages (see A.1)

Upon invocation, `jc` attempts to locate the `jc.red` file in the current directory. If there is no such file, it loads the *master redirection file* from the directory where `jc` executable resides.

Other system files are then sought by paths defined in the redirection file. If `jc.red` was not found, or if it does not contain paths for a particular system file, that file is sought in the current directory. If the file is not found, the *master file* is loaded from the directory where the `jc` utility resides.

We recommend you not to edit the master redirection file. Instead, use **LOOKUP** statements in your project files to specify directories for file search and creation. However, if you compile several projects in a single directory, you may wish to create a local `jc.red` file in that directory.

A configuration file contains default compiler settings for several projects. Project specific settings are defined in project files (See 16.8). A so-called template file is used to automate the program build process (See A.4).

A redirection file, a configuration file, and, optionally, a project file (see 16.8) constitute the working environment for the given invocation of the `jc` utility. The compiler preprocesses all those files as described in 16.7.

16.1.4 Directory hierarchies

JET gives you complete freedom over where to store your bytecode files and files the compiler creates for you. We recommend you to work in a project oriented manner — i.e. to have a separate directory hierarchy for each independent project.

In JET implementation a Java package corresponds to a directory. It imposes the following (quite natural) restriction: since disk location of a package is defined as a directory, location of a subpackage is implicitly defined as a subdirectory of that directory. The default package corresponds to the current directory, but its disk location may be redefined with the help of *redirection* (See the 16.1.5 subsection and description of the **LOOKUP** equation).

16.1.5 Redirection file

A redirection file defines directories and archives in which JET looks for files and creates files. It consists of several redirections of the form²:

¹A name of a system file is constructed from the name of the compiler utility and the correspondent filename extension. If you rename the `jc` utility, you should also rename all system files accordingly.

²See also 16.7

```
Redirection = pattern "=" Location { ";" Location }
Location    = ( directory | archive )
```

pattern is a regular expression with which names of files the compiler has to open or create are compared. A pattern usually contains wildcard symbols "*" and "?", where

Symbol	Matches
*	any (possibly empty) string
?	any single character.

For a full description of regular expressions see [A.3](#).

It is also possible to have comment lines in a redirection file. A comment line should start with the "%" symbol.

directory or archive name may be specified using the portable notation (see [16.1.2](#)).

You may specify absolute pathnames, such as

```
/usr/myproj/class
```

or names relative to the current directory, such as

```
src/common
```

denoting the directory `src/common` which is a subdirectory of the current directory. A single dot in a pathname represents the current directory, a double dot represents the parent, i.e. the directory which has the current directory as a subdirectory.

When looking for a file, `jc` sequentially matches its name against each `pattern`. If a match was found, the file is sought in the first of the directories listed on that line, then in the second directory, and so on until either the file is found, or there are no more directories to search or there are no more patterns to match.

If `jc` could not locate a file which is needed for correct operation, e.g. a necessary symbol file, it terminates with an appropriate error message.

When creating a file, `jc` also uses redirection: the file is created in the directory that appears first in the search path list following the pattern that matched the file name. If that directory does not exist, it will be created automatically.

If no pattern matching the given filename can be found in the `jc.red` file, then the file will be read from (or created in) the current working directory.

Note: If a pattern matching the given filename is found then `jc` will *not* look into the current directory, unless it is explicitly specified in the search path.

`jc` can read class files (`*.class`), symbol files (`*.sym`), and precompiled files (`*.bod`) from ZIP and JAR archives. Make sure you specify archive file names with extension:

```
*.class = ./lib/myLib.jar; /usr/local/jdbc/driver.zip
```

Note: The compiler cannot write to archive files.

The base directory macro `$!` can be used in a directory name. It denotes the path to the redirection file. For instance, if the redirection file is placed in the directory `/usr/local/jet/bin`, then `$!/../lib/x86` denotes the directory

```
/usr/local/jet/lib/x86.
```

Sample redirection file:

```
jc.msg = /usr/local/jet/bin
*.class = ./myclass
*.sym = sym; $!/../sym.zip
```

Note: The **LOOKUP** equation, used in project files and on the command line, *extends* the list of search paths defined in the redirection file.

16.1.6 Options

JET options allow you to control file placement, code generation, internal limits and settings, etc. We distinguish between boolean options (or just options) and equations. An *option* can be set ON (TRUE) or OFF (FALSE), whereas an *equation* has a string value. In this chapter we only describe the syntax of setup directives that define and set options and equations. The full list of jc options and equations is provided in section 16.9.

Options and equations may be set in a configuration file (see 16.1.7), on the command line (see 16.4), and in a project file (see 16.8)).

The same syntax of a setup directive is used in configuration and project files and on the command line. The only difference is that spaces between tokens are permitted in files, but not on the command line. Option and equation names are case independent.

```
SetupDirective = SetOption
                | ShortSetOption
                | SetEquation
                | DeclareOption
                | DeclareEquation
SetOption      = '-' name ('+' | '-')
ShortSetOption = ('+' | '-') name
SetEquation    = '-' name '=' [ value ]
DeclareOption  = '-' name ':' [ '+' | '-' ]
DeclareEquation = '-' name ':=' [ value ]
```

All options and equations used by jc are predeclared.

Examples:

Directive	Means
-gendll+	GENDLL is set ON
+gendll	ditto
-stacklimit=10M	STACKLIMIT is set to 10MB
-Demo:+	DEMO is declared and set ON
-Vers:=1.0	VERS is declared and set to "1.0"

16.1.7 Configuration file

A configuration file is used to set the default values of options and equations (see 16.9) for all projects (or a set of projects). A non-empty line of a configuration file may contain a single compiler option or equation setup directive (see 16.1.6) or a comment. Arbitrary spaces are permitted. The "%" character indicates a comment; it causes the rest of a line to be discarded. **Note:** the comment character cannot be used when setting an equation.

The *master configuration file*, placed alongside the `jc` utility, usually contains default settings for the target platform and declarations of platform-specific options and equations, which may be used in project and template files.

Sample configuration file:

```
% this is a comment line
% Set predeclared equation:
  -stacklimit=32000000
% Set predeclared options:
  -inline+      % enable inlining
  -genasserts % do not generate asserts
% Declare new options and equations:
  -rmi:+
  -mode:=release
% end of configuration file
```

16.1.8 Compiler memory usage

Note: This section has nothing to do with your application's memory usage, which is discussed in 10.6.

The JET compiler uses garbage collection to reclaim memory. Most contemporary operating systems, including Linux, provide virtual memory. If the compiler exceeds the amount of available physical memory, the garbage collector becomes very inefficient. Thus, it is important to restrict the amount of memory that can be used by the compiler.

By default, the compiler automatically determines how much memory it may allocate, depending on system load. However, in this mode it will never use more than the amount of *available physical memory*. So when using JET on low-end systems you may need to specify the maximum compiler heap size explicitly, using the equation **COMPILERHEAP**. You can modify the default value of that equation in the configuration file (`jc.cfg`), but for better compiler performance you may need to adjust it on a per-project basis.

Increase **COMPILERHEAP** if you get the "out of memory" (F950) message. Vice versa, if you notice unusually intensive disk activity when compiling your program, it may indicate that the value of the **COMPILERHEAP** equation is too large for your system configuration.

16.2 Invoking the compiler

The JET compiler is integrated with the project subsystem into a single utility called `jc`. If invoked without parameters, `jc` outputs brief help information.

`jc` command line syntax is as follows:

```
jc { mode | option | file }
```

where `file`, depending on the operation mode, can be a bytecode file name or a project file name. See 16.4 for a full description of operation modes.

`option` is a compiler setup directive (See 16.1.6). All options are applied to all operands, notwithstanding their relative order on the command line. On some platforms, it may be necessary to enclose setup directives in quotation marks:

```
jc hello.class "-genstacktrace+"
```

See section 16.9 for the descriptions of all compiler options and equations.

16.3 Precedence of compiler options

The `jc` utility receives its options in the following order:

1. from the configuration file `jc.cfg` (See 16.1.7)
2. from the command line (See 16.2)
3. from the project file (if present) (See 16.8)

At any point during operation, the most recent setting is in effect. Thus, if the option **IN-LINE** was switched ON in the configuration file, but then switched OFF on the command line, the compiler will treat it as OFF.

16.4 Compiler operation modes

The JET compiler has the following operation modes:

Mode	Meaning
MAKE	Check dependencies and recompile (default)
PROJECT	Build all projects given on the command line
HELP	Print help and terminate

Both the PROJECT and MAKE modes have auxiliary operation submodes — OPTIONS (see 16.4.4) and EQUATIONS (see 16.4.5) that can be used to inspect the set of compiler options and equations and their values; the ALL (see 16.4.3) submode forces full rebuild of the project.

On the command line, the mode is specified with the “=” symbol followed by the mode name. Mode names are not case sensitive, and specifying an unique portion of a mode name is sufficient, thus

=PROJECT is equivalent to =p

Operation modes and options can be placed on the command line in arbitrary order, so the following two command lines are equivalent:

```
jc =make hello.class -inline+
jc -inline+ =m hello.class
```

16.4.1 MAKE mode

```
jc [=make] { FILENAME | OPTION }
```

In the MAKE mode, given a set of one or more classes, the compiler automatically determines all classes necessary for their operation. Starting from the classes specified on the command line, `jc` tries to find bytecode and symbol files of imported Java classes. It then does the same for each of the found classes until all required classes are determined and their bytecode or symbol files are located. If, for a particular class, a bytecode file could not be found, but a symbol file is present, it is assumed that the compiled class resides in a library that will be available at link time. If a symbol file could not be found either, the compiler displays a fault message and terminates. See section 16.6 for more details. All bytecode files found on the previous step are compiled.

Usually, the application's main class is specified on the command line. In this case, the linker is invoked automatically upon successful compilation. This feature allows you to build simple programs without creating project files.

For convenience, `jc` also allows you to specify `.jar` files on the command line:

```
jc MyApp.jar
```

16.4.2 PROJECT mode

```
jc =project { PROJECTFILE | OPTION }
```

The PROJECT mode is essentially the same as the MAKE mode except that the classes to be compiled are specified in a project file, which name is passed to `jc`. A project file specifies compiler options and a set of input files. See 16.8 for further details.

If file extension is omitted on the command line, `jc` will append the default extension `.prj`.

See also:

- MAKE operation mode: [16.4.1](#)
- Make strategy: [16.6](#)

16.4.3 ALL submodule

The ALL submodule forces full rebuild of the project as if it were built from scratch. Thus, the submodule ignores the results of previous build sessions and cancels the default (smart) build strategy. See 16.6 for details.

On the command line, the submode is specified with =a, for example

```
jc =p =a MyApp.prj
```

16.4.4 OPTIONS submode

The OPTIONS submode allows you to inspect the values of options which are set in the configuration file, project file and on the command line. It can be used alone or together with MAKE (see 16.4.1), and PROJECT (see 16.4.2) modes.

Note: The **PRJ** option affects the output in the MAKE mode.

In the PROJECT mode options are listed for each project file given on the command line.

See also the EQUATIONS submode.

16.4.5 EQUATIONS submode

The EQUATIONS submode allows you to inspect the values of equations which are set in the configuration file, project file and on the command line. It can be used together with MAKE (see 16.4.1), and PROJECT (see 16.4.2) modes.

Note: The **PRJ** option affects the output in the MAKE mode.

See also the OPTIONS submode.

16.4.6 HELP mode

In the HELP mode, jc prints information on its usage to the standard output. For instance, run

```
jc =help optsyntax
```

to get information on compiler options syntax, or

```
jc =help genstackalloc
```

to get help on the option **GENSTACKALLOC**.

If no topic or option name is specified, the list of available topics is printed.

16.5 Project Data Base

Project Data Base (PDB) is a directory created by the JET compiler at the very start of compilation. This directory will later hold all auxiliary files produced by the compiler. If you are compiling a project file, PDB will be created in the directory where that project file resides. Otherwise, it will be created in the current directory. The precise name and placement of Project Data Base are summarized in the following table:

Command line	PDB Placement	PDB Name
jc Foo.class	<i>current directory</i>	jetpdb
jc Bar.jar	<i>current directory</i>	jetpdb
jc =p some.prj	<i>some.prj directory</i>	some_jetpdb

Typically, Project Data Base contains several ZIP files (`obj.zip`, `sym.zip`, `bod.zip`, `rdf.zip`) and a few configuration files.

The main purpose of PDB is to store auxiliary files between different compilation sessions. Thus, PDB is only needed on the developer's machine and should not be deployed to enduser systems. Project Data Base also simplifies creation of multi-component applications as described in section [12.3](#).

16.6 Make strategy

By default, the JET Optimizer tracks changes in the project³ to avoid recompilation of the entire application if only *auxiliary* application files and/or project settings were modified. That is, the classes are not recompiled if you change project settings not affecting the generated code, e.g. add an icon to the executable or adjust the default GC settings.

If necessary, you may trigger full rebuild by specifying the ALL (see [16.4.3](#)) submode on the command line.

16.7 Control file preprocessing

The JET compiler reads the following control files during execution:

- a redirection file (see [16.1.5](#))
- a configuration file (see [16.1.7](#))
- a project file (see [16.8](#))
- a template file (see [A.4](#))

All these files are preprocessed when read according to the rules set forth in this section.

A control file is a plain text file containing a sequence of lines.

The following constructs are recognized and handled during control file preprocessing:

- macros of the kind $\$(name)$. A macro expands to the value of the equation *name* or, if such equation is not defined, to the value of the environment variable *name*.
- the *base directory* macro ($\$(!)$) This macro expands to the directory in which the file containing it resides.

³The Optimizer uses the information stored in PDB (see [16.5](#)) to determine if there were any changes in the project.

- directives denoted by the exclamation mark ("!") as a first non-whitespace character on a line.

Directives have the following syntax (all keywords are case independent):

```
Directive = "!" "NEW" SetOption | SetEquation
           | "!" "SET" SetOption | SetEquation
           | "!" "IF" Expression "THEN"
           | "!" "ELSIF" Expression "THEN"
           | "!" "ELSE"
           | "!" "END".
SetOption  = name ( "+" | "-" ).
SetEquation = name "=" string.
```

The `NEW` directive declares a new option or equation:

```
!new useawt+
!new mode=debug
```

The `SET` directive changes the value of an existent option or equation.

```
!set gendl1+
!set main=com/MyCorp/MyPackage/main
```

The above two directives are typically not used anywhere but in template files (see [A.4](#)), since all other control files permit more convenient syntax described in [16.1.6](#).

The `IF` directives tell the compiler to process or skip portions of control files depending on the value of `Expression`. `IF` directives may be nested.

The syntax of `Expression` is as follows:

```
Expression = Simple [ Relation Simple ].
Simple     = Term { "+" | OR Term }.
Relation   = "=" | "#" | "<" | ">".
Term       = Factor { AND Factor }.
Factor     = "(" Expression ")"
           | String
           | NOT Factor
           | DEFINED name
           | name.
String     = "'" { character } "'"
           | '"' { character } '"'.
```

An operand in an expression is either string, equation name, or option name. In the case of equation, the value of equation is used. In the case of option, a string "TRUE" or "FALSE" is used. The "+" operator denotes string concatenation. Relation operators perform case insensitive string comparison. The NOT operator may be applied to a string with value "TRUE" or "FALSE". The DEFINED operator yields "TRUE" if an option or equation name is declared and "FALSE" otherwise.

See project file example in section [16.8](#).

16.8 Project files

A project file is a plain text file of the following structure:

```
{ setup-directive }  
{ classloaderentry-directive | classpathentry-directive |  
  module-directive | batch-directive }
```

Setup directives define options and equations that all classes which constitute the project should be compiled with. See also [16.1.6](#) and [16.7](#).

Every line in a project file can contain not more than one directive. A line beginning with the “%” character is considered a comment and ignored.

It is recommended to specify only those settings in the configuration and redirection files which are applied to all your projects, and use project files for all project-specific options and redirection directives.

Project files are subject to preprocessing described in [16.7](#).

A simple project file:

```
-ignoreclassduplication+  
-ignorememberabsence+  
-classabsence=HANDLE  
  
!classpathentry ./SwingSet2.jar  
  -pack=noncompiled  
  -protect=nomatter  
  -optimize=all  
!end
```

Typical project file contains some options and equations settings, followed by a list of classpath entries, which describe the directories and JAR files with classes and application resources.

Given the sample project file above, the compiler will compile all classes from `SwingSet2.jar`, and optimize them for maximum performance. All resource files, such as images, will be packed into the resulting executable and can be loaded at run time.

The above shown project file is a typical example, but it does not cover the advanced lookup and conditional define directives supported by the JET project system.

More complex project file example:

```

% define default value of the equation 'mode'
!if not defined mode then
  -mode:=work
!end

% perform conditional settings
!if (mode = "work") then
  % turn off optimizations for faster compilation
  -inline-

  % set output file location
  -outputdir=debug
!else
  % turn on some optimizations
  -inline+

  % set output file location
  -outputdir=release
!end

% suppress console window at application startup
-gui+

% set input file locations
-lookup = *.jar = ./lib
-lookup = *.class = ./classes;./ext

% compile class and all its import
!module GUIApplication.class

% compile all classes in .jar file
!module SampleJAR.jar

% compile all .jar files from the 'ext' directory
!batch *.jar ext

```

The more complex project file shown above contains conditional sections that depend on a named variable. If the compiler is run as `jc =p advanced.prj`, then the value of `mode` is `work`; if the compiler is run as `jc =p advanced.prj -mode=release`, the value of `mode` is `release`, and then the other set of directives is used.

A project file can contain several **LOOKUP** equations, which allow you to define additional search paths. Given the sample project file above, the compiler will search JAR files within `./lib`, and class files within `./classes` and `./ext` directories. Placement for compiler-generated intermediate files (`.sym`, `.bod`, `.obj`, `.rdf`) is also controlled through the **LOOKUP** equation, as shown above.

The `!classpathentry`, `!classloaderentry`, `!module`, and `!batch` directives are described below.

16.8.1 The `!classpathentry` directive

```
!classpathentry name
    -pack      = none | noncompiled | all
    -protect   = nomatter | all
    -optimize= autodetect | all
!end
```

A classpath entry is a JAR file or a directory that contains either class files, or resource files, or both. Each `!classpathentry` directive comprises a name, which is a full or relative path to the directory or the JAR file, and up to three equations. The equations specify the optimization, protection and deployment settings for this entry.

- **pack** specifies a packaging mode:
 - *noncompiled* : resource files and non-compiled class files from this classpath entry are packed to the executable;
 - *none* : neither class nor resource files from this classpath entry are packed; the original JAR file or directory must be distributed along with your application.
 - *all* : the whole JAR is packed, including all classes and resources⁴
- **protect** specifies a protection mode:
 - *all* : all classes from this entry are compiled to native code to provide maximum protection from reverse engineering;
 - *nomatter* : The JET Optimizer detects which classes are unlikely to be used at run time, and may deploy them in the bytecode form, thus reducing the download size of your application;
- **optimize** specifies an optimization mode:
 - *all* : all classes from the entry are optimized for maximum performance, at the cost of a larger executable size.
 - *autodetect* : the JET Optimizer determines optimization level suitable for each class;

⁴ An example to consider is a Java API which implementation requires the original class files to be available at run time. For instance, Java Crypto Extensions also known as security providers check the sizes of their class files during execution. In such cases, the classes serve as *both* program code and resources. Even if all the class files are pre-compiled, you have to make them available to the running application. Packing such JAR files as a whole easily resolves the problem.

Let us consider an example. Suppose your Java application is run with the following command line:

```
java -cp core.jar;utils.jar;lib\foo.jar com.product.Main
```

There are three classpath entries:

- `core.jar` contains core classes whose performance is critical, and they must be thoroughly protected from reverse engineering;
- `utils.jar` contains auxiliary classes which are not that critical, let compiler find a balance between performance and code size for these classes
- `foo.jar` is a third-party API that does not require IP protection for the implementation classes;

The respective fragment of the JET project file for such an application would look like this:

```
% turn on global optimizer
+global

!classpathentry ./core.jar
    -pack      = noncompiled
    -protect   = all
    -optimize  = all
!end'

!classpathentry ./utils.jar
    -pack      = noncompiled
    -protect   = nomatter
    -optimize  = autodetect
!end'

!classpathentry ./lib/foo.jar
    -pack      = noncompiled
    -protect   = nomatter
    -optimize  = autodetect
!end
```

The Global Optimizer (see [13](#)) uses these settings as follows. The resulting executable will contain all code of `core.jar` fully protected and optimized for maximum performance. Other classpath entries will be analyzed, and their classes will be divided in two groups:

- Classes that will be probably used at run time are optimized for performance;
- Classes that are unlikely to be used at run time are optimized for size;

As you already guessed, the equations `protect`, `optimize` and `pack` directly correspond to the values of **Protect**, **Optimize**, and **Pack into exe** columns of the Classpath grid of the JET Control Panel. Refer to the JET Control Panel chapter (see [4.7.2](#)) for more details.

Note: The `protect`, `optimize` and `pack` equations can be used in the global context, outside of the scope of a `!classpathentry` directive. In this case, the specified values are used as default settings and can be overridden by the respective equations within `!classpathentry` directive scope.

16.8.2 The `!classloaderentry` directive

```
!classloaderentry loader-type name
    -pack      = none | noncompiled | all
    -protect   = nomatter | all
    -optimize= autodetect | all
!end
```

This directive is a generalized version of the `!classpathentry` directive (see [16.8.1](#)). It also allows specifying *loader-type* - one of the standard or custom classloaders supported by the static compiler. Currently, this directive is used in projects for Tomcat Web applications (see [Chapter 8](#)) and Eclipse RCP applications (see [Chapter 7](#)).

16.8.3 The `!module` directive

```
!module file-name
```

file-name is a name of a file which should be compiled, linked, or otherwise processed when a project is being built, e.g. a bytecode file, an additional library, a resource file, etc. The type of a file is determined by its extension (by default `.class` is assumed). The compiler only processes Java bytecode files (`.class`, `.jar` and `.zip`) and files of a few other types. The remaining files are only taken into account when a response file is created (see [A.4](#)).

The compiler recursively traverses imports of all explicitly specified classes (including those in jar files) and builds the full list of classes used in the project. Thus, a project file for an executable program would usually contain a single `!module` directive for the file which contains the main class and, optionally, several `!module` directives for dynamically loaded classes that are not explicitly imported, and jar/zip files that have to be compiled entirely.

Note: using `!module` for adding classes or jar files to the project is obsolete; it is recommended to use the `!classpathentry` directive instead.

16.8.4 The `!batch` directive

```
!batch pattern directory
```

The `!batch` directive is intended for adding a group of files of the same type to the project's module list and works as follows:

1. *pattern* is sought along the redirections and **LOOKUP** directives.
2. *directory* is appended to each file lookup directory defined for *pattern*. If the resulting directory exists, all files matching *pattern* that reside in that directory and all its subdirectories are appended to the project module list.
3. Similarly, if the lookup archive files defined for *pattern* contain *directory*, all files from it and its subdirectories are appended to the project module list.

The `!batch` directive is typically used for compilation of an entire Java package. Notion of a package corresponds with that of a directory in file system terms. Therefore, to compile the entire package `com.MyCorp.MyPkg`, you have to add the following line to the project file:

```
!batch *.class com/MyCorp/MyPkg
```

Note: the `!batch` directive is obsolete. It is recommended to put all classes to jars and then use the `!classpathentry` directive to add the jars to the project.

16.8.5 The `!uses` directive

```
!uses path-to-the-project-file
```

The `!uses` directive imports Project Data Base (see 16.5) associated with the specified project file. You should use this directive when compiling multi-component applications (see 12.3) or optimizing JIT cache (see 11.6). Using PDB, the compiler retrieves the necessary information about the classes compiled into other components.

In the case of multi-component applications (see 12.3), compiler also determines the name of shared library compiled with the specified project file and forces its load-time linking.

16.9 Compiler options and equations

A rich set of JET compiler options allows you to control the generated code, and the internal limits and settings. We distinguish between boolean options (or just options) and equations. An option can be set ON (TRUE) or OFF (FALSE), while an equation has a string value.

16.9.1 Options

Options control the process of compilation, run-time checks and code generation. An option can be set ON (TRUE) or OFF (FALSE).

A compiler setup directive (See 16.1.6) is used to set the option value or to declare a new option.

Option	Meaning
COMPILEWITHFIXEDHEAP	use baseline compiler if there is not enough memory
GLOBAL	turns on the global optimizer
DISABLECLASSSAVING	disable caching of JIT-compiled class files
DISABLESTACKTRACE	disable support for stack trace
GENDLL	generate a shared library
GENSTACKALLOC	generate code with stack allocated objects
GENSTACKTRACE	generate stack trace data
GUI	suppress console window <i>Windows</i>
IGNORECLASSDUPLICATION	ignore duplicated classes
IGNOREMEMBERABSENCE	ignore usage of absent class members
INLINE	enable in-line method expansion
MULTIAPP	enable the multi-app mode for the executable
NOFF	suppress notices
SPLASHCLOSEONAWTWINDOW	hide splash when AWT window opens
SPLASHCLOSEONCLICK	hide splash on mouse click
SPLASHGETFROMMANIFEST	get splash screen image from JAR manifest
VERIFYALL	force strict verification
WOFF	suppress warning messages

Table 16.1: JET compiler options

Options may be set in a configuration file (see [16.1.7](#)), on the command line (see [16.4](#)), in a project file (see [16.8](#)). At any point of operation, the last value of an option is in effect.

The alphabetical list of options along with their descriptions may be found in section [16.9.2](#). See also table [16.1](#) (page [191](#)).

16.9.2 Options reference

Run-time check options are ON by default. If not explicitly specified, other options are OFF (FALSE) by default.

COMPILEWITHFIXEDHEAP

This option instructs the JET compiler what to do when it requires more memory than specified by the `COMPILERHEAP` equation.

If this option is OFF, the compiler terminates with the respective message.

If this option is ON, the method that requires more memory to compile, is compiled in *baseline* mode — with weaker optimizations. Then, the compilation process continues normally.

When the compiler finishes its job, it prints a notice if some methods were compiled with weaker optimization level, either due to the lack of memory, or due to other reasons. See [10.16](#) for more information.

This option is OFF by default.

GLOBAL

This option turns on the global optimizer that compiles classes of your application together with classes of the Java standard API, providing faster startup and smaller download size, at the cost of longer compilation time.

This option is OFF by default.

See [13](#) for more information.

DISABLECLASSSAVING

The process of JIT cache optimization (see [11.6](#)) involves flushing of dynamically loaded class files to the JIT cache directory, enabled by setting the property `jet.jit.save.classes`. It may be desirable to disable this behavior in the end-user version of the application, e.g. for security reasons. If that is the case, turn this option ON and the JET runtime will ignore the property setting.

This option is OFF by default.

DISABLESTACKTRACE

If this option is set ON, stack trace output (see [10.11](#)) will not work in the resulting executable even if the `jet.stack.trace` property is set.

Note: The executable will still contain symbolic information used when producing stack trace output, because that information is also required for reflection to work.

You may wish to enable this option when building the end-user version of your application in order to make its reverse engineering more complicated.

This option is OFF by default.

GENDLL

Setting this option ON forces generation of object and symbol files suitable for dynamic linking, and linking them into a shared library.

See Chapter [12](#) for details on building shared libraries.

GENSTACKALLOC

If the option is set ON, the compiler places some of the dynamically allocated objects on the stack rather than on the heap. This optimization is performed according with the results of escape analysis and is applied only to short living objects.

According to the Java Language Specification, all objects are allocated on the heap by the `new` operator. However, a particular object that is never assigned to a static or instance field may be safely allocated on the stack. Typically, these are temporary objects local to a particular method. Stack allocation works much faster and also reduces GC impact on performance.

Note: enabling this option may reduce the program's scalability as each thread shall require more stack space, so it is recommended to set this option OFF for server-side applications running thousands of threads, if their scalability is more important than throughput.

GENSTACKTRACE

If this option is set ON, the compiler propagates information about source file names and line numbers into the resulting executable, so that it could appear in the output produced by the `printStackTrace()` method. Otherwise, such output contains class names only.

Note: The side effect of setting this option ON is the substantial growth (20% or more) of the resulting executable.

The property `jet.stack.trace` property still has to be set explicitly for stack tracing to work.

This option is OFF by default.

GUI

This option is ignored on Linux.

On Windows, if the option is set ON, the compiler generates executables for the Windows subsystem (no console window shall appear when the executable is run by clicking on its name or icon in the Windows Explorer.) Otherwise the compiler generates executables for the console subsystem. This is the default.

IGNORECLASSDUPLICATION

Some third-party APIs are known to contain copies of Java platform classes. Platform classes are precompiled into JET runtime shared libraries when you create a profile (see 2.3). By default, the option is ON and the compiler ignores this inconsistency. **Warning:** The compiler will still use precompiled classes.

By setting this option ON, you force the compiler to issue an error on an attempt to compile a class that was precompiled.

Note: You may override endorsed standards (e.g. packages `org.omg.CORBA` or `org.xml.sax`) during profile creation. See 2.3 for details.

IGNOREMEMBERABSENCE

The option determines the behavior of the compiler when it is unable to find a class member (field or method) referenced in the code.

By default, the option is ON and the compiler generate code throwing `java.lang.NoSuchMethodError` or `java.lang.NoSuchFieldError` for locations where such members are referenced. However, if the code referencing that member never receives control at run time, the program will execute OK on a JVM.

By setting this option ON, you tell the compiler to report errors upon encountering such inconsistencies.

INLINE

If the option is ON, the compiler tries to expand methods in-line. In-line expansion of a method eliminates the overhead produced by a method call, parameter passing, register saving, etc. In addition, more optimizations become possible because the optimizer may process the actual parameters used in a particular call.

A method is not expanded in-line under the following circumstances:

- the method is overridden and the compiler is unable to determine the method of exactly which class is called
- the method is deemed too complex or too large by the compiler;
- there are too many calls of the method

The equations **INLINELIMIT** and **INLINETOLIMIT** control the aggressiveness of this optimization.

See [10.3](#) for more information.

MULTIAPP

If the option is turned ON, the compiler generates an executable that may have multiple entry points thus enabling you to compile more than one application into a single executable.

You can select the application to be launched via the command line arguments using an extended version of the `java` launcher command line syntax. For more details, refer to section [10.7](#)

By default, this option is turned OFF.

NOFF

When the option `NOFFnnn` (e.g. `NOFF328`) is set ON, the compiler does not report the notice `nnn` (328 in the above example). See the `jc.msg` file for notice texts and numbers.

`-NOFF+` disables all notices.

VERIFYALL

When this option is set ON, the compiler verifies that all class files strictly obey language constraints. Otherwise, classes are still verified, but some checks are disabled, namely those that are disabled in Sun HotSpot VM for non-remote classes by default.

This option is OFF by default.

SPLASHCLOSEONAWTWINDOW

When the option is set ON, the splash screen hides when the application opens any AWT window.

Note that if this option is turned OFF and the equation **SPLASHCLOSEONTITLE** is not specified, the splash screen disappears when the application opens any window, no matter if it is AWT or native one.

This option is valid only for profiles based on Java SE 6. For other profiles, the option is always turned OFF regardless of the specified value.

See [10.10.1](#) for details.

This option is OFF by default.

SPLASHCLOSEONCLICK

When the option is set ON, the splash screen hides when user clicks on it.

This option can be turned OFF only if the active profile is based on Java SE 6. Otherwise, the option is always switched ON regardless of the specified value.

See [10.10.1](#) for details.

By default, the option is turned OFF.

SPLASHGETFROMMANIFEST

When the option is set ON, the Runtime displays the splash screen using the image from the manifest of JAR file that contains the main application class.

This option is valid only for profiles based on Java SE 6. For other profiles, the option is turned OFF regardless of the specified value.

This option is ignored if the equation **SPLASH** is specified.

See [10.10.1](#) for details.

WOFF

When the option `WOFFnnn` (e.g. `WOFF325`) is set ON, the compiler does not report the warning `nnn` (325 in the above example). See the `jc.msg` file for warning texts and numbers.

`-WOFF+` disables all warnings.

16.9.3 Equations

An *equation* is a pair (name,value), where `value` is, in the general case, an arbitrary string. Some equations have a limited set of valid values, some may not have the empty string as a value.

A compiler setup directive (See [16.1.6](#)) is used to set an equation value or to declare a new equation.

Equations may be set in a configuration file (see [16.1.7](#)), on the command line (see [16.4](#)) and in a project file (see [16.8](#)). At any point of operation, the most recent value of an equation is in effect.

The alphabetical list of equations may be found in section [16.9.4](#). See also table [16.2](#) (page [210](#)).

16.9.4 Equations reference

OPTIMIZE

Sets optimization mode. Valid values are “ALL”, and “AUTODETECT”.

If set to “ALL”, all class files are compiled to native code with current optimization settings. Typically, this mode provides the best performance, however, the size of resulting executable may be quite large.

If set to “AUTODETECT”, the compiler detects which classes will be used at run time, and optimizes those classes only, leaving the rarely used code in compact, but

no-optimized form. This mode provides smallest binaries, but it may negatively affect application performance, if the compiler assumptions fail.

If this equation is placed within a “!classpathentry .. !end” scope, it affects that classpath entry only. Otherwise, the setting has global effect.

PROTECT

Sets code protection mode. Valid values are “ALL”, and “NOMATTER”.

If set to “ALL”, all class files are compiled to native code, thus protecting them from reverse engineering.

If set to “NOMATTER”, the JET Optimizer may avoid compilation of some classes in favor of reducing the download size of the application.

If this equation is placed within a “!classpathentry .. !end” scope, it affects that classpath entry only. Otherwise, the setting has global effect.

CRYPTSEED

Enables encryption of data in the generated executable. Valid value is any ASCII string without spaces containing at most 128 characters.

This equation enables protection of the executable data, including string constants, reflection info, resource files packed into the executable, etc. For more details, see [5.2](#).

Using the specified seed string, the Optimizer generates a key and encrypts the data.

PACK Sets code deployment mode. Valid values are “NONE”, “NONCOMPILED”, and “ALL”.

If set to “NONCOMPILED”, everything but compiled .class files is packed to the executable. This is the default mode. Simply speaking, in this mode non-compiled class files and all resource files are packed to the executable, and can be accessed by the application at run time. The original classpath entry is no longer required for the running application.

If set to “NONE”, neither original class files, nor resource files are packed into the executable. The classpath entry should be distributed along with the application executable.

If set to “ALL”, all class and resource files are packed to the executable. This makes sense when compiling JARs of security providers and other APIs that require class files to be available at run time.

If this equation is placed within a “!classpathentry .. !end” scope, it affects that classpath entry only. Otherwise, the setting has global effect.

CLASSABSENCE

This equation controls the actions taken by the compiler when it is unable to find an imported class.

Valid values are “ERR”, and “HANDLE” (the default).

If set to “ERR”, the compiler reports an error if an imported class is unavailable. You should add the location containing that class file to the classpath. You may

use the mode to force the compiler to perform consistency checks for the set of compiled classes.

However, if that imported class is not actually used at run time, the program will execute OK on a JVM. This *bytecode inconsistency* (see 10.15) is often present in third-party libraries. If you encounter such problem, switch this equation back to “HANDLE”. If some of the absent classes become available, they will be loaded by the JET’s dynamic compiler (reflective shield (see 11.5)) and the program will execute correctly, though loss of performance is possible.

COMPILERHEAP

Sets the maximum amount of heap memory (in bytes), that can be used by the compiler. For systems with virtual memory, we recommend to use a value which is less than the amount of physical memory.

See 16.1.8 for more information.

DECOR

The equation controls output of the `jc` utility. The value of equation is a string that contains any combination of letters "h", "p", "r", "t" (capital letters are also allowed). Each character turns on output of

- h header line, which contains the name and version of the compiler’s front-end and back-end
- p progress messages
- r compiler report: number of errors, lines, etc.
- t the summary of compilation of multiple files

By default, the equation value is "hp`r`t".

DLLNAME

This equation has to be set to the name of the DLL into which the currently compiled class will be linked. It has no effect if **GENDLL** option is OFF.

This equation has been superseded by **OUTPUTNAME**. It is still recognized by the compiler, but may be removed in future versions. The JET Control Panel does the replacement automatically when you open a “legacy” project.

ERRFMT

Sets the error message format. See A.2 for details.

ERRLIM

Sets the maximum number of errors allowed for one compilation unit (by default 16).

EXPIRE

Set an expiration date for the resulting executable. The value is the expiration period that is either a number of days *after the build date of the executable* or a fixed date (both formats are supported.)

See 10.9 for details.

EXPIREMSG

Set an expiration message for the resulting executable. The value is the message text to be displayed as the evaluation period is over.

See [10.9](#) for details.

IMAGEBASE

This equation is ignored on Linux.

By default, all Windows executables (EXEs and DLLs) produced by the JET compiler use the same *image base address*, which is 0x400000. You change the default base address by setting this equation:

```
-imagebase=0x10000000
```

INLINELIMIT

Sets the maximum size of method that may be considered for inline substitution into another method (see the option **INLINE**). The units are syntax tree nodes.

See also **INLINETOLIMIT**.

INLINETOLIMIT

Sets the maximum size of methods permitting inline substitution (see the option **INLINE**). The units are syntax tree nodes. See also **INLINELIMIT**.

JETVMPROP

Defines system properties to be hard-wired into the output executable.

```
-JETVMPROP=-Dp1name -Dp2name:p2value
```

See [10.5](#) for more information.

JETRT

This equation controls the JET runtime flavor that serves the compiled application.

Valid values: “CLASSIC”, “DESKTOP”, and “SERVER”.

See [10.2](#) for more details.

Note: Enterprise Edition of Excelsior JET allows enabling any kind of Runtime, Professional Edition does not feature Server Runtime, and Standard Edition has only Classic Runtime.

LINK

Defines the command line to be executed after successful compilation of a project. As a rule, the equation is used for calling a linker or make utility.

Normally, you do not need to change the default value of this equation.

LOOKUP

The equation is used to define additional search paths that would complement those set in the redirection file.

Syntax:

```
Lookup ::= -LOOKUP = RegExp = Source {";" Source}
Source ::= Directory | Archive
```

jc can read class files (*.class), symbol files (*.sym), and precompiled files (*.bod and *.rdf) from zip and jar archives. Make sure to specify archive file names with extensions. A configuration or project file may contain several **LOOKUP** equations; they are also permitted on the command line.

Example:

```
-lookup=*.class=./class;/usr/local/MyJars/MyLib.jar
```

See also [16.1.5](#) and [16.8](#).

MAIN

By default, upon encountering a method with signature `static public void main (String[])`, the compiler designates it as a program's starting point. If your program contains more than one class with such method, the linker would report duplicate names and fail to build the program. This equation allows you to explicitly specify the name of the class, the `main` method of which has to be the program's entry point.

Notes:

- If the class belongs to a certain package, make sure to specify its full name, using slashes ("/") as separators:

```
-main=com/MyCompany/MyMainClass
```

OUTPUTDIR

This equation can be used to specify the pathname of the directory into which the compiler places the resulting file (executable or shared library.) If this equation is not set, the target directory is defined as follows:

1. If the compiler is run in the PROJECT mode (see [16.4.2](#)), the directory where the project file resides is used
2. Otherwise, the compiler is run in the MAKE mode (see [16.4.1](#)), the current directory is used

OUTPUTNAME

This equation can be used to explicitly set the name of the resulting file (executable or shared library). If this equation is not set, the name is computed as follows:

1. If the **GENDLL** option is switched ON and **DLLNAME** is not empty, its value is used (backward compatibility with JET versions prior to 3.5 is thus ensured.)
2. Otherwise, if the **GENDLL** option is switched ON but **DLLNAME** is empty, jc terminates with an error message suggesting the user to specify a value for **OUTPUTNAME**.
3. Otherwise, if the compiler is invoked in the project mode (see [16.4.2](#)), project file name without path and extension is assumed.

4. Otherwise, if the **MAIN** equation is specified, the name the main class (without package names) is used.
5. Otherwise, if there is just one class file name present on the command line, its name (without package names) is used.
6. Otherwise, if there is just one jar file name present on the command line, and it has main class specified in the manifest, the name of that class (without package names) is used.
7. Otherwise, jc terminates with an error message suggesting the user to specify the **MAIN** equation.

Note: If the result will be a shared library, its name *must* be known at compile time. Do not rename shared libraries created by JET if they will be referenced from other JET-compiled executables.

See [12.3.1](#) for more information.

PRJ

In the MAKE operation mode, this equation defines a project file to read settings from. In the PROJECT mode, the compiler sets this equation's value to the project file name from the command line. See [16.4.2](#).

PROJECT

If a project file name is defined, the compiler sets the equation to a project name without file path and extension. For example, if the project file name is `prj/Work.prj`, the value of the equation is set to `Work`. The equation is typically used in template files (see [A.4](#)), for instance, to set the name of the executable file.

SAPROFMODE

Enables/disables the Startup Accelerator (see [14.2.2](#)).

Valid values are "NONE", "JETCPOONLY", and "ALWAYS".

If set to "ALWAYS", application will be automatically run and profiled immediately after each build, including unattended builds with the JC compiler.

"JETCPOONLY" value enables the profiling only if the application is compiled from within the JET Control Panel.

"NONE" disables the Startup Accelerator in the JET Control Panel and the JC compiler. You may use `xstartup` utility (see [14.3.2](#)) in the case you want to profile and optimize the executable independently of its compilation.

SAPROFTIMEOUT

If the Startup Accelerator (see [14.2.2](#)) is enabled, this equation specifies the duration of the profiling session after which the application will be automatically terminated.

Value is the timeout in seconds with the default of 20.

If set to "INFINITE", the application will not be terminated automatically.

STACKLIMIT

Sets the maximum size of the stack in the generated program. The value is set in bytes.

STARTUPPROFILE

Enables the adaptive optimization of application start up time (see chapter 14 for details). The value is the pathname of a startup profile file.

SPLASH

Assigns the resulting executable a splash screen image. The value is the full path of an image file. The set of supported image formats depends of the active profile (see 2.3).

The supported formats are PNG, GIF and JPEG, including images with transparency, translucency, and animation.

See 10.10.1 for details.

SPLASHCLOSEONTITLE

Hides the splash screen when the application opens a window whose title starts with the specified prefix. The value is the prefix string.

Note that if this option is not specified and the equation **SPLASHCLOSEONAWTWINDOW** is OFF, the splash screen hides when any application's window is opened.

See 10.10.1 for details.

SPLASHMINTIME

Delays the splash on the screen for the specified time (unless the user closes it by mouse click.) The value is the delay in seconds.

This equation is ignored if the option **SPLASHCLOSEONAWTWINDOW** is ON.

See 10.10.1 for details.

TEMPLATE

Specifies the linker response file template. See A.4.

16.9.5 Internal options and equations

In project files created by the JET Control Panel (see 4), you may encounter certain internal options and equations not described in the previous sections. You *should not* use these options and equations when writing project files manually or editing auto-generated projects.

16.10 Compiler messages

This chapter gives explanation for compiler diagnostics. For each error, an error number is provided along with a text of error message and an explanation. An error message can contain a format specifier in the form `%d` for a number or `%s` for a string. In this case, an argument (or arguments) is described on the next line.

In most cases the compiler prints a source line in which the error was found. The position of the error in the line is marked with a `#` sign placed directly before the point at which the error occurred.

16.10.1 Java bytecode front-end errors and warnings

F020

```
undeclared identifier %s.%s
```

A referenced field or method does not exist in the imported class. This is typical situation in big projects.

Suggested solution: switch to the Options Page (see 4.8), select the **Advanced** tab, and set the **Ignore member absence** mode. Please refer to JET consistency checks (see 4.8.4) for more details.

W918

```
java.lang.NoSuchFieldError will be thrown here
```

This warning is issued if the option **IGNOREMEMBERABSENCE** is ON and a field not present in an imported class is referenced in the program.

W919

```
java.lang.NoSuchMethodError will be thrown here
```

This warning is issued if the option **IGNOREMEMBERABSENCE** is ON and a method not present in an imported class is referenced in the program.

16.10.2 Project system errors

E370

```
No class contains static method "main([Ljava/lang/String;)V"
```

The compiler could not determine the program's entry point because no class in the compilation set has a method `static public void main (String[])`. Add a class with such method to the compilation set.

E371

```
Invalid MAIN equation specified: class %s not found
```

The class specified in the **MAIN** equation could not be found along the classpath. Add the appropriate directory, jar, or zip file to the classpath or adjust the value of the **MAIN** equation.

E374

```
The class specified in the MAIN equation does not have the
```

```
"public static void main(String[])" method
```

The class you specified in the **MAIN** equation was found, but it turned out that the method `public static void main(String[])` is absent. Check that the correct version of the class is accessible to the compiler, or specify a different class. Valid values for the **MAIN** equation are listed below the error message.

E375

```
The MAIN equation must be specified to resolve ambiguity
```

There is more than one class having a method `public static void main(String[])` in the classpath, so the compiler cannot designate the program's entry point automatically. Specify the name of the desired class using the **MAIN** equation, valid values for which are listed below the error message.

E349

```
file %s not found
```

The file is specified as a part of the project file but could not be found, perhaps you deleted it by accident. If you created the file manually, recreate it again. If the file ends with ".usg", it was automatically generated. In this case, use the test run (see [13.4.2](#)) to re-generate it.

E350

```
Error parsing .usg file: Factor is already opened
```

E351

```
Error parsing .usg file: Factor is not opened
```

E352

```
Error parsing .usg file: Factor is not closed
```

One of the .usg files included to your project contains syntax errors as a result of manual editing. If you intend to edit.usg files, contact Excelsior for details on their syntax.

F402

```
#compile "%s" ("%s" not found)
```

Neither symbol nor bytecode file could not be found for an imported class.

There is a chance you missed a JAR file when specifying the class path for your project. If this is the case, add the JAR file on the Start Page (see [4.5](#)).

If you are sure all relevant JARs and directories with class files are specified, switch to the Options Page (see [4.8](#)), select the **Advanced** tab, and set the "**Class absence results in**" combo box to "**Dynamic compilation**". Please refer to JET consistency checks (see [4.8.4](#)) for more details.

E955

```
class/sym conflict
```

Classpath entries of your project contain two or more copies of the same class file. Please switch to the Options Page (see [4.8](#)), select the **Advanced** tab, and set the "**Ignore conflicting classes**" mode. Please refer to JET consistency checks (see [4.8.4](#)) for more details.

F424

```
file create error: %s
```

The specified file could not be created.

Most probably, you do not have sufficient access rights. Also check if the file system is full.

F425

```
file open error: %s
```

The specified file could not be opened.

You do not have sufficient access rights or the file is already opened in exclusive mode.

F448

```
#file "%s": name conflict with class %s declared in the
file
```

Compiler encountered a difference between name of a class and name of a file this class is defined in. Usually, this happens when a file name and a class name differs with case only (like file `Hello.class` with class `hello` in it).

Solution: Provide JET with correct case-sensitive class name (either in project file or in command line).

W640

Wrong value "%s" for the PACK equation. Permitted values of the PACK equation are NONE, NONCOMPILED or ALL.

W642

Wrong value "%s" for the OPTIMIZE equation. Permitted values of the OPTIMIZE equation are AUTODETECT or ALL.

W643

Wrong value "%s" for the PROTECT equation. Permitted values of the PROTECT equation are NOMATTER or ALL.

See description of the `!classpathentry` (see [16.8.1](#)) directive for details on the syntax and possible values of these equations.

W641

The `%s` option is obsolete. Please, set PACK to either ALL or NONCOMPILED

Since version 4.5, JET uses new syntax for describing classpath entries where class files and resources can be found. Therefore, when reading project file with obsolete options, the warning is issued.

These options are fully supported, so you may ignore the warnings safely. However, we recommend you to read the description of the `!classpathentry` (see [16.8.1](#)) directive for details on the PACK equation syntax and its possible values.

W620

Class %s within usgfile has not found in the project **W621**

Method %s within usgfile has not found in the project **W622**

Field %s within usgfile has not found in the project

The usage list file (`.usg`) contains references to the entities not found in your project. Typically, this happens when the application is updated, while the `.usg` file is not. Recreate

your .usg file using the test run (see [13.4.2](#)).

E623

Cannot make backup file for "%s". Check access properties for it.

JET failed to rename the file before overwriting it with the fresh copy. Make sure the current user has write/rename rights to the given file.

F603

Incorrect "%s.usg" file version

The usage list file (.usg) is obsolete. Recreate it using the test run (see [13.4.2](#)).

F475

```
#project contains !classpathentry directive without !end
```

Your project file contains a syntax error. Please refer to the description of the !classpathentry (see [16.8.1](#)) directive for details on its proper syntax.

F480

```
#classpath entry not found "%s"
```

A JAR file or a directory referred from the project file is not found. Make sure all components of your project are accessible and the paths in the project file are correct.

16.10.3 Symbol files read/write errors

F190

```
incorrect header in symbol file "%s" (sym-file name)
```

A symbol file for the given class is corrupted. Recompile it.

F191

```
incorrect version of symbol file "%s" (%d instead of %d)  
(sym-file name, sym-file version, current version)
```

The given symbol file was generated by a different version of the compiler. Recompile the class file or use the respective versions of the compiler and/or symbol file.

F192

```
key inconsistency of imported module "%s" (sym-file name)
```

The error occurs if a symbol file was changed but not all its clients (classes that uses it) were recompiled. For example, if class A uses classes B and M and B in turn uses M:

```
class M {
}

class B extends M {
}

class A {
    B b;
}
```

Let us recompile `M.class`, `B.class`, and then add a new instance field in M and recompile it again. The error will be reported when compiling `A.class`, because version key of module M used through B is not equal to the version key of M used directly.

To fix the problem modules must be compiled in appropriate order. We recommend to use the JET compiler make facility, i.e. to compile your program in the `MAKE` (see [16.4.1](#)) or `PROJECT` (see [16.4.2](#)) operation mode. If you always use the make facility this error will never be reported.

F194

module name does not match symbol file name "%s" (module name)

A symbol file name (without the `.sym` extension) must be equal to the actual name of the class.

F195

cannot read symbol file "%s" generated by %s (module name, compiler name)

F638

Incorrect header of meta sym file "%s"

A symbol file was generated by other version of the JET compiler. Rebuild all of your projects using the same version of the compiler.

16.10.4 Warnings

In many cases a warning may help you to find a bug or a serious drawback in your source text. We recommend not to switch warnings off and carefully check all of them. In many cases warnings may help you to find and fix bugs very quickly.

W300

variable "%s" declared but never used

This variable is of no use, it is not exported, assigned, passed as a parameter, or used in an expression. The compiler will not allocate space for it.

W301

parameter "%s" is never used

This parameter is not used in the method.

W303

method "%s" declared but never used

This method is not exported or called. The compiler will not generate it.

W310

infinite loop

Execution of this loop (`while`, `for` or `do`) will not terminate normally. It means that statements after the loop will never be executed and the compiler will not generate them. Check that the loop was intentionally made infinite.

W311

unreachable code

This code cannot be executed and the compiler will not generate it (this is called dead code elimination). It may be statements after a `return` or `throw`, infinite loop, statements under constant false condition (`if (false)`), etc.

W312

loop is executed exactly once

It may be a loop like

```
for (i=1; i<2; i++) ...;
```

or

```
while (true) { ...; break;}
```

Check that you wrote it intentionally.

W314

variable "%s" has compile time defined value here

The compiler was able to determine the run-time value of the given variable (due to constant propagation) and will use it instead of accessing the variable. For the following example:

```
i=5; if (i == 5) S;
```

the compiler will generate the code as if the source was:

```
i=5; S;
```

This warning is reported for local variables only.

W315

`java.lang.NullPointerException` will be thrown here

After constant propagation, the compiler determined that the value of an object variable is null when access to its instance field or method occurs, e.g:

```
p=null;
p.field=1;
```

The code will be generated and will throw "NullPointerException" at run-time.

This warning is reported for local variables only.

W318

redundant for statement

The `for` statement is redundant (and not generated) if its body is empty or it may be determined at compile-time that it would be executed zero times.

16.10.5 Native back-end warnings**W900**

redundant code eliminated

This warning is reported if a program fragment does not influence flows of control and data, e.g.:

```
i=1;
i=2;
```

The first assignment is redundant and will be deleted (unless `i` is declared as volatile).

W901

redundant code not eliminated - can throw exception

The same as W900, but the redundant code is preserved because it can throw an exception, e.g.:

```
i = a / b; (* would throw exception if b == 0 *)
i=2;
```

W902

constant condition eliminated

The warning is reported if a boolean condition can be evaluated at compile-time, for example:

```
if (i==1 & i==1) (* the second condition is true *)
```

or

```
j=2;
if (i==1 | j!=2) (* the second condition is false *)
```

W903

method result is not used

The compiler ignores method result, like in:

```
if (foo());
```

16.10.6 Native back-end errors

This section describes errors reported by a native code generator (back-end). The code generator is invoked only if no errors were found by a language parser.

F950

out of memory

The JET compiler could not compile your project due to the lack of free memory. Use the "**Create build script**" button to generate a build script file, then close all running applications *including the JET Control Panel*, and run the build script just created. This way, you give the compiler the maximum amount of memory to operate.

If the problem still persists, edit the JET config file (`jc.cfg`) as follows, and then run the compilation again.

- set `COMPILERHEAP` to 70-90% of your computer RAM;
- turn on `COMPILEWITHFIXEDHEAP`;

F951

```
expression(s) too complex
```

The compiler has failed to generate an expression due to its complexity. This should not normally occur. If you see this message, please file a defect report.

16.10.7 Internal errors

Internal compiler errors usually indicate a compiler bug, but may also occur as a result of inadequate recovery from previous errors. In any case we recommend to provide us with a bug report, including:

- version of the compiler
- description of your environment (OS, CPU)
- minimal set of files sufficient for reproducing the error

Example:

```
INTERNAL ERROR(ME): value expected
```

F450

```
compilation aborted: %s
```

An unrecoverable error occurred, causing immediate termination of the compiler. This message in most cases indicates a compiler bug. For instance,

```
compilation aborted: invalid location
```

means that the compiler has attempted to access memory at an address that is not valid.

Name	Meaning
OPTIMIZE	optimization domain: all classes or auto-detection
PROTECT	code protection: complete or partial
PACK	deployment mode: all, resource files only, or none
CLASSABSENCE	treatment of import of absent classes
COMPILERHEAP	heap limit of the compiler
DECOR	control of compiler messages
DLLNAME	resulting shared library name (<i>deprecated</i>)
CRYPTSEED	seed of data encryption key
ERRFMT	error message format
ERRLIM	maximum number of errors
EXPIRE	expiration date for creation of trials
EXPIREMSG	expiration message
IMAGEBASE	executable image base <i>Windows</i>
INLINELIMIT	maximum size of inlineable method
INLINETOLIMIT	maximum size of inlined-to method
JETVMPROP	hardwired system properties
JETRT	JET runtime flavor
LINK	linker command line
LOOKUP	file lookup paths
MAIN	application main class
OUTPUTDIR	target directory
OUTPUTNAME	resulting executable name
PRJ	project file name
PROJECT	project name
SAPROFMODE	Startup Accelerator profiling mode
SAPROFTIMEOUT	Startup Accelerator auto-shutdown timeout
SPLASH	splash screen image
SPLASHCLOSEONTITLE	hide splash when particular window opens
SPLASHMINTIME	minimum display time for splash
STACKLIMIT	generated program stack limit
STARTUPPROFILE	start up profile file
TEMPLATE	linker response file template

Table 16.2: JET compiler equations

Appendix A

Customizing the compiler

A.1 Customizing compiler messages

The file `jc.msg` contains texts of error messages in the form

```
number text
```

The following is an extract from `jc.msg`:

```
001 illegal character
002 comment not closed; started at line %d
...
042 incompatible assignment
...
```

Some messages contain format specifiers for additional arguments. In the above example, the message 002 contains a `%d` specifier used to print a line number.

To use a language other than English for compiler messages it is sufficient to translate `jc.msg`, preserving error numbers and the order of format specifiers.

A.2 Error message format specification

The format in which JET reports errors is user configurable through the **ERRFMT** equation. Its syntax is as follows:

```
{ string ", " [ argument ] "; " }
```

Any format specification allowed in the C procedure `printf` can be used in `string`.

Argument	Type	Meaning
line	integer	position in a source text
column	integer	position in a source text
file	string	name of a source file
module	string	class name
errmsg	string	message text
errno	integer	error code
mode	string	ERROR, FAULT, NOTICE, or WARNING
utility	string	name of an utility

Argument names are not case sensitive. By default, the error message format includes the following clauses:

" (%s", file;	—	a file name
" %d", line;	—	a line number
", %d", column;	—	a column number
") [% .1s] ", mode;	—	the first letter of an error mode
" %s\n", errmsg;	—	an error message

If a warning is reported for the file `test.mod` at line 5, column 6, the generated error message will look like this:

```
(test.mod 5,6) [W] variable declared but never used
```

A.3 Regular expression

The compiler recognizes filename patterns in the redirection file (see 16.1.5) and **LOOKUP** directives. Though in most cases you would use patterns like `*.class`, they are actually treated like regular expressions:

Sequence	Denotes
*	an arbitrary sequence of any characters, possibly empty (equivalent to <code>{\000-\377}</code> expression)
?	any single character (equivalent to <code>[\000-\377]</code> expression)
[. . .]	one of the listed characters
{ . . . }	an arbitrary sequence of the listed characters, possibly empty
\nnn	the ASCII character with octal code nnn, where n is [0-7]
&	the logical AND
	the logical OR
^	the logical NOT
(. . .)	the priority of operations

A sequence of the form `a-b` used within either `[]` or `{ }` brackets denotes all characters from `a` to `b`.

Examples

```
project.*
files which name is project with an arbitrary extension
```

```
*.obj|*.sym|*.bod
```

matches any file with extension `.obj`, `.sym` or `.bod`

```
{a-z}*X.class
```

matches files starting with any sequence of lowercase letters, ending in one final "X" and having the extension `.class`

A.4 Template files

A *template file*¹ is used to build a "makefile" in the PROJECT (see 16.4.2) operation mode.

The compiler copies lines from a template file into the output file verbatim, except lines marked with an exclamation mark ('!')

A template file is also subject to preprocessing (see 16.7).

A marked line (or template) has the following format²:

```
Template = "!" { Sentence }.
Sentence = Item { ", " Item } ";" | Iterator.
Item      = Atom | [ Atom | "^" ] "#" [ Extension ].
Atom      = String | EquationName.
String    = "'" { character } "'"
          | '"' { character } '".
Extension = [ ">" ] Atom.
Iterator  = "{" Set ":" { Sentence } "}".
Set       = { Keyword | String }
Keyword   = JAVA | CLASS | OBJ | MAIN.
```

EquationName is a name of a compiler or user-defined equation. Not more than three Items may be used in a Sentence. A first item in a sentence is a format string, while others are arguments.

The JET distribution contains a template file `jc.tem` which can be used to produce a linker response file.

A.4.1 Using equation values

In the simplest form, a template line may be used to output a value of an equation. For example, if the template file contains the line

```
! "The current project is %s.\n",prj;
```

and the project `prj/test.prj` is processed, the output will contain the line

```
The current project is prj/test.prj.
```

Note: the line

¹In the most, it is efficient to use default template file (`jc.tem`)

²The same syntax is used in the **LINK** equation.

```
! prj;
```

is valid, but may produce unexpected results under systems in which the backslash character ("\") is used as a directory names separator (e.g. Windows):

```
prj      est.prj
```

because "\t" in a format string is replaced with the tab character. Use the following form instead:

```
! "%s",prj;
```

A.4.2 File name construction

The "#" operator constructs a file name from a name and an extension, each specified as an equation name or literal string. A file is then searched for using lookup paths and the resulting name is substituted. For example, if the file `useful.lib` resides in the directory `'./mylibs'` and the redirection file contains the following line:

```
*.lib = /xds/lib;../mylibs
```

the line

```
! "useful"#"lib"
```

will produce

```
../mylibs/useful.lib
```

If the modifier ">" is specified, the compiler assumes that the file being constructed is an output file and creates its name according to the strategy for output files (See [16.1.5](#)).

The "#" operator is also used to represent the current value of an iterator (see [A.4.3](#)). The form in which a name or extension is omitted can be used in an iterator only.

The form "^#" may be used in a second level iterator to represent the current value of the first level iterator.

A.4.3 Iterators

Iterators are used to generate some text for all modules from a given set. Sentences inside the first level of braces are repeated for all modules of the project, while sentences inside the second level are repeated for all modules imported into the module currently iterated at the first level. A set is a sequence of keywords and strings. Each string denotes a specific module, while a keyword denotes all modules of specific kind.

The meaning of keywords is as follows:

Keyword	Meaning
CLASS	bytecode file
JAVA	Java source file
MAIN	Java main class (Java bytecode or source file with public static void main (String[]) method.)
OBJ	object file

A keyword not listed above is treated as filename extension. Sentences are repeated for all files with that extension which are explicitly specified in the project file using `!module` directives (see 16.8). This allows, for instance, additional libraries to be specified in a project file:

```
sample.prj:
    -template = mytem.tem
    !module Sample.class
    !module mylib.lib
```

```
mytem.tem:
    . . .
    ! "%s", "libxds"#"lib"
    ! { lib: "+%s", #; }
    ! "\n"
    . . .
```

Generated file:

```
. . .
d:\xds\lib\x86\libxds.lib+mylib.lib
. . .
```

A.4.4 Examples

Consider a sample project which consists of a main class A, which uses classes B and C; and B, in turn, uses D (all modules are bytecode files):

```

      A.class
      / \
     B.class C.class
      |
     D.class
```

The following examples illustrate template files usage:

This template line lists all project modules for which bytecode files are available:

```
! { main class: "%s ", #; }
```

For the sample project, it would generate the following line:

```
A.class B.class C.class D.class
```

To output main classes, the following line may be used:

```
! { main: "%s ", #; }
```

The output would be:

```
A.class
```

Index

- NAME+, 178
- NAME-, 178
- NAME :, 178
- NAME :=, 178
- NAME=, 178
- Java Runtime Slim-Down, 105
- jc, 176
- jc.cfg, 178
- jc.msg, 211
- jc.red, 176
- jc, 180
- advanced, 211
- CLASSABSENCE**, 136, 196, 210
- COMPILERHEAP**, 45, 93, 103, 171, 172, 179, 197, 210
- COMPILEWITHFIXEDHEAP**, 191, 191
- configuration, 175
 - configuration file, 178
 - directories, 176
 - redirection file, 176
 - search paths, 175
- configuration file
 - master, 179
- configuration file (jc.cfg), 178
- CRYPTSEED**, 49, 196, 210
- custom launcher, 149
- DECOR**, 197, 210
- DISABLECLASSSAVING**, 145, 191, 192
- DISABLESTACKTRACE**, 133, 191, 192
- DLLNAME**, 197, 199, 210
- equations, 195, 210
 - CLASSABSENCE**, 196
 - COMPILERHEAP**, 197
 - CRYPTSEED**, 196
 - DECOR**, 197
 - DLLNAME**, 197
 - ERRFMT**, 197
- ERRLIM**, 197
- EXPIRE**, 197
- EXPIREMSG**, 198
- IMAGEBASE**, 198
- INLINELIMIT**, 198
- INLINETOLIMIT**, 198
- JETRT**, 198
- JETVMPROP**, 198
- LINK**, 198
- LOOKUP**, 198
- MAIN**, 199
- OPTIMIZE**, 195
- OUTPUTDIR**, 199
- OUTPUTNAME**, 199
- PACK**, 196
- PRJ**, 200
- PROJECT**, 200
- PROTECT**, 196
- SAPROFMODE**, 200
- SAPROFTIMEOUT**, 200
- SPLASH**, 201
- SPLASHCLOSEONTITLE**, 201
- SPLASHMINTIME**, 201
- STACKLIMIT**, 201
- STARTUPPROFILE**, 201
- TEMPLATE**, 201
- ERRFMT**, 197, 210, 211
- ERRLIM**, 197, 210
- error message format, 211
- EXPIRE**, 129, 197, 210
- EXPIREMSG**, 130, 198, 210
- file name
 - portable notation, 175
- garbage collection, 121
- GENDLL**, 191, 192, 197, 199
- GENSTACKALLOC**, 117, 118, 182, 191, 192
- GENSTACKTRACE**, 133, 191, 193
- GLOBAL**, 191, 192

- Global Optimizer, 159
- GUI, 191, **193**
- IGNORECLASSDUPLICATION**, 136, 191, **193**
- IGNOREMEMBERABSENCE**, 136, 191, **193**, 202
- IMAGEBASE**, **198**, 210
- improving code performance, 116
- INLINE**, 117, 180, 191, **193**, 198
- INLINELIMIT**, 117, 194, **198**, 198, 210
- INLINETOLIMIT**, 117, 194, **198**, 198, 210
- JETRT**, 116, **198**, 210
- JETVMPROP**, 118, 123, 133, 157, 172, **198**, 210
- LINK**, **198**, 210, 213
- LOOKUP**, 176, 178, 186, 190, **198**, 210, 212
- MAIN**, **199**, 200, 202, 203, 210
- master configuration file, 179
- master redirection file, 176
- memory usage
 - application, 121
 - compiler, 179
- message
 - E349, 203
 - E350, 203
 - E351, 203
 - E352, 203
 - E370, 202
 - E371, 202
 - E374, 203
 - E375, 203
 - E623, 205
 - E955, 203
 - F020, 202
 - F190, 205
 - F191, 205
 - F192, 205
 - F194, 206
 - F195, 206
 - F402, 203
 - F424, 203
 - F425, 204
 - F448, 204
 - F450, 209
 - F475, 205
 - F480, 205
 - F603, 205
 - F638, 206
 - F950, 208
 - F951, 209
 - W300, 206
 - W301, 206
 - W303, 207
 - W310, 207
 - W311, 207
 - W312, 207
 - W314, 207
 - W315, 207
 - W318, 208
 - W620, 204
 - W621, 204
 - W622, 204
 - W640, 204
 - W641, 204
 - W642, 204
 - W643, 204
 - W900, 208
 - W901, 208
 - W902, 208
 - W903, 208
 - W918, 202
 - W919, 202
- MULTIAPP**, 125, 191, **194**
- NOFF**, 191, **194**
- operation modes, 180
 - ALL, 181
 - EQUATIONS, 182
 - HELP, 182
 - MAKE, 181
 - OPTIONS, 182
 - PROJECT, 181
- OPTIMIZE**, **195**, 210
- option precedence, 180
- options, 190, 191
 - COMPILEWITHFIXEDHEAP**, **191**
 - DISABLECLASSSAVING**, **192**
 - DISABLESTACKTRACE**, **192**
 - GENDLL**, **192**
 - GENSTACKALLOC**, **192**
 - GENSTACKTRACE**, **193**

- GLOBAL**, 192
- GUI**, 193
- IGNORECLASSDUPLICATION**, 193
- IGNOREMEMBERABSENCE**, 193
- INLINE**, 193
- MULTIAPP**, 194
- NOFF**, 194
- SPLASHCLOSEONAWTWINDOW**, 194
- SPLASHCLOSEONCLICK**, 194
- SPLASHGETFROMMANIFEST**, 195
- VERIFYALL**, 194
- WOFF**, 195
- OUTPUTDIR**, 199, 210
- OUTPUTNAME**, 197, 199, 199, 210
- PACK**, 135, 196, 210
 - precedence of options, 180
- PRJ**, 182, 200, 210
- PROJECT**, 200, 210
 - project files, 184
- PROTECT**, 48, 196, 210
- redirection file, 176
 - master, 176
- reflective shield, 142
- regular expressions, 212
- SAPROFMODE**, 168, 200, 210
- SAPROFTIMEOUT**, 168, 200, 210
- SPLASH**, 131, 195, 201, 210
- SPLASHCLOSEONAWTWINDOW**, 132, 191, 194, 201
- SPLASHCLOSEONCLICK**, 132, 191, 194
- SPLASHCLOSEONTITLE**, 132, 194, 201, 210
- SPLASHGETFROMMANIFEST**, 131, 191, 195
- SPLASHMINTIME**, 132, 201, 210
- STACKLIMIT**, 118, 173, 201, 210
- startup optimization, 165
- STARTUPPROFILE**, 168, 201, 210
- TEMPLATE**, 201, 210
 - template files, 213
- VERIFYALL**, 191, 194

This page had been intentionally left blank.



Excelsior, LLC

6 Lavrenteva Ave.

Novosibirsk 630090 Russia

Tel: +7 (383) 330-5508

Fax: +1 (509) 271-5205

Email: info@excelsior-usa.com

Web: <http://www.excelsior-usa.com>