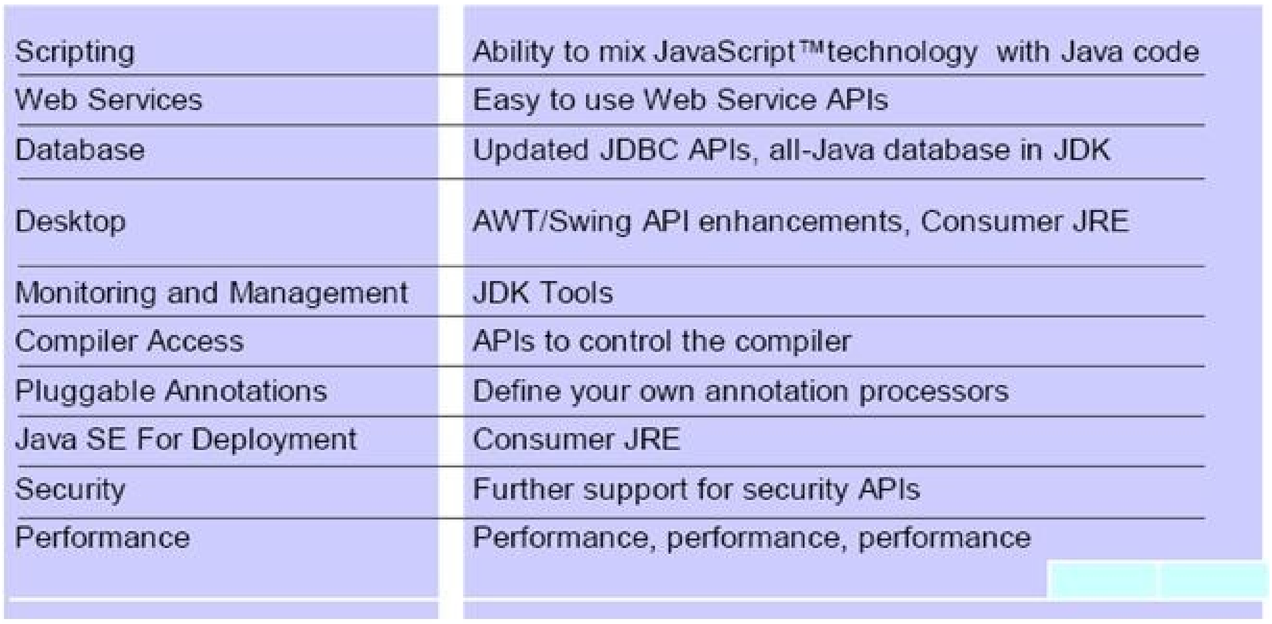
Java SE 6 Platform: Top 10 Features



Scripting	Ability to mix JavaScript™technology with Java code
Web Services	Easy to use Web Service APIs
Database	Updated JDBC APIs, all-Java database in JDK
Desktop	AWT/Swing API enhancements, Consumer JRE
Monitoring and Management	JDK Tools
Compiler Access	APIs to control the compiler
Pluggable Annotations	Define your own annotation processors
Java SE For Deployment	Consumer JRE
Security	Further support for security APIs
Performance	Performance, performance, performance

Scripting

Java 6 comes with built-in support for scripting languages. You can embed scripts in various scripting languages into your Java applications, passing parameters, evaluating expressions, and retrieving results. And you can do it all pretty seamlessly.

First of all, you obtain a new ScriptEngine object from a ScriptEngineManager, as shown here:

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("js");
```

Each scripting language has its own unique identifier. The "js" here means you're dealing with JavaScript.

You can assign scripting variables using the put() method and evaluate the script using the eval() method,. Which returns the most recently evaluated expression processed by the script.. Here's an example that puts it all together:

```
engine.put("cost", 1000);
String decision = (String) engine.eval(
"if ( cost >= 100){ " +
" decision = 'Ask the boss'; " +
"} else {" +
" decision = 'Buy it'; " +
"}");
assert ("Ask the boss".equals(decision));
```

You can do more than just pass variables to your scripts— you can also invoke Java classes from within your scripts. Using the importPackage() function enables you to import Java packages, as shown here:

```
engine.eval("importPackage(java.util); " +
"today = new Date(); " +
"print('Today is ' + today);");
```

Another feature is the Invocable interface, which lets you invoke a function by name within a script. This lets you write libraries in scripting languages, which you can use by calling key functions from your Java application. You just pass the name of the function you want to call, an array of Objects for the parameters, and you're done! Here's an example:

```
engine.eval(
"function calculateInsurancePremium(age) {...}");
Invocable invocable = (Invocable) engine;
Object result
= invocable.invokeFunction("calculateInsurancePremium",
new Object[] {37});
```

You actually can do a fair bit more than what I've shown here. For example, you can pass a Reader object to the eval() method, which makes it easy to store scripts in external files, or bind several Java objects to JavaScript variables using a Map-like Binding object. You can also compile some scripting languages to speed up processing. But you probably get the idea that the integration with Java is smooth and well thought-out.k

JDBC Enhancements

The major features added in JDBC 4.0 include:

1. Auto-loading of JDBC driver class
2. Connection management enhancements
3. Support for `RowId` SQL type
4. `DataSet` implementation of SQL using `Annotation`s
5. SQL exception handling enhancements
6. SQL XML support

There are also other features such as improved support for large objects (BLOB/CLOB) and National Character Set Support. These features are examined in detail in the following section.

Auto-Loading of JDBC Driver

In JDBC 4.0, we no longer need to explicitly load JDBC drivers using `Class.forName()`. When the method `getConnection` is called, the `DriverManager` will attempt to locate a suitable driver from among the JDBC drivers that were loaded at initialization and those loaded explicitly using the same class loader as the current application.

The `DriverManager` methods `getConnection` and `getDrivers` have been enhanced to support the Java SE Service Provider mechanism (SPM). According to SPM, a service is defined as a well-known set of interfaces and abstract classes, and a service provider is a specific implementation of a service. It also specifies that the service provider configuration files are stored in the `META-INF/services` directory. JDBC 4.0 drivers must include the file `META-INF/services/java.sql.Driver`. This file contains the name of the JDBC driver's implementation of `java.sql.Driver`. For example, to load the JDBC driver to connect to a Apache Derby database, the `META-INF/services/java.sql.Driver` file would contain the following entry:

```
org.apache.derby.jdbc.EmbeddedDriver
```

Let's take a quick look at how we can use this new feature to load a JDBC driver manager. The following listing shows the sample code that we typically use to load the JDBC driver. Let's assume that we need to connect to an Apache Derby database, since we will be using this in the sample application explained later in the article:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
Connection conn =
DriverManager.getConnection(jdbcUrl, jdbcUser, jdbcPassword);
```

But in JDBC 4.0, we don't need the `Class.forName()` line. We can simply call `getConnection()` to get the database connection.

Note that this is for getting a database connection in stand-alone mode. If you are using some type of database connection pool to manage connections, then the code would be different.

Connection Management

Prior to JDBC 4.0, we relied on the JDBC URL to define a data source connection. Now with JDBC 4.0, we can get a connection to any data source by simply supplying a set of parameters (such as host name and port number) to a standard connection factory mechanism. New methods were added to `Connection` and `Statement` interfaces to permit improved connection state tracking and greater flexibility when managing `Statement` objects in pool environments. The metadata facility (JSR-175) is used to manage the active connections. We can also get metadata information, such as the state of active connections, and can specify a connection as standard (`Connection`, in the case of stand-alone applications), pooled (`PooledConnection`), or even as a distributed connection (`XAConnection`) for XA transactions. Note that we don't use the `XAConnection` interface directly. It's used by the transaction manager inside a Java EE application server such as WebLogic, WebSphere, or JBoss.

RowId Support

The `RowID` interface was added to JDBC 4.0 to support the `ROWID` data type which is supported by databases such as Oracle and DB2. `RowId` is useful in cases where there are multiple records that don't have a unique identifier column and you need to store the query output in a `Collection` (such `Hashtable`) that doesn't allow duplicates. We can use `ResultSet`'s `getRowId()` method to get a `RowId` and `PreparedStatement`'s `setRowId()` method to use the `RowId` in a query.

An important thing to remember about the `RowId` object is that its value is not portable between data sources and should be considered as specific to the data source when using the `set` or `update` methods in `PreparedStatement` and `ResultSet` respectively. So, it shouldn't be shared between different `Connection` and `ResultSet` objects.

The method `getRowIdLifetime()` in `DatabaseMetaData` can be used to determine the lifetime validity of the `RowId` object. The return value or row id can have one of the values listed in Table 1.

RowId Value	Description
ROWID_UNSUPPORTED	Doesn't support `ROWID` data type.
ROWID_VALID_OTHER	Lifetime of the `RowID` is dependent on database vendor implementation.
ROWID_VALID_TRANSACTION	Lifetime of the `RowID` is within the current transaction as long as the row in the database table is not deleted.
ROWID_VALID_SESSION	Lifetime of the `RowID` is the duration of the current session as long as the row in the database table is not deleted.
ROWID_VALID_FOREVER	Lifetime of the `RowID` is unlimited as long as the row in the database table is not deleted.

Annotation-Based SQL Queries

The JDBC 4.0 specification leverages annotations (added in Java SE 5) to allow developers to associate a SQL query with a Java class without writing a lot of code to achieve this association. Also, by using the Generics (JSR 014) and metadata (JSR 175) APIs, we can associate the SQL queries with Java objects specifying query input and output parameters. We can also bind the query results to Java classes to speed the processing of query output. We don't need to write all the code we usually write to populate the query result into a Java object. There are two main annotations when specifying SQL queries in Java code: `Select` and `Update`.

Select Annotation

The `Select` annotation is used to specify a select query in a Java class for the `get` method to retrieve data from a database table. Table 2 shows various attributes of the `Select` annotation and their uses.

Name	Type	Description
sql	String	SQL Select query string.
value	String	Same as `sql` attribute.
tableName	String	Name of the database table against which the `sql` will be invoked.
readOnly, connected, scrollable	Boolean	Flags used to indicate if the returned `DataSet` is read-only or updateable, is connected to the back-end database, and is scrollable when used in `connected` mode respectively.
allColumnsMapped	Boolean	Flag to indicate if the column names in the `sql` annotation element are mapped 1-to-1 with the fields in the DataSet.

Here's an example of `Select` annotation to get all the active loans from the loan database:

```
interface LoanAppDetailsQuery extends BaseQuery {
@Select("SELECT * FROM LoanDetais where LoanStatus = 'A'")
DataSet<LoanApplication> getAllActiveLoans();
}
```

The `sql` annotation allows I/O parameters as well (a parameter marker is represented with a question mark followed by an integer). Here's an example of a parameterized `sql` query.

```
interface LoanAppDetailsQuery extends BaseQuery {
@Select(sql="SELECT * from LoanDetails
where borrowerFirstName= ?1 and borrowerLastName= ?2")
DataSet<LoanApplication> getLoanDetailsByBorrowerName(String borrFirstName,
String borrLastName);
}
```

Update Annotation

The `Update` annotation is used to decorate a `Query` interface method to update one or more records in a database table. An `Update` annotation must include a `sql` annotation type element. Here's an example of `Update` annotation:

```
interface LoanAppDetailsQuery extends BaseQuery {
```

```
@Update(sql="update LoanDetails set LoanStatus = ?1
where loanId = ?2")
boolean updateLoanStatus(String loanStatus, int loanId);
}
```

SQL Exception Handling Enhancements

Exception handling is an important part of Java programming, especially when connecting to or running a query against a back-end relational database. `SQLException` is the class that we have been using to indicate database related errors. JDBC 4.0 has several enhancements in `SQLException` handling. The following are some of the enhancements made in JDBC 4.0 release to provide a better developer's experience when dealing with `SQLException`s:

1. New `SQLException` sub-classes
2. Support for causal relationships
3. Support for enhanced for-each loop

New SQLException classes

The new subclasses of `SQLException` were created to provide a means for Java programmers to write more portable error-handling code. There are two new categories of `SQLException` introduced in JDBC 4.0:

- SQL non-transient exception
- SQL transient exception

Non-Transient Exception: This exception is thrown when a retry of the same JDBC operation would fail unless the cause of the `SQLException` is corrected. Table 3 shows the new exception classes that are added in JDBC 4.0 as subclasses of `SQLNonTransientException` (`SQLState` class values are defined in SQL 2003 specification.):

Exception class	SQLState value
`SQLFeatureNotSupportedException`	0A
`SQLNonTransientConnectionException`	08
`SQLDataException`	22
`SQLIntegrityConstraintViolationException`	23
`SQLInvalidAuthorizationException`	28
`SQLSyntaxErrorException`	42

Transient Exception: This exception is thrown when a previously failed JDBC operation might be able to succeed when the operation is retried without any intervention by application-level functionality. The new exceptions extending `SQLTransientException` are listed in Table 4.

Exception class	SQLState value
`SQLTransientConnectionException`	08
`SQLTransactionRollbackException`	40
`SQLTimeoutException`	None

Causal Relationships

The `SQLException` class now supports the Java SE chained exception mechanism (also known as the Cause facility), which gives us the ability to handle multiple `SQLException`s (if the back-end database supports a multiple exceptions feature) thrown in a JDBC operation. This scenario occurs when executing a statement that may throw more than one `SQLException` .

We can use `getNextException()` method in `SQLException` to iterate through the exception chain. Here's some sample code to process `SQLException` causal relationships:

```
catch(SQLException ex) {
while(ex != null) {
LOG.error("SQL State:" + ex.getSQLState());
LOG.error("Error Code:" + ex.getErrorCode());
LOG.error("Message:" + ex.getMessage());
Throwable t = ex.getCause();
while(t != null) {
LOG.error("Cause:" + t);
t = t.getCause();
}
ex = ex.getNextException();
}
}
```

Enhanced For-Each Loop

The `SQLException` class implements the `Iterable` interface, providing support for the for-each loop feature added in Java SE 5. The navigation of the loop will walk through `SQLException` and its cause. Here's a code snippet showing the enhanced for-each loop feature added in `SQLException`.

```
catch(SQLException ex) {
for(Throwable e : ex ) {
LOG.error("Error occurred: " + e);
}
}
```

JDBC 4.0 API: New Classes

RowId (java.sql)

As described earlier, this interface is a representation of an `SQL ROWID` value in the database. `ROWID` is a built-in SQL data type that is used to identify a specific data row in a database table. `ROWID` is often used in queries that return rows from a table where the output rows don't have an unique ID column.

Methods in `CallableStatement`, `PreparedStatement`, and `ResultSet` interfaces such as `getRowId` and `setRowId` allow a programmer to access a `SQL ROWID` value. The `RowId` interface also provides a method (called `getBytes()`) to return the value of `ROWID` as a byte array. `DatabaseMetaData` interface has a new method called `getRowIdLifetime` that can be used to determine the lifetime of a `RowId` object. A `RowId`'s scope can be one of three types:

1. Duration of the database transaction in which the `RowId` was created
2. Duration of the session in which the `RowId` was created
3. The identified row in the database table, as long as it is not deleted

DataSet (java.sql)

The `DataSet` interface provides a type-safe view of the data returned from executing of a SQL Query. `DataSet` can operate in a connected or disconnected mode. It is similar to `ResultSet` in its functionality when used in connected mode. A `DataSet`, in a disconnected mode, functions similar to a `CachedRowSet`. Since `DataSet` extends `List` interface, we can iterate through the rows returned from a query.

There are also several new methods added in the existing classes such as `Connection` (`createSQLXML`, `isValid`) and `ResultSet` (`getRowId`).

SQL/XML

A large amount of data now exists in the XML format. Databases have extended support for the XML data type by defining a standard XML type in the SQL 2003 specification. Most database vendors have an implementation of the XML data type in their new releases. With the inclusion of such a type, an XML dataset or document could be one of the fields or column values in a row of a database table. Prior to JDBC 4.0, perhaps the best way to manipulate such data within the JDBC framework is to use proprietary extensions from the driver vendors or access it as a `CLOB` type.

JDBC 4.0 now defines `SQLXML` as the Java data type that maps the database SQL XML type. The API supports processing of an XML type as a string or as a StAX stream. Streaming API for XML, which for Java has been adopted via JSR 173, is based on the Iterator pattern, as opposed to the Simple API for XML Processing (SAX), which is based on the Observer pattern.

Invoking the `Connection` object's `createSQLXML()` method can create a `SQLXML` object. This is an empty object, so the data can be attached to it by either using the `setString()` method or by associating an XML stream using the `createXMLStreamWriter()` method with the object. Similarly, XML data can be retrieved from a `SQLXML` object using `getString()` or associating an XML stream using `createXMLStreamReader()` with the object.

The `ResultSet`, the `PreparedStatement`, and the `CallableStatement` interfaces have `getSQLXML()` methods for retrieving a `SQLXML` data type. `PreparedStatement` and `CallableStatement` also have `setSQLXML()` methods to add `SQLXML` objects as parameters.

The `SQLXML` resources can be released by calling their `free()` methods, which might prove pertinent where the objects are valid in long-running transactions. `DatabaseMetaData`'s `getTypeInfo()` method can be called on a datasource to check if the database supports the `SQLXML` data type, since this method returns all the data types it supports.

Desktop Features and Improvements

Java has long been regarded as an excellent language for server-based software, but as second-rate for desktop GUI applications. The Java desktop team at Sun has been working to change this perception by integrating Java more closely with the host system it runs on. The result is not only improved GUI performance in Java SE 6, but also improvements in the behavior of Java GUI applications.

Many of the new desktop features in Java SE 6 are based on the JDesktop Integration Components (JDIC) project. The JDIC project gives Java applications access to features available on the native OS desktop, such as the browser, email editor, file-type associations, the system tray, application launching, and printing. The following are some of the more prominent Java desktop improvements in Java SE 6:

- *Splash screen support*—Splash screens inform a user that an application is starting while he waits. Java SE 6 adds support for splash screens that can be displayed even before the JVM starts.
- *Java Foundation Classes (JFC) and Swing Improvements*:
  - Java SE 6 leverages Windows APIs to both improve performance and ensure the Windows look-and-feel on current and future versions of Windows.
  - It improves layout management to include customizable layout managers and includes other enhancements to simplify GUI component layout.
  - It has vastly improved Swing drag-and-drop, making it customizable.
  - True double-buffering provides quick, smooth graphic transitions.
- *System Tray Support*—Two new classes, *SystemTray* and *TrayIcon*, in the *java.awt* package allow you to add icons, tool tips, and pop-up menus to the Windows or Gnome Linux system tray. The system tray is the desktop area shared by all applications, usually located in the lower-right corner. Actions and events allow your Java application to track mouse clicks on the items you place in the tray, and respond to those clicks. I have found this feature useful for my server applications as well. For instance, used with the Desktop API (see below), I now add an icon to the system tray to easily launch a browser for the application's administrative HTML page. Regardless of OS (Linux or Windows), I no longer need to recall the application's administrative port or URL—simply click the icon and the page appears.
- *Improved print support for JTable*
- *Java 2D enhancements*—Improvements have been made to text display quality, especially on LCD monitors. Integration with the host desktop's font anti-aliasing settings ensures consistent text

rendering.

- *The new java.awt.Desktop API*—The new Java SE 6 Desktop package aims to make Java UI applications "first-class citizens." With this package, Java applications can launch the default browser and email client, and integrate with common desktop applications (such as OpenOffice) to open, edit, and print files of specific types. The Desktop package offers this ability through action events (Desktop.Action) that you can integrate into your applications.
- *Internationalization*—Java SE 6 supports "plugability" for some locale-specific features, such as date formatting, Unicode text normalization, and resource bundles.

Monitoring and Management

Java 6 has made some good progress in the field of application monitoring, management, and debugging. The Java Monitoring and Management Console, or JConsole, is well known to system administrators who deploy Java applications (see Figure 1). First appearing in Java 5, JConsole is a graphical monitoring tool based on JMX, which provides a very complete view of the performance and resource consumption of Java applications. Java 6 enhances JConsole in several ways, making it both easier to use and more powerful. The graphical look has been improved; you can now monitor several applications in the same JConsole instance, and the summary screen (shown in Figure 1) has been redesigned. It now displays a graphical dashboard of the key statistics. You also can now export any graph data in CSV form for further analysis in a spreadsheet.

One great thing about application monitoring in Java 6 is that you don't need to do anything special to your application to use it. In Java 5, you needed to start any application that you wanted to monitor with a special command-line option (-Dcom.sun.management.jmxremote). In Java 6, you can monitor any application that is running in a Java 6 VM.

Java 6 comes with sophisticated thread-management and monitoring features as well. The Java 6 VM can now monitor applications for deadlocked threads involving object monitors and java.util.concurrent ownable synchronizers. If your application seems to be hanging, JConsole lets you check for deadlocks by clicking on the Detect Deadlock button in the Threads tab.

At a lower level, Java 6 helps resolve a common hard-to-isolate problem: the dreaded java.lang.OutOfMemoryError. In Java 6, an OutOfMemoryError will not just leave you guessing; it will print out a full stack trace so that you can have some idea of what might have caused the problem.

Managing the File System

Java 6 gives you much finer control over your local file system. For example, it is now easy to find out how much free space is left on your hard drive. The java.io.File class has the following three new methods to determine the amount of space available (in bytes) on a given disk partition:

```
File homeDir = new File("/home/john");
System.out.println("Total space = " + homeDir.getTotalSpace());
System.out.println("Free space = " + homeDir.getFreeSpace());
        System.out.println("Usable space = " + homeDir.getUsableSpace());
```

As their names would indicate, these methods return the total amount of disk space on the partition (getTotalSpace()), the amount of currently unallocated space (getFreeSpace()), and the amount of available space (getUsableSpace()), after taking into consideration OS-specific factors such as write permissions or other operating-system constraints. According to the documentation, getUsableSpace() is more accurate than getFreeSpace().

When I ran this code on my machine, it produced the following output:

```
Total space = 117050585088
Free space = 100983394304
Usuable space = 94941515776
```

File permissions are another area where Java 6 brings new enhancements. The java.io.File class now has a set of functions allowing you to set the readable, writable, and executable flags on files in your local file system, as you would with the Unix chmod command. For example, to set read-only access to a file, you could do the following:

```
File document = new File("document");
documentsDir.setReadable (true);
documentsDir.setWritable(false);
documentsDir.setExecutable (false);
```

This will set read-only access for the owner of the file. You can also modify the access rights for all users by setting the second parameter (ownerOnly) to false:

```
documentsDir.setReadable (true, false);
documentsDir.setWritable(false, false);
documentsDir.setExecutable (false, false);
```

Naturally, this will work only if the underlying operating system supports this level of file permissions.

Compiler API

All the API (the client interfaces and the classes) needed by the developers for playing with the Java compiler API is available in the new javax.tools package. Not only this package represents classes and methods for invoking a java compiler, but it provides a common interface for representing any kind of Tool. A tool is generally a command line program (like javac.exe, javadoc.exe or javah.exe).

Instead of looking into all the classes and interfaces that are available in the javax.tools package, it makes more sense to go through some sample programs, then examining what the classes and the methods are doing.

Compiling a java source file from another Java source

Following is the small sample program that will demonstrate how to compile a Java source file from another java file on the fly.

[All the examples given here are written and tested with Mustang build 1.6.0-b105, and it seems that more API changes and restructuring of classes and methods are occurring in the newer builds].

```
Listing for MyClass.java file that is going to get compiled.

MyClass.java:

package test;
public class MyClass {
public void myMethod(){
System.out.println("My Method Called");
}
}

Listing for SimpleCompileTest.java that compiles the MyClass.java file.

SimpleCompileTest.java:

package test;
import javax.tools.*;
public class SimpleCompileTest {
public static void main(String[] args) {
String fileToCompile = "test" + java.io.File.separator +"MyClass.java";
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
int compilationResult = compiler.run(null, null, null, fileToCompile);
if(compilationResult == 0){
System.out.println("Compilation is successful");
}else{
System.out.println("Compilation Failed");
}
}
}
```

The entry point for getting a compiler instance is to depend on the ToolProvider class. This class provides methods for locating a Tool object. (Remember a Tool can be anything like javac, javadoc, rmic, javah etc...) Though the only Tool available in Mustang build (1.6.0-b105) is the JavaCompiler as of now, it is expected that many more tools are to be added in the future.

The getSystemJavaCompiler() method in the ToolProvider class returns an object of some class that implements JavaCompiler (JavaCompiler is an interface that extends Tool interface and not a class). To be more specific, the method returns a JavaCompiler implementation that is shipped along with the Mustang Distribution. The implementation of the Java Compiler is available in the tools.jar file (which is usually available in the <JDK60_INSTALLATION_DIR>\lib\tools.jar).

After getting an instance of the JavaCompiler, compilation on a set of files (also known as compilation units) can be done by invoking the run(InputStream inputStream, OutputStream outputStream, OutputStream errorStream, String … arguments) method. To use the defaults, null can be passed for the first three parameters (which correspond to System.in, System.out and System.err), the fourth parameter which cannot be null is a variable argument that refers to the command-line arguments we usually pass to the javac compiler.

The file that we are going to compile is MyClass.java which is the same test package as of the SimpleCompileTest. The complete file name (along with the directory 'test') is passed as 4th argument to the run method. If there are no errors in the source file (MyClass.java, in this case), the method will return 0 which means that the source file was compiled successfully.

After compiling and running the SimpleCompileTest.java, one can see the following output message in the console.

'Compilation is successful'

Let us modify the source code by introducing a small error by removing the semi-colon after the end of the println() statement and see what happens to the output.

```
MyClass.java:
package test;

public class MyClass {
public void myMethod(){
System.out.println("My Method Called")
// Semi-colon removed here purposefully.
}
}
Now, running the SimpleCompileTest.java leads to the following output,

test\MyClass.java:5: ';' expected
System.out.println("My Method Called")
^
1 error
Compilation Failed
```

This is the error message one normally sees when compiling a java file using javac.exe in the command prompt. The above represents the error message(s) and since we have made the error output stream point to null (which defaults to System.err, which is the console) we are getting the output error messages in the console. If instead we have pointed the errorStream to something like this,

```
FileOutputStream errorStream = new FileOutputStream("Errors.txt");
int compilationResult = compiler.run(null, null, errorStream, fileToCompile);
```

a new file called Errors.txt will be created in the current directory and the file would be populated with the error messages that we saw before.

Compiling multiple files

One might be tempted to think that the following code will work for compiling multiple java files (assuming that the two files that are to be compiled are One.java and Two.java).

```
String filesToCompile = new String("One.java Two.java") ;
```

But surprisingly, when you try this, you will get a 'Compilation Failed' error message in the console.

The answer is JavaCompiler needs to extract individual options and arguments and these options and arguments should not be separated by spaces, but should be given as individual strings.

So, this won't work at all.

```
compiler.run(null, null, null, "One.java Two.java");
```

But, the below code will work nicely.

```
compiler.run(null, null, null, "One.java", "Two.java");
```

One reason for forcing this kind of restriction is that sometimes the complete java file names (which includes directory path as well) itself can have white-spaces , in such a case it would be difficult for the parser to parse all the tokens correctly into options and arguments.

Following is a sample code that compiles multiple java files.

```
MyClass.java:
package test;

public class MyClass {
}


MyAnotherClass.java:
package test;

public class MyAnotherClass {
}


MultipleFilesCompileTest.java:
package test;
import javax.tools.*;

public class MultipleFilesCompileTest {
public static void main(String[] args) throws Exception{
String file1ToCompile = "test" + java.io.File.separator + "MyClass.java";
String file2ToCompile = "test" + java.io.File.separator + "MyAnotherClass.java";
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
int compilationResult = compiler.run(null, null, null, file1ToCompile, file2ToCompile);
if(compilationResult == 0){
System.out.println("Compilation is successful");
}else{
System.out.println("Compilation Failed");
}
}
}
```

The above program compiles fine with the output message ‘Compilation is successful’.

Do remember, that the final argument is a variable argument and it can accept any number of arguments.

[Starting with Java 5.0, variable argument is a new feature where the callers (the calling method) can pass any number of arguments. To illustrate this concept, look at the sample code.

```
public int addNumbers(int …numbers){

int total = 0;
For(int temp : numbers){
Total = total + temp;
}
return total;
}
```

A variable argument is represented by ellipsis (…) preceding the variable name like this int … numbers. And , one can call the above method in different styles, like the below

```
addNumbers(10, 10, 30,40); // This will work.
addNumbers(10,10) // This also will work.
```

type must be the last one. Variable arguments are internally treated as arrays. So, this is also possible now.

```
addNumbers(new int[]{10, 34, 54});
```

So, great care should be exercised when passing multiple options along with values to the run method. As an example, the ideal way to pass options along with its option values might me,

```
compiler.run(null, null, null, ”-classpath”, ”PathToClasses”, ”-sourcepath”,
”PathToSources”, ”One.java”, ”Two.java”);
```

Ass one can see, even the option and its option values must be treated as a separate string.

Compiling Files with the ‘verbose’ option

As you are aware of the fact, when invoking the java compiler with the ‘verbose’ option specified, the javac will output messages that will occur during the compilation life-cycle (like parsing the input, validating them, scanning for the paths for both source and class files, loading all the necessary class files, then finally creating the class files in the specified destination directory). Let us achieve the same effect through the following sample code.

```
SimpleCompileTestWithVerboseOption.java
package test;
import java.io.FileOutputStream;
import javax.tools.*;
public class SimpleCompileTestWithVerboseOption {
public static void main(String[] args) throws Exception{
String fileToCompile = "test" + java.io.File.separator + "MyClass.java";
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
FileOutputStream errorStream = new FileOutputStream("Errors.txt");
int compilationResult = compiler.run(
null, null, errorStream, "-verbose", fileToCompile);
if(compilationResult == 0){
System.out.println("Compilation is successful");
}else{
System.out.println("Compilation Failed");
}
}
}
```

May be a bug in Mustang

One might see that the errorStream (3rd argument) has been pointed out to a file to collect the output messages instead of the outputStream (2nd argument).

The following code was tried for capturing the verbose output, but this code failed, in the sense, the output was still written to the standard console, though the code seems to re-direct the output to a file.

```
FileOutputStream outputStream = new FileOutputStream("Output.txt");
int compilationResult = compiler.run(
null, outputStream, null, "-verbose", fileToCompile);
```

The Java Compiler API is treating the output messages (in this case, the output messages obtained by specifying the ‘verbose’ option) as error messages, and so even though the output is pointing to the file (‘Output.txt’), it is spitting all the output messages to the console.

Also, the documentation for the run method is unclear; it tells that any diagnostics (errors, warnings or information) may be written either to the output stream or the error messages.

Advanced Compilation

In the above section, we saw how to compile files using the JavaCompiler tool. For advanced compilation related stuffs, the JavaCompiler depends on two more services namely the file manager services and the diagnostics services. These services are provided by the JavaFileManager and the DiagnosticListener classes respectively.

JavaFileManager

The JavaFileManager (being given implementation as StandardJavaFileManager) manages all the file objects that are usually associated with tools. This JavaFileManager is not only to JavaCompiler, but instead it can work with any kinds of objects that conform to the standard Tool interface. To understand why JavaFileManager is so important, let us understand what could be the things that may happen during compilation process.

```
javac MyClass.java
```

When we issue this command in the command prompt, so many things will happen. The very first thing is that the compiler will parse all the options that are specified by the user, have to validate them, and them have to scan the source and class path for java source files and the jar files. It then has to deal with the input files (in this case it is MyClass.java) and output files (MyClass.class).

So, JavaFileManager which is associated with any kind of tool (normally all tools have some kind of input and the output files for processing), deals with managing all the input files and output files. By managing, we mean that JavaFileManager is responsible for creating output files, scanning for the input files, caching them for better performance. One such implementation given to the JavaFileManager is the StandardJavaFileManager.

A file being managed by JavaFileManager doesn't mean the file is necessarily a file in the hard-disk. The contents of the file managed may come from a physical file in a hard-disk, may come from in-memory or can even come from a remote socket. That's why JavaFileManager doesn't deal with java.io.File objects (which usually refer to the physical files in the operating system File System). Rather, JavaFileManager manages all the files and their contents in the form of FileObject and JavaFileObject which represents the abstract representation for any kind of files by managed by the JavaFileManager.

FileObject and JavaFileObject refer to the abstract file representation that are being managed by the JavaFileManager and also have support related to reading and writing the contents to the right destination. The only difference between a FileObject and a JavaFileObject is that a FileObject can represent any kind of FileObject (like text file, properties file, image file etc), whereas a JavaFileObject can represent only java source file (.java) or a class file (.class). If a FileObject represents a java source file or a class file, then implementations should take care to return a JavaFileObject instead.

Diagnostics

The second dependent service by the JavaCompiler is the Diagnostic Service. Diagnostic usually refers to errors, warnings or informative messages that may appear in a program. For getting diagnostics messages during the compilation process, we can attach a listener to the compiler object. The listener is a DiagnosticListener object and its report() method will be called with a diagnostic method which contains many a more information like the kind of diagnostics (error, warning, information or other), the source for this diagnostics, the line number in the source code, a descriptive message etc.

CompilationTask

Before going into the sample code to clarify things like JavaFileManager, Diagnostic,and the DiagnosticListener classes, let us have a quick review on the class CompilationTask. From its name, obviously one can tell that it represents an object that encapsulates the actual compilation operation. We can initiate the compilation operation by calling the call() method in the CompilationTask object.

But how to get the CompilationTask object?

Since, CompilationTask is closely associated to a JavaCompiler object, one can easily say JavaCompiler.getCompilationTask(arguments) to get the CompilationTask object. Let we now see the parameters that are needed to pass to the getCompilationTask(…) method.

Almost all the arguments can be null (with their defaults, which is discussed later), but the final argument represents the list of java objects to be compiled (or the compilation units) which cannot be null. The last argument is Iterable<? Extends JavaFileObject> javaObjects.

So, how can one populate this argument?

[Iterable is a new interface that was added with jdk 5.0 which is use to iterate (or traverse over a collection of objects. It has one method called iterator() which returns an Iterator object, and using it one can traverse over the collection by using the combinational hasNext() and the next() methods.

Considering type safety which started from Java 5.0, it is the role of the developer to specify what exactly is the type of object to be iterated. Iterable<? Extends JavaFileObject> means that this argument is an iterable that is acting on any class that implement the JavaFileObject interface].

JavaFileManager has 4 convenience methods like getJavaFileObject(File … files) , getJavaFileObjects(String … filenames), getJavaFileObjectsFromFiles(Iterable(? extends JavaFileObject) and getJavaFileObjectFromString(Iterable<? extends String> filenames) that returns javaObjects in the form of Iterable<? Extends JavaFileObject> type.

So, we can construct the 4th arguments using any of the convenience methods.

With more bits of theory in the last few sections, let us see a sample program that incorporates all the classes and the methods that we saw listed.

Purposefully we create a java file called MoreErrors.java that has some error code in it. The listing for MoreErrors.java is shown below.

```
MoreErrors.java:
package test;
```

```
public class MoreErrors {
public void errorOne ()
// No open braces here. Line a
}
public void errorTwo(){
System.out.println("No semicolon") // Line b
// No semicolon at the end of the statement.
}
public void errorThree(){
System.out.prntln("No method name called prntln()"); // Line c
}
}
```

As one can see, the statement above line a, line b and line c has errors.

Let us look into the AdvancedCompilationTest.java that uses all the classes and the methods that we discussed above.

```
AdvancedCompilationTest.java:
package test;
import java.io.*;
import java.util.*;
import javax.tools.*;
import javax.tools.JavaCompiler.*;
public class AdvancedCompilationTest {
public static void main(String[] args) throws Exception {
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
// Line 1.
MyDiagnosticListener listener = new MyDiagnosticListener(); // Line 2.
StandardJavaFileManager fileManager =
compiler.getStandardFileManager(listener, null, null); // Line 3.
String fileToCompile = "test" + File.separator + "ManyErrors.java";
// Line 4
Iterable<? extends JavaFileObject> fileObjects =
fileManager.getJavaFileObjectsFromStrings(
Arrays.asList(fileToCompile)); // Line 5
CompilationTask task = compiler.getTask(null, fileManager, listener, null, null,
fileObjects); // Line 6
Boolean result = task.call(); // Line 7
if(result == true){
System.out.println("Compilation has succeeded");
}
}
}
class MyDiagnosticListener implements DiagnosticListener<JavaFileObject>{
public void report(Diagnostic<? extends JavaFileObject> diagnostic) {
System.out.println("Code->" + diagnostic.getCode());
System.out.println("Column Number->" + diagnostic.getColumnNumber());
System.out.println("End Position->" + diagnostic.getEndPosition());
System.out.println("Kind->" + diagnostic.getKind());
System.out.println("Line Number->" + diagnostic.getLineNumber());
System.out.println("Message->"+ diagnostic.getMessage(Locale.ENGLISH));
System.out.println("Position->" + diagnostic.getPosition());
System.out.println("Source" + diagnostic.getSource());
System.out.println("Start Position->" + diagnostic.getStartPosition());
System.out.println("\n");
}
}
```

Let us explore the above code in greater detail.

Line 1 is essentially creating an object of type JavaCompiler using the ToolProvider class. This is the entry point.

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler()
```

Line 2 and Line 3 is making the compiler to make use of the FileManager and the Diagnostic's Services. To rewind the theory a bit, JavaFileManager object is used to manage the input and the output files that a tool normally deals with. And Diagnostic's objects refer the diagnostics (errors, warnings or information) that may occur during the compilation process within a program.

```
MyDiagnosticListener listener = new MyDiagnosticListener();
```

Line 2 creates an object of Type DiagnosticListener, since we want to monitor the diagnostics that may happen during the process of compilation (diagnostics, in the form of error messages will always happen in our case, as we have purposefully done some errors in the java code). We have overridden the report(Diagnostic) method and have extracted all the possible information. Since diagnostics can happen in any kind of file object, how can we specifically tell that this Diagnostics is for a java file object?

The answer has become easy because of Java 5.0 generics. One can notice that MyDiagnosticListener is a typed class (having some typed parameter) meaning that it can act on any kind of object that has diagnostics properties; here we are explicitly telling that the diagnostics is for JavaFileObject and not for any other file object by mentioning the JavaFileObject in the class declaration and the method declaration (shown in bold).

```
class MyDiagnosticListener implements DiagnosticListener<JavaFileObject>{
public void report(Diagnostic<? extends JavaFileObject> diagnostic) {
System.out.println("Code->" + diagnostic.getCode());
System.out.println("Column Number->" + diagnostic.getColumnNumber());
....
....
}
}
```

In Line 3, we are associating the Diagnostics listener object to the compiler object through the standard java file manager by making this method call.

```
StandardJavaFileManager fileManager =
compiler.getStandardFileManager(listener, null, null);
```

This method tells to attach the diagnostics listener object to this compiler object, so whenever this compiler object executes a compilation operation, and if any diagnostics related errors or warnings have occurred in a program, then the diagnostics being encapsulated by the Diagnostic object will be passed back to the report() method of the DiagnosticListener interface.

The last 2 arguments refer the locale and charset arguments for formatting the diagnostic messages in a particular locale and by using the specified charset which can be null.

Line 4 and Line 5 populates the file objects to be compiled by using the convenience objects in the StandardJavaFileManager class.

```
String fileToCompile = "test" + File.separator + "ManyErrors.java";
Iterable<? extends JavaFileObject> fileObjects =
fileManager.getJavaFileObjectsFromStrings(
Arrays.asList(fileToCompile));
```

Line 6 gets an instance of the CompilationTask object by calling the getCompilationTask() method and by passing the fileManager, listener and the fileObjects objects. The null arguments refer to the Writer object (for getting the output from the compiler), list of options (the options that we pass to javac like –classpath classes, -sourcepath Sources…) and classes (for processing the custom annotations that are found in the source code).

Finally, the actual compilation operation is done by calling the call() method which returns true if all the files (compilationUnits) succeed compilation. If any of the files have errors in it , then the call() method will return false. Anyway, in our case, we have some error code in the MoreErrors.java file, so the report method will be called printing all the diagnostics information.

DiagnosticCollector

In the previous program, we saw that we have a customized class called MyDiagnosticListener. Its sole purpose it to collect and print all the diagnostic messages to the console. This class can be completely eliminated since Mustang already has a class called DiagnosticCollection that does the same thing. It has a method called getDiagnostics() which returns a list , through which we can iterate and can output the diagnostic messages to the console.

The following code achieves the same using DiagosticCollector class.

```
AdvancedCompilationTest2.java:
package test;
import java.io.*;
import java.util.*;
import javax.tools.*;
import javax.tools.JavaCompiler.*;
public class AdvancedCompilationTest2 {
public static void main(String[] args) throws Exception {
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler(); // Line 1.
DiagnosticCollector<JavaFileObject> diagnosticsCollector =
new DiagnosticCollector<JavaFileObject>();
StandardJavaFileManager fileManager =
compiler.getStandardFileManager(diagnosticsCollector, null, null); // Line 3.
String fileToCompile = "test" + File.separator + "ManyErrors.java"; // Line 4
Iterable<? extends JavaFileObject> fileObjects =
fileManager.getJavaFileObjectsFromStrings(Arrays.asList(fileToCompile)); // Line 5
CompilationTask task = compiler.getTask(null, fileManager, diagnosticsCollector, null,
null, fileObjects); // Line 6
Boolean result = task.call(); // Line 7
List<Diagnostic<? extends JavaFileObject>> diagnostics =
diagnosticsCollector.getDiagnostics();
for(Diagnostic<? extends JavaFileObject> d : diagnostics){
// Print all the information here.
}
if(result == true){
System.out.println("Compilation has succeeded");
}else{
System.out.println("Compilation fails.");
}
}
}
```

Compilation of Java Source from a String object

Having discussed about the various ways of compiling java file sources, it's now time to look at how to compile a java source that is encapsulated in a string object. As previously mentioned, it is not mandatory that the contents of a java source must reside in hard-disk, it can even reside in memory. By saying that compiling a java source from a string object, we are implicitly saying that the java source is residing in memory, more specifically, the contents are residing in RAM.

For this to happen, we have to encapsulate a class the represents the java source from a string. We can extend this class from the SimpleJavaFileObject (a convenient class that overrides all the methods in the JavaFileObject with some default implementation). The only method to override is the getCharContent() that will be called internally by the Java compiler to get the java source contents.

```
JavaObjectFromString.java:
package test;
import java.net.URI;
class JavaObjectFromString extends SimpleJavaFileObject{
private String contents = null;
public JavaObjectFromString(String className, String contents) throws Exception{
super(new URI(className), Kind.SOURCE);
this.contents = contents;
}
public CharSequence getCharContent(boolean ignoreEncodingErrors) throws IOException {
return contents;
}
}
```

Since the SimpleJavaFileObject has a protected two argument constructor that accepts an URI (the URI representation of the file object) and a Kind object (a special type that tells what kind is this object, the Kind may be a Kind.SOURCE, Kind.CLASS, Kind.HTML, Kind.OTHER, in our case, it is Kind.SOURCE, since we are inferring a Java source object), we have defined a two argument constructor in JavaObjectFromString class and delegates the control back the base class. The getCharContent() method has to be overridden (since this is the method that will be called by the JavaCompiler to get the actual java source contents) to return the string (remember, String implements CharSequence) that represents the entire java source (which was previously saved in the constructor).

The code that uses this JavaObjectFromString object looks like this.

```
AdvancedCompilationTest3.java:
package test;
import java.io.*;
import java.util.*;
import javax.tools.*;
import javax.tools.JavaCompiler.*;
public class AdvancedCompilationTest3 {
public static void main(String[] args) throws Exception {
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
DiagnosticCollector<JavaFileObject> diagnosticsCollector =
new DiagnosticCollector<JavaFileObject>();
StandardJavaFileManager fileManager =
compiler.getStandardFileManager(diagnosticsCollector, null, null);
JavaFileObject javaObjectFromString = getJavaFileContentsAsString();
Iterable<? extends JavaFileObject> fileObjects = Arrays.asList(javaObjectFromString);
CompilationTask task = compiler.getTask(null, fileManager, diagnosticsCollector, null,
null, fileObjects);
Boolean result = task.call();
List<Diagnostic<? extends JavaFileObject>> diagnostics =
diagnosticsCollector.getDiagnostics();
for(Diagnostic<? extends JavaFileObject> d : diagnostics){
// Print all the information here.
}
if(result == true){
System.out.println("Compilation has succeeded");
}else{
System.out.println("Compilation fails.");
}
}
static SimpleJavaFileObject getJavaFileContentsAsString(){
StringBuilder javaFileContents = new StringBuilder("" +
"class TestClass{" +
" public void testMethod(){" +
" System.out.println(" + "\"test\"" + ");" +
"}" +
"}");
JavaObjectFromString javaFileObject = null;
try{
javaFileObject = new JavaObjectFromString("TestClass", javaFileContents.toString());
}catch(Exception exception){
exception.printStackTrace();
}
return javaFileObject;
}
}
```

Pluggable Annotation Processing API

The first part of this article listed out the major new features of ***Java 6 (Mustang)*** related to areas like ***Common Annotations (JSR 250)***, ***Scripting Language for the Java Platform (JSR 223)*** and ***JDBC 4.0***. This article assumed that Readers have got sufficiently fair bit of knowledge in the various concepts of Java 5.0. First-time Readers of Java 6 are strongly encouraged to read the first part of this article titled "Introduction to Java 6.0 New Features, Part–I". This article covers the left-over features of Part-I. More specifically, it will cover the ***Pluggabable Annotation Processing API (JSR 269)***, ***Java API for XML Binding (JSR 222)*** and ***Streaming API for XML (JSR 173)***.

2) Pluggable Annotation Processing API

2.1) Introduction to Annotation

Annotations have been there in the Java World from Java 5.0. Java Annotations are a result of the ***JSR 175*** which aimed in providing a ***Meta-Data Facility*** to the ***Java Programming Language***. It can be greatly used by the Build-time Tools and Run-time Environments to do a bunch of useful tasks like ***Code Generation***, ***Validation*** and other valuable stuffs. Java 6 has introduced a new JSR called ***JSR 269***, which is the ***Pluggable Annotation Processing API***. With this API, now it is possible for the Application Developers to write a ***Customized Annotation Processor*** which can be plugged-in to the code to operate on the set of Annotations that appear in a Source File.

Let us see in the subsequent sections how to write a Java File which will make use of Custom Annotations along with a Custom Annotation Processor to process them.

2.2) Writing Custom Annotations

This section provides two ***Custom Annotations*** which will be used by a Sample Java File and a ***Custom Annotation Processor***. One is the ***Class Level Annotation*** and the other is the ***Method Level Annotation***. Following is the listing for both the Annotation Declarations. See how the ***Targets*** for the `Annotations` `ClassLevelAnnotation.java` and `MethodLevelAnnotation.java` are set to `ElementType.TYPE` and `ElementType.METHOD` respectively.

ClassLevelAnnotation.java

```
package net.javabeat.articles.java6.newfeatures.customannotations;
import java.lang.annotation.*;
@Target(value = {ElementType.TYPE})
public @interface ClassLevelAnnotation {
}
```

MethodLevelAnnotation.java

```
package net.javabeat.articles.java6.newfeatures.customannotations;
import java.lang.annotation.*;
@Target(value = {ElementType.METHOD})
public @interface MethodLevelAnnotation {
}
```

AnnotatedJavaFile.java

```
package net.javabeat.articles.java6.newfeatures.customannotations;
@ClassLevelAnnotation()
public class AnnotatedJavaFile {

    @MethodLevelAnnotation
    public void annotatedMethod(){
    }
}
```

The above is a Sample Java File that makes use of the Class Level and the Method Level Annotations. Note that `@ClassLevelAnnotation` is applied at the Class Level and the `@MethodLevelAnnotation` is applied at the method Level. This is because both the Annotation Types have been defined to be tagged to these respective Elements only with the help of `@Target` Annotation.

2.3) Writing a Simple Custom Annotation Processor

TestAnnotationProcessor.java

```
package net.javabeat.articles.java6.newfeatures.customannotations;
import java.util.*;
import javax.annotation.processing.*;
import javax.lang.model.*;
import javax.lang.model.element.*;
@SupportedAnnotationTypes(value= {"*"})
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class TestAnnotationProcessor extends AbstractProcessor  {

    @Override
    public boolean process(
        Set<?> extends TypeElement> annotations, RoundEnvironment roundEnv){

        for (TypeElement element : annotations){
            System.out.println(element.getQualifiedName());
        }
        return true;
    }
}
```

Let us discuss the core points in writing a ***Custom Annotation Processor*** in Java 6. The first notable thing is that Test Annotation Processor class extends `AbstractProcessor` class which encapsulates an Abstract Annotation Processor. We have to inform what Annotation Types our Test Annotation Processor Supports. This is manifested through the Class-Level Annotation called `@SupportedAnnotationTypes().` A value of "*" indicates that all types of Annotations will be processed by this Annotation Processor. Which version of ***Source Files*** this Annotation Processor supports is mentioned through `@SupportedSourceVersion` Annotation.

The javac compiler of Mustang has an option called '-processor' where we can specify the ***Name of the Annotation Processor*** along with a Set of ***Java Source Files*** containing the Annotations. For example, in our case, the command syntax would be something like the following,

```
javac -processor
    net.javabeat.articles.java6.newfeatures.customannotations.TestAnnotationProcessor
        AnnotatedJavaFile.java
```

The above command tells that the name of the Annotation Processor is `net.javabeat.articles.java6.newfeatures.customannotations.TestAnnotationProcessor` and it is going to process the `AnnotatedJavaFile.java`. As soon as this command is issued in the console, the `TestAnnotationProcessor.process()` method will be called by passing the ***Set of Annotations*** that are found in the Source Files along with the ***Annotation Processing Information*** as represented by `RoundEnvironment`. This TestAnnotationProcessor just list the various Annotations present in the Sample Java File (AnnotatedJavaFile.java) by iterating over it.

Following is the output of the above program

```
net.javabeat.articles.java6.newfeatures.customannotations.ClassLevelAnnotation
net.javabeat.articles.java6.newfeatures.customannotations.MethodLevelAnnotation
```

Streaming API for XML

3.1) Introduction

Streaming API for XML, simply called ***StaX***, is an API for reading and writing XML Documents. Why need another XML Parsing API when we already have ***SAX*** (Simple API for XML Parsing) and ***DOM*** (Document Object Model)? Both SAX and DOM parsers have their own advantages and disadvantages and StaX provides a solution for the disadvantages that are found in both SAX and DOM. It is not that StaX replaces SAX and DOM.

SAX, which provides an ***Event-Driven XML Processing***, follows the ***Push-Parsing*** Model. What this model means is that in SAX, Applications will register ***Listeners*** in the form of Handlers to the Parser and will get notified through ***Call-back methods***. Here the SAX Parser takes the control over Application thread by ***Pushing Events*** to the Application. So SAX is a ***Push-Parsing*** model. Whereas StaX is a ***Pull-Parsing*** model meaning that Application can take the control over parsing the XML Documents by pulling (taking) the Events from the Parser.

The disadvantage of DOM Parser is, it will keep the whole ***XML Document Tree in memory*** and certainly this would be problematic if the size of the Document is large. StaX doesn't follow this type of model and it also has options for ***Skipping a Portion*** of a large Document during Reading.

The core StaX API falls into two categories and they are listed below. They are

1. Cursor API
2. Event Iterator API

Applications can any of these two API for parsing XML Documents. Let us see what these APIs' are in detail in the following sections.

3.2) Cursor API

The ***Cursor API*** is used to traverse over the XML Document. Think of a Cursor as some kind of ***Pointer*** pointing at the start of the XML Document and then ***Forwarding the Document*** upon properly instructed. The working model of this Cursor API is very simple. When given a XML Document and asked to parse, the Parser will start reading the XML Document, and if any of the ***Nodes*** (like Start Element, Attribute, Text, End Element) are found it will stop and will give information about the Nodes to the Processing Application if requested. This cursor is a ***Forward only Cursor***, it can never go backwards. Both Reading and Writing operations is possible in this cursor API.

3.3) Sample Application

Let us consider a sample Application which will read data from and to the XML Document with the help of the Cursor API. Following is the sample XML Document. The below XML File contains a list of Events for a person in his/her Calendar.

myCalendar.xml

```
<calendar>
    <event type = "meeting">
        <whom>With my Manager</whom>
        <where>At the Conference Hall</where>
        <date>June 09 2007</date>
        <time>10.30AM</time>
    </event>

    <event type = "birthday">
        <whom>For my Girl Friend</whom>
        <date>01 December</date>
    </event>

</calendar>
```

ReadingUsingCursorApi.java

```
package net.javabeat.articles.java6.newfeatures.stax;
import java.io.*;
import javax.xml.stream.*;
import javax.xml.stream.events.*;
public class ReadingUsingCurorApi {

    private XMLInputFactory inputFactory = null;
    private XMLStreamReader xmlReader = null;

    public ReadingUsingCurorApi() {
        inputFactory = XMLInputFactory.newInstance();
    }

    public void read() throws Exception{

        xmlReader = inputFactory.createXMLStreamReader(
            new FileReader(".\\src\\myCalendar.xml"));

        while (xmlReader.hasNext()){

            Integer eventType = xmlReader.next();
            if (eventType.equals(XMLEvent.START_ELEMENT)){
                System.out.print(" " + xmlReader.getName() + " ");
            }else if (eventType.equals(XMLEvent.CHARACTERS)){
                System.out.print(" " + xmlReader.getText() + " ");
            }else if (eventType.equals(XMLEvent.ATTRIBUTE)){
                System.out.print(" " + xmlReader.getName() + " ");
            }else if (eventType.equals(XMLEvent.END_ELEMENT)){
                System.out.print(" " + xmlReader.getName() + " ");
            }
        }
        xmlReader.close();
    }

    public static void main(String args[]){
        try{
            ReadingUsingCurorApi obj = new ReadingUsingCurorApi();
            obj.read();
        }catch(Exception exception){
            exception.printStackTrace();
        }
    }
}
```

`XMLInputFactory` is the ***Factory Class*** for creating Input Stream objects which is represented by `XMLStreamReader`. An instance of type `XMLStreamReader` is created by calling `XMLInputFactory.createXMLStreamReader()` by passing the XML File to be parsed. At this stage, the Parser is ready to read the XML Contents if a combination call to `XMLStreamReader.hasNext()` and `XMLStreamReader.next()` is made. The entire Document is traversed in the while loop and the appropriate node's value is taken by checking the various Element Types.

3.4) Event Iterator API

The Working Model of this ***Event Iterator API*** is no more different from the Cursor API. As the Parser starts traversing over the XML Document and if any of the ***Nodes*** are found, it will provide this information to the Application that is processing in the form of ***XML Events***. Applications can loop over the entire Document, by requesting for the Next Event. This Event Iterator API is implemented on top of Cursor API.

3.5) Sample Application

Now let us take over a Sample Application using the Event Iterator API which is parsing on the XML Document myCalendar.xml.

ReadingUsingEventIterator.java

```
package net.javabeat.articles.java6.newfeatures.stax;
import java.io.*;
import javax.xml.stream.*;
import javax.xml.stream.events.*;
public class ReadingUsingEventIteratorApi {
    private XMLInputFactory inputFactory = null;
    private XMLEventReader xmlEventReader = null;

    public ReadingUsingEventIteratorApi() {
        inputFactory = XMLInputFactory.newInstance();
    }

    public void read() throws Exception{

        xmlEventReader = inputFactory.createXMLEventReader(
            new FileReader(".\\src\\myCalendar.xml"));

        while (xmlEventReader.hasNext()){

            XMLEvent xmlEvent = xmlEventReader.nextEvent();
            if (xmlEvent.isStartElement()){
                System.out.print(" " + xmlEvent.asStartElement().getName() + " ");
            }else if (xmlEvent.isCharacters()){
                System.out.print(" " + xmlEvent.asCharacters().getData() + " ");
            }else if (xmlEvent.isEndElement()){
                System.out.print(" " + xmlEvent.asEndElement().getName() + " ");
            }
        }

        xmlEventReader.close();
    }
    public static void main(String args[]){
        try{
            ReadingUsingEventIteratorApi obj = new ReadingUsingEventIteratorApi();
            obj.read();
        }catch(Exception exception){
            exception.printStackTrace();
        }
    }
```

Mapping Java Objects and XML Documents using JAXB

4.1) Introduction

JAXB (Java API for XML Binding) technology which was included as part of ***JWSDP (Java Web Services Developer Pack)*** before, is now included with the Mustang Distribution. Simply put, it is a ***Mapping Technology*** for Java and XML Documents. Using JAXB, one can ***Generate XML Documents*** from Java Objects and also they can ***Construct Java Objects*** from one or more XML Documents. In JAXB terms, ***Marshalling*** refers to the process of converting a Java Object to a XML Document and ***Un-Marshalling*** is the reverse of Marshalling which is simply getting a Java Object from one or more XML Documents.

Let us see along with Samples how to work with Marshalling and Un-Marshalling with JAXB.

4.2) Generating XML Documents from Java Objects

Assume that you want to have a XML Representation of a Java object. Using JAXB Marshalling you can do with much ease and the following Sample Application illustrates the same,

Person.java

```
package net.javabeat.articles.java6.newfeatures.jaxb;
```

```
import java.util.Date;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement()
public class Person {
    private String name;
    private int age;
    private Date dateOfBirth;
    private String type;

    public Person() {
    }

    public Person(String name, int age, Date dateOfBirth, String type) {
        this.name = name;
        this.age = age;
        this.dateOfBirth = dateOfBirth;
        this.setType(type);
    }

    @XmlElement()
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @XmlElement()
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @XmlElement()
    public Date getDateOfBirth() {
        return dateOfBirth;
    }

    public void setDateOfBirth(Date dateOfBirth) {
        this.dateOfBirth = dateOfBirth;
    }

    @XmlAttribute()
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
}
```

The above `Person` class has various properties namely ***'name'***, ***'age'***, ***'dateOfBirth'*** and ***'type'*** and objects of this Class are the targets we wish to Marshal. The Person class is annotated with `@XmlRootElement` since we want the name of the Class to be the Root Node. We want ***'name'***, ***'age'***, and ***'dateOfBirth'*** as Child Elements so it has been tagged with `@XmlElement`. Since we want ***'type'*** to appear as an Attribute we had tagged that as `@XmlAttribute`.

Java2XML.java

```
package net.javabeat.articles.java6.newfeatures.jaxb;

import java.io.FileWriter;
import java.util.*;
import javax.xml.bind.*;

public class Java2XML {
    public Java2XML() {
    }

    public static void main(String args[]) throws Exception {
        JAXBContext context = JAXBContext.newInstance(Person.class);
        Marshaller marshaller = context.createMarshaller();
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
        Person person = new Person("Anonymous", 32, new Date(1970, 1, 
10),
                "employee");
        marshaller.marshal(person, new FileWriter(".\\src\\Person.xml"));
    }
}
```

`JAXBContext` serves as the ***Entry Point*** for making use of the JAXB API. It is initiated by the list of Class objects whose objects want to be represented as XML Documents. Then an instance of `Marshaller` is created by calling `JAXBContext.createMarshaller()` method. The Java object of type Person is marshalled into the XML Document by calling the `Marshaller.marshall()` method. Since we want the XML Document to look ***Well-indented***, we have set the property `Marshaller.JAXB_FORMATTED_OUTPUT` to true.

A file named Person.xml would have got generated and following is the listing of that file,

Person.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<person type = "employee">
    <age>32</age>
    <dateOfBirth>1970-01-10 T00:00:00+05:30</dateOfBirth>
    <name>Anonymous</name>
</person>
```

4.3) Representing Java Objects from XML Documents

In the previous section, we saw how to generate an XML Document from a Java Object. Now, let us see the ***Unmarshalling*** process here. As mentioned, ***Unmarshalling*** is the process of representing Java Objects from a XML Document. The process of Unmarshalling is a bit complicated as so many steps are involved. Let us break the steps one by one in greater detail.

4.3.1) Creating the XML Schema Definition File

If ***Java Classes*** represent the templates wherein objects can be created for a particular type, then in XML World, it is the ***XML Schema Definition File*** through which XML Document Instances can be instantiated. The first thing before creating a XML Document is to create a ***XML Schema*** and then to attach the Schema to a XML Document Instance. Let us look into a sample XML Schema File called Items, which will contain a list of ***'Item'*** Elements along with its ***'Name'*** and ***'Price'***. Following is the listing of that File.

Items.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema"
elementFormDefault = "qualified">
    <xs:element name = "Items">
        <xs:complexType>
            <xs:sequence>
                <xs:element maxOccurs = "unbounded" ref = "Item"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name = "Item">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref = "Name"/>
                <xs:element ref = "Price"/>
            </xs:sequence>
            <xs:attribute
                name = "id"
                use = "required"
                type = "xs:NCName"/>
        </xs:complexType>
    </xs:element>
    <xs:element name = "Name" type = "xs:string"/>
    <xs:element name = "Price" type = "xs:string"/>
</xs:schema>
```

The above is the XML Schema Definition File which represents a list of Items. Since ***'Items'*** element contain a list of Item elements and ***'Item'*** element in turn contains both ***'Name'*** and ***'Price'*** Element, both these elements are Composite Elements and the same is represented by ***complexType*** elements.

4.3.2) Generating Java Files from the Schema Definition Files

Now, its time to generate the Java Source Files from this XML Schema Definition File. Mustang comes with a Utility called ***xjc (which stands for XML to Java Compiler)*** which generates Java Source Files when given a XML Schema File. Xjc is nothing but a bat file location in the bin directory of the Java 6 Installation Path.

Run the following command to generate the Java Source Files.

```
xjc –p net.javabeat.articles.java6.newfeatures.jaxb Items.xsd
```

The only argument to xjc is the name of the ***XML Schema File*** which is represented by Items.xsd and the '-p' option specifies the package to which the generated Java Files have to be associated. This results is the generation of the following source files.

Item.java

```
package net.javabeat.articles.java6.newfeatures.jaxb;

import javax.xml.bind.annotation.*;
import javax.xml.bind.annotation.adapters.*;
import net.javabeat.articles.java6.newfeatures.jaxb.*;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = { "name", "price" })
@XmlRootElement(name = "Item")
public class Item {
    @XmlElement(name = "Name")
    protected String name;
    @XmlElement(name = "Price")
    protected String price;
    @XmlAttribute(required = true)
    @XmlJavaTypeAdapter(CollapsedStringAdapter.class)
    protected String id;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public String getPrice() {
        return price;
    }

    public void setPrice(String value) {
        this.price = value;
    }

    public String getId() {
        return id;
    }

    public void setId(String value) {
        this.id = value;
    }
}
```

This file is just a Data file for the corresponding ***'Item'*** element which contains getter and setter methods with properly instructed Annotations. Note that the two sub-elements namely ‘***name’*** and ‘***type’*** has been tagged with `@XmlElement` Annotation and the attribute ‘***id’*** has been tagged with `@XmlAttribute` Annotation.

Items.java

```
package net.javabeat.articles.java6.newfeatures.jaxb;

import java.util.*;
import javax.xml.bind.annotation.*;
import net.javabeat.articles.java6.newfeatures.jaxb.*;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = { "item" })
@XmlRootElement(name = "Items")
public class Items {
    @XmlElement(name = "Item")
    protected List<Item> item;

    public List<Item> getItem() {
        if (item == null) {
            item = new ArrayList<Item>();
        }
        return this.item;
    }
}
```

This is the ***Container Class*** for the Item element and it just contains a getter for the Item object. Note how the list of ***'Item'*** elements in the Schema Document has been mapped to List of Item (`List<Item>`) elements. This is the ***Default Mapping*** given by the Sun's Reference Implementation for JAXB.

ObjectFactory.java

```
package net.javabeat.articles.java6.newfeatures.jaxb;

import javax.xml.bind.JAXBElement;
import javax.xml.bind.annotation.XmlElementDecl;
import javax.xml.bind.annotation.XmlRegistry;
import javax.xml.namespace.QName;
import net.javabeat.articles.java6.newfeatures.jaxb.Item;
import net.javabeat.articles.java6.newfeatures.jaxb.Items;
import net.javabeat.articles.java6.newfeatures.jaxb.ObjectFactory;

@XmlRegistry
public class ObjectFactory {
    private final static QName _Price_QNAME = new QName("", "Price");
    private final static QName _Name_QNAME = new QName("", "Name");

    public ObjectFactory() {
    }

    public Item createItem() {
        return new Item();
    }

    public Items createItems() {
        return new Items();
    }

    @XmlElementDecl(namespace = "", name = "Price")
    public JAXBElement<String> createPrice(String value) {
        return new JAXBElement<String>(_Price_QNAME, String.class, null,
value);
    }

    @XmlElementDecl(namespace = "", name = "Name")
    public JAXBElement<String> createName(String value) {
        return new JAXBElement<String>(_Name_QNAME, String.class, null,
value);
    }
}
```

As guessed from its name, this is the ***Factory Class*** for creating various elements like ***'Items'***, ***'Item'***, ***'Name'*** and ***Price'***. Appropriate methods are available for creating ***'Items'***, ***'Item'***, ***'Name'*** and ***'Price'*** in the form of `ObjectFactory.createItems()`, `ObjectFactory.createItem()`, `ObjectFactory.createName()` and `ObjectFactory.createPrice()` respectively.

4.3.3) Creating a Sample XML Document

Now let us create a Sample XML Document against the Schema Definition File Items.xsd. Following is the sample XML file called Items.xml. Note how the XML Document Instance is related with the XML Schema Definition File with the help of '***xsi:noNamespaceSchemaLocation'*** attribute.

Items.xml

```
<Items xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "file:C:/Items.xsd">
    <Item id = "LAP001">
        <Name>Laptop</Name>
        <Price>4343$</Price>
    </Item>
    <Item id = "TV001">
        <Name>Television</Name>
        <Price>433$</Price>
    </Item>
    <Item id = "DVD001">
        <Name>DVD Player</Name>
        <Price>763$</Price>
    </Item>
</Items>
```

4.3.4) Unmarshalling the XML Document to construct the Java object

The code to Unmarshall the XML Document is given below. The code first represents a `JAXBContext` object by passing in a ***Context Path*** which is usually the package name where the compiled Java Class files are located. Then an `Unmarshaller` object is created by calling `JAXBContext.createUnmarshaller()` which does the unmarshalling operation by calling the `Unmarshaller.unmarshall()` method passing in the XML File. Then the object is iterated over the get the actual data contents from the XML File.

XML2Java.java

```
package net.javabeat.articles.java6.newfeatures.jaxb;

import java.io.*;
import javax.xml.bind.*;

public class XML2Java {
   public static void main(String args[]) throws Exception {
      JAXBContext context = JAXBContext
            .newInstance("net.javabeat.articles.java6.newfeatures.jaxb");
      Unmarshaller unmarshaller = context.createUnmarshaller();
      Items items = (Items) unmarshaller.unmarshal(new FileReader(
            ".\\src\\Items.xml"));
      List listOfItems = items.getItem();
      for (Item item : listOfItems) {
         System.out.println("Name = " + item.getName() + ", Price = "
               + item.getPrice() + ", Id = " + item.getId());
      }
   }
}Following is the output of the above program
```

```
Name = Laptop, Price = 4343$, Id = LAP001 Name = Television, Price =
433$, Id = TV001 Name = DVD Player, Price = 763$, Id = DVD001
```

5) Conclusion

This article covered the leftover features of the Part I New features of Java 6. Starting with ***Pluggable Annotation Processing API***, it covered what Annotations are, then guided us how to write ***Custom Annotations*** and ***Custom Annotation Processor***. Then it traversed over the ***Cursor API*** and the ***Event Iterator API*** of StaX such as how to read XML Documents along with well-defined samples. Finally through ***Java API for XML Binding***, it details how to map Java and XML Documents with some sample applications.

```
}
```

If `XMLStreamReader` class represents the Reader for stream reading the XML Contents, then `XMLEventReader` represents the class for reading the XML Document as ***XML Events*** (represented by `javax.xml.stream.events.XMLEvent`). The rest of the reading logic is the same as that of the `ReadingUsingCurorApi.java`.

Deployment

Overview

Desktop clients have a wide variety of Java platforms installed, from Microsoft® Java Virtual Machine (MSJVM) to the latest Java SE 6 updates from Sun Microsystems. They run various operating systems from Sun, Microsoft, Apple, Red Hat, and others, and are connected to the internet at a wide range of connection speeds. How are content providers to deliver Java content to all of these clients with the best possible user experience?

Various sources have published JavaScript techniques for detecting and deploying the Java platform for use by Java Plug-In applets and Java Web Start applications. These scripts generally have serious limitations and fail to support the varied combinations of browser, OS, and configuration options found on today's clients.

This section outlines an enhanced JavaScript that can be used to deploy applets and applications to a large variety of clients. It also provides advice on using some of the most powerful features available in Java Web Start and Java Plug-In, and an outline of the differences between these two deployment vehicles. These tips come from the Sun Java Deployment Team and include the following topics:

Applets Versus Applications

The Java Plug-In runs (possibly) multiple Java applets within the same Java Virtual Machine, in the same process as the web browser. Java Web Start runs standalone Java applications in their own Java Virtual Machine, each in its own process, separate from any running web browsers.

Although they share much of the same code and have many of the same features, this fundamental difference lends advantages and disadvantages to each. The choice of which to use should depend on which, if any, of these differences you want to take advantage of in your Java program.

- **Java Version**

  Java Plug-In can run only the version of Java with which it was distributed. This means that to take advantage of the latest features in Java Plug-In, you must also use the latest version of Java. Further, recent changes in the Deployment Security Policy require that only the latest version deployed on the client machine be used. This is usually not a problem, since versions of Java are generally forward compatible.

  For cases where compatibility between major releases is an issue, we have also provided a feature called the Family ClassID (for Microsoft Internet Explorer only).

  Java Web Start will also be used only from the latest Java version installed on a system, but the version of Java Web Start run is independent of the version of the Java platform used. A Java Web Start application can specify the exact version, or version range, of the Java platform that it will use.

- **Cookies**

  Java Plug-In applets run in the same session as the browser, and so have access to both "persistent" cookies and "session" cookies already stored in the browser session.

  Java Web Start runs outside the browser process. It can access "persistent" cookies stored by the browser (it does this by using the WinINet API on Windows), but has no access to "session" cookies unless set in that application using the java.net.CookieManager and java.net.CookieHandler API's.

- **Browser interactions**

  Java Plug-In includes the ability for applets to interact with the current browser session using the Common DOM API, Java to JavaScript, and JavaScript to Java communications.

  Java Web Start is limited to invoking the default browser to display specific URL's (see JNLP API).

- **Java Runtime**

  Java Plug-In shares one Java Runtime Environment among multiple Java applets. As such, no applet can have control over the startup parameters of the JRE (Java Runtime parameters). Java Runtime parameters must be set by the user (for all applets) in the Java Control Panel.

  Java Web Start applications have (limited) control over the Java Runtime parameters that is used to launch that application.

- **JNLP API**

  Java Web Start includes access to the JNLP API, which allows unsigned applications access to persistent storage, download control, file I/O, and more. These API's. are not available to a Java Plug-in applet.

- **Shortcuts**

  Java Web Start applications can install shortcuts that allow the application to be rerun, either online or offline, independent of the browser.

- **Other differences**

  There are several other minor differences caused by need to maintain compatibility with earlier versions. For further information on the specific differences between Java Web Start and Java Plug-In can be found in the Migration Guide.

JavaScript used for deploying Java content

The JavaScript (*deployJava.js*) documented below makes it possible to automatically install Java for Java Plug-in applets and Java Web Start applications.

The script exposes a single object, named *deployJava*, which contains the following public functions:

- *getJREs()* - Returns a list of currently-installed JRE versions.
- *installLatestJRE()* - Triggers the installation of the latest JRE.
- *installJRE(requestVersion)* - Triggers the installation of the specified *requestVersion*, the latest version matching the specified *requestVersion*, or the latest JRE.
- *versionCheck(version)* - Returns true if there is a matching JRE version currently installed (among those detected by *getJREs()*).
- *writeAppletTag(attributes, parameters)* - Outputs an applet tag with the specified attributes and parameters. The parameters argument is optional.
- *runApplet(attributes, parameters, minimumVersion)* - Ensures that an appropriate JRE is installed and then runs an applet.
- *isWebStartInstalled(minimumVersion)* - Returns true if an installation of Java Web Start of the specified *minimumVersion* is found.
- *createWebStartLaunchButton(jnlp, minimumVersion)* - Outputs a launch button for the specified JNLP URL. When clicked, the button will ensure that an appropriate JRE is installed and then launch the JNLP application.

Deploying Java Plug-In Applets

With recent changes to the Deployment Security Policy, Java Plug-In applets will normally run with only the latest deployed Java version on a client machine.

You can use the *runApplet()* function in *deployJava* to ensure that a minimum Java Runtime Environment is available on a client machine before launching the applet.

```
<script src="http://java.com/js/deployJava.js"></script>
<script>
    var attributes =
{codebase:'http://java.sun.com/products/plugin/1.5.0/demos/jfc/Java2D',
                      code:'java2d.Java2DemoApplet.class',
                      archive:'Java2Demo.jar',
                      width:710, height:540} ;
    var parameters = {fontSize:16} ;
    var version = '1.6.0' ;
    deployJava.runApplet(attributes, parameters, version);
</script>
```

The above example will launch the *Java 2D* applet with one parameter (*fontSize*) after ensuring that Java SE version 1.6.0 is installed on the client.

Click the following link to run the Java 2D Demo Applet.

Deploying Java Web Start Applications

A Java Web Start Application can be deployed simply by creating a jnlp file that describes only the title, vendor, java version, jar file(s), and main class of the application. Here is an example of a minimal jnlp file:

```
<jnlp>
    <information>
        <title>Notepad</title>
        <vendor>Sun Microsystems</vendor>
    </information>
    <resources>
        <java version="1.4+"/>
        <jar
href="http://java.sun.com/products/javawebstart/apps/notepad.jar"/>
    </resources>
    <application-desc main-class="Notepad"/>
</jnlp>
```

The application can then be deployed simply by providing a link to the jnlp file on your web page:

```
<a
href="http://java.sun.com/products/javawebstart/apps/notepad.jnlp">Launch
Notepad</a>
```

Many other elements can be added to the jnlp file to control the user experience, security, and update process of the application, or to take advantage of several other features of the Java Network Launching Protocol (JNLP). For a complete list of jnlp elements and attributes see:
http://java.sun.com/javase/6/docs/technotes/guides/javaws/developersguide/syntax.html#intro

Java Web Start can use its Auto-Download mechanism to download the version of the JRE that it requires to run an application, but if the application wants to use advanced features of JNLP and Java Web Start that were added to a particular version, you may want to ensure that at least that version is installed before launching Java Web Start.

Suppose an application wants to use the SingleInstance Service (introduced in 1.5.0). Instead of just specifying *<jnlp spec="1.5.0" .../>* and letting the application fail on systems where only 1.4.2 or earlier is installed, you can use the *deployJava* javascript to ensure that at least version 1.5.0 is installed before launching Java Web Start.

```
<script src="http://java.com/js/deployJava.js"></script>
<script>
    var url =
"http://java.sun.com/products/javawebstart/apps/notepad.jnlp";
    deployJava.createWebStartLaunchButton(url, '1.6.0');
</script>
```

For an application not having specific version requirements you can just use the function without supplying *minimumVersion*.

```
<script>
    var url =
"http://java.sun.com/products/javawebstart/apps/notepad.jnlp";
    deployJava.createWebStartLaunchButton(url);
</script>
```

Pack200 in Java Plug-In or Java Web Start

Pack200 technology (see Deployment Tools), introduced in Java SE 5.0, drastically reduces the size of Java programs deployed within Java Archive (JAR) files.

Enabling Pack200 compression for a particular application deployed via Java Web Start or the Java Plug-In requires some support on the web server, because the Java client needs to negotiate with the web server regarding whether it can decompress Pack200-compressed JAR files. However, no changes to the client are needed, and the JNLP files and/or applet tags in the HTML remain unmodified when enabling Pack200 support.

There are two basic approaches available. For the Apache 2 web server, a simple configuration file change can be made which allows any Java application on the web server to be served using Pack200 compression.

Another option which provides more functionality is to run a Java servlet on the web server, if a servlet container is available. Two sample servlets are provided by Sun. The first is a simple servlet that only implements Pack200 compression. The source to it can be found in the Deployment Guide.

The second is the *JnlpDownloadServlet*. This servlet also implements the JNLP version-based and extension protocols, automatic generation of JARDiff files, and automatic codebase substitution. It can be found in the sample directory of the JDK.

Order of Installation of JRE's

If multiple JRE's are required to run various Java Plug-in applets on the same machine, it is recommended to install the JRE's in the order of their versions. The oldest version should be installed first and the newest version installed last. This will avoid the problem of the dynamic clsid *{8AD9C840-044E-11D1-B3E9-00805F499D93}* being used in an object tag that is not using the latest version of the JRE on the machine.

Starting from JRE 5.0u6 with SSV support, the above is not an issue because the latest version of JRE on the machine will be used. In addition, we have added a new dynamic version clsid *{CAFEEFAC-FFFF-FFFF-FFFF-ABCDEFFEDCBA}*. If the new dynamic clsid is used in the object tag, the latest version of the JRE will be used independently of the installation order of the JRE's.

Installation order should have no effect on Java Web Start. In any case the highest version of the JRE on the system will contain the version of Java Web Start that is run.

ClassLoader and Accessing Resources

Resources accessed in a Java Web Start application or Java Plug-in applet may be cached on the client machine in the Deployment Cache. It is unwise to assume the format or content of this cache, as it may change between versions.

When porting stand alone programs to Java Web Start or Java Plug-in, problems can occur when code has inherent assumptions that it is loaded by the *SystemClassLoader*. In Java Plug-in resources are loaded by the *PluginClassLoader* (which extends *sun.applet.AppletClassLoader*, which in turn extends *java.net.URLClassLoader*). In Java Web Start resources are loaded by the *JNLPClassLoader* (which as of JDK 6 extends *java.net.URLClassLoader*).

Access the *ClassLoader* being used with:

```
ClassLoader cl = Thread.getCurrent().getContextClassLoader();
```

ClassLoader.getResource() returns a URL, but any code that assumes the URL is a JarURL to a FileURL, and then tries to decompose that FileURL to find the underlying file path will fail. The correct way to access resources is to use *getResourceAsStream()* which will return the correct content whatever type of *ClassLoader* is used to access the resource. If the resource is already cached, the contents of the resource will be returned from the cache directly, so there won't be extra network connections to the resource itself.

We do not recommend modifying the contents of the Java deployment cache directly. The cache is a private implementation of Java Web Start / Java Plug-in, and is subject to change anytime.

Many applications and libraries try to deploy properties files and other "resources" by including them in the same directory as the jar file that uses them, and then expect to be able to decompose the the URL returned from *getResource()* to construct the path to these files. Developers argue that this is needed so the application can later modify these property files or other "resources" for use in subsequent launchings of the app. When porting to web deployed applications, they then find they need to repackage these into the jar files of the app, and consider them only the "default" content, and use one of several other mechanisms to persist the modified versions on the client machine (by writing files or by using either the Preference API or the JNLP PersistenceService.)

Lazy Downloading

When applications are large, it can be useful to only download the part of the application that is required to start up, and then download the rest on demand. This process is referred to as lazy downloading.

Java Web Start has support for lazy downloading, but few developers use it. It can be a way to significantly improve the download and startup time in some applications. To effectively use lazy downloading, Java Web Start must be aware which jar to download to resolve a request for a specific resource. Previous versions of Java Web Start required a complex specification of parts and packages to provide this information. Beginning with version 6.0, the same thing can be accomplished using Jar Indexing.

Jar Indexing is a simple and effective way to download only the required jars, and avoid downloading everything when a nonexistent resource is requested. See Jar Indexing.

Java Plug-in has built-in support for lazy downloading (that is, downloading is lazy by default), and also supports Jar Indexing. Developers should also try to **NOT** use individual classes but package them as JARs instead.

Security Enhancements

1.**XML Digital Signature API**
2.Smart Card I/O API

Introduction

One of the principal design centers for Java Platform, Standard Edition 6 (Java SE 6) was to improve performance and scalability by targeting performance deficiencies highlighted by some of the most popular Java benchmarks currently available and also by working closely with the Java community to determine key areas where performance enhancements would have the most impact.

This guide gives an overview of the new performance and scalability improvements in Java Standard Edition 6 along with various industry standard and internally developed benchmark results to demonstrate the impact of these improvements.

New Features and Performance Enhancements

Java SE 6 includes several new features and enhancements to improve performance in many areas of the platform. Improvements include: synchronization performance optimizations, compiler performance optimizations, the new Parallel Compaction Collector, better ergonomics for the Concurrent Low Pause Collector and application start-up performance.

2.1 Runtime performance optimizations

2.1.1 Biased locking

Biased Locking is a class of optimizations that improves uncontended synchronization performance by eliminating atomic operations associated with the Java language's synchronization primitives. These optimizations rely on the property that not only are most monitors uncontended, they are locked by at most one thread during their lifetime.

An object is "biased" toward the thread which first acquires its monitor via a monitorenter bytecode or synchronized method invocation; subsequent monitor-related operations can be performed by that thread without using atomic operations resulting in much better performance, particularly on multiprocessor machines.

Locking attempts by threads other that the one toward which the object is "biased" will cause a relatively expensive operation whereby the bias is revoked. The benefit of the elimination of atomic operations must exceed the penalty of revocation for this optimization to be profitable.

Applications with substantial amounts of uncontended synchronization may attain significant speedups while others with certain patterns of locking may see slowdowns.

Biased Locking is enabled by default in Java SE 6 and later. To disable Biased Locking, please add to the command line -XX:-UseBiasedLocking .

For more on Biased Locking, please refer to the ACM OOPSLA 2006 paper by Kenneth Russell and David Detlefs: "Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk

Rebiasing".

2.1.2 Lock coarsening

There are some patterns of locking where a lock is released and then reacquired within a piece of code where no observable operations occur in between. The lock coarsening optimization technique implemented in hotspot eliminates the unlock and relock operations in those situations (when a lock is released and then reacquired with no meaningful work done in between those operations). It basically reduces the amount of synchronization work by enlarging an existing synchronized region. Doing this around a loop could cause a lock to be held for long periods of times, so the technique is only used on non-looping control flow.

This feature is on by default. To disable it, please add the following option to the command line: -XX:-EliminateLocks

2.1.3 Adaptive spinning

Adaptive spinning is an optimization technique where a two-phase spin-then-block strategy is used by threads attempting a contended synchronized enter operation. This technique enables threads to avoid undesirable effects that impact performance such as context switching and repopulation of Translation Lookaside Buffers (TLBs). It is “adaptive" because the duration of the spin is determined by policy decisions based on factors such as the rate of success and/or failure of recent spin attempts on the same monitor and the state of the current lock owner.

For more on Adaptive Spinning, please refer to the presentation by Dave Dice: "Synchronization in Java SE 6"

2.1.4 Support for large page heap on x86 and amd64 platforms

Java SE 6 supports large page heaps on x86 and amd64 platforms. Large page heaps help the Operating System avoid costly Translation-Lookaside Buffer (TLB) misses to enable memory-intensive applications perform better (a single TLB entry can represent a larger memory range).

Please note that large page memory can sometimes negatively impact system performance. For example, when a large amount of memory is pinned by an application, it may create a shortage of regular memory and cause excessive paging in other applications and slow down the entire system. Also please note for a system that has been up for a long time, excessive fragmentation can make it impossible to reserve enough large page memory. When it happens, the OS may revert to using regular pages. Furthermore, this effect can be minimized by setting `-Xms == -Xmx, -XX:PermSize == -XX:MaxPermSize` and `-XX:InitialCodeCacheSize == -XX:ReserverCodeCacheSize .`

Another possible drawback of large pages is that the default sizes of the perm gen and code cache might be larger as a result of using a large page; this is particularly noticeable with page sizes that are larger than the default sizes for these memory areas.

Support for large pages is enabled by default on Solaris. It's off by default on Windows and Linux. Please add to the command line -XX:+UseLargePages to enable this feature. Please note that Operating System configuration changes may be required to enable large pages. For more information, please refer to the documentation on Java Support for Large Memory Pages on Sun Developer Network.

2.1.5 Array Copy Performance Improvements

The method instruction System.arraycopy() was further enhanced in Java SE 6. Hand-coded assembly stubs are now used for each type size when no overlap occurs.

2.1.6 Background Compilation in HotSpot™ Client Compiler

Prior to Java SE 6, the HotSpot Client compiler did not compile Java methods in the background by default. As a consequence, Hyperthreaded or Multi-processing systems couldn't take advantage of spare CPU cycles to optimize Java code execution speed. Background compilation is now enabled in the Java SE 6 HotSpot client compiler.

2.1.7 New Linear Scan Register Allocation Algorithm for the HotSpot™ Client Compiler

The HotSpot client compiler features a new linear scan register allocation algorithm that relies on static single assignment (SSA) form. This has the added advantage of providing a simplified data flow analysis and shorter live intervals which yields a better tradeoff between compilation time and program runtime. This new algorithm has provided performance improvements of about 10% on many internal and industry-standard benchmarks.

For more information on this new new feature, please refer to the following paper: Linear Scan Register Allocation for the Java HotSpot™ Client Compiler

2.2 Garbage Collection

2.2.1 Parallel Compaction Collector

Parallel compaction is a feature that enables the parallel collector to perform major collections in parallel resulting in lower garbage collection overhead and better application performance particularly for applications with large heaps. It is best suited to platforms with two or more processors or hardware threads.

Previous to Java SE 6, while the young generation was collected in parallel, major collections were performed using a single thread. For applications with frequent major collections, this adversely affected scalability.

Parallel compaction is used by default in JDK 6, but can be enabled by adding the option -XX:+UseParallelOldGC to the command line in JDK 5 update 6 and later.

Please note that parallel compaction is not available in combination with the concurrent mark sweep collector; it can only be used with the parallel young generation collector (-XX:+UseParallelGC). The documents referenced below provide more information on the available collectors and recommendations for their use.

For more on the Parallel Compaction Collection, please refer to the Java SE 6 release notes. For more information on garbage collection in general, the HotSpot memory management whitepaper describes the various collectors available in HotSpot and includes recommendations on when to use parallel compaction as well as a high-level description of the algorithm.

2.2.2 Concurrent Low Pause Collector: Concurrent Mark Sweep Collector Enhancements

The Concurrent Mark Sweep Collector has been enhanced to provide concurrent collection for the System.gc() and Runtime.getRuntime().gc() method instructions. Prior to Java SE 6, these methods stopped all application threads in order to collect the entire heap which sometimes resulted in lengthy pause times in applications with large heaps. In line with the goals of the Concurrent Mark Sweep Collector, this new feature is enabling the collector to keep pauses as short as possible during full heap collection.

To enable this feature, add the option -XX:+ExplicitGCInvokesConcurrent to the Java command line.

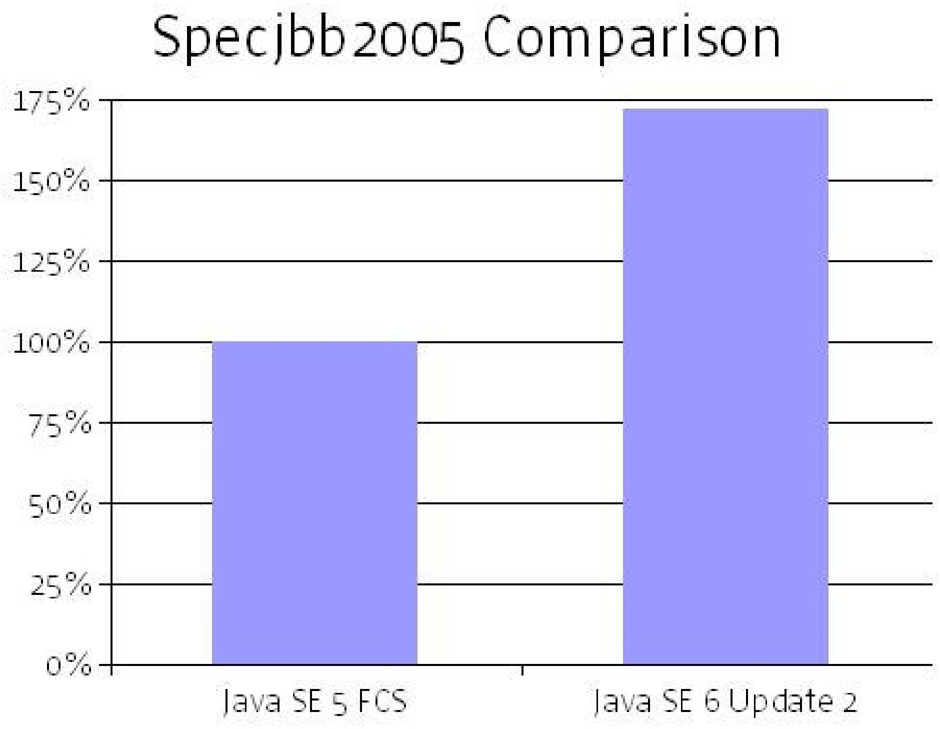
The concurrent marking task in the CMS collector is now performed in parallel on platforms with multiple processors . This significantly reduces the duration of the concurrent marking cycle and enables the collector to better support applications with larger numbers of threads and high object allocation rates, particularly on large multiprocessor machines.

For more on these new features, please refer to the Java SE 6 release notes.

2.3 Ergonomics in the 6.0 Java Virtual Machine

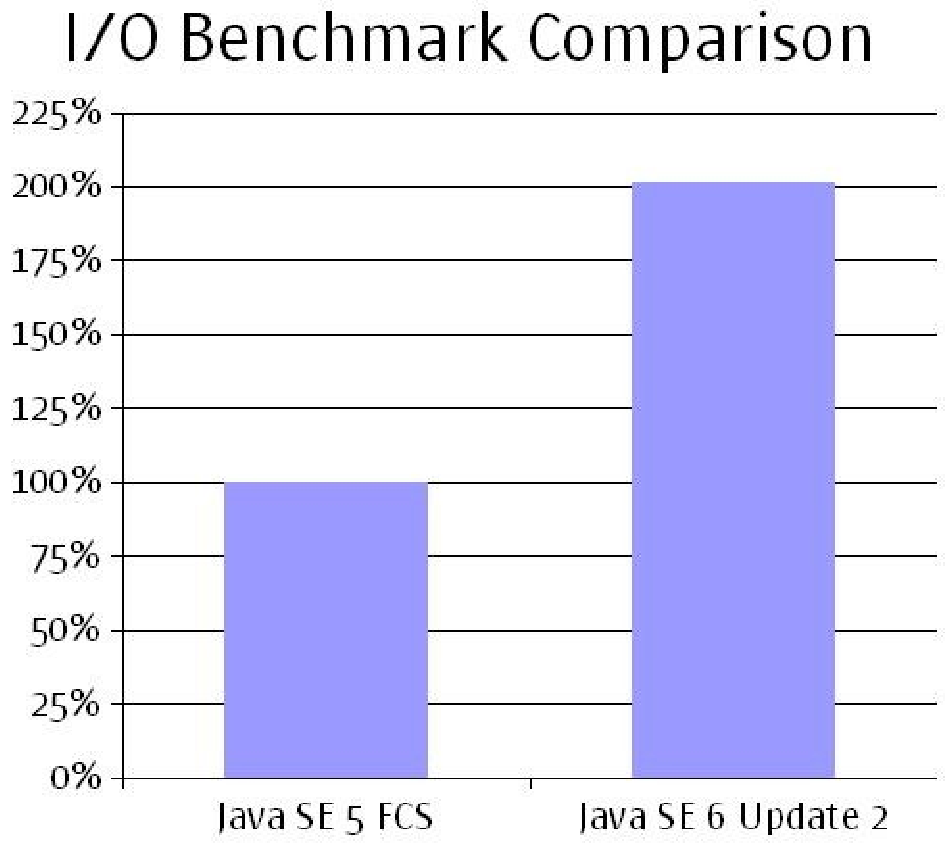
In Java SE 5, platform-dependent default selections for the garbage collector, heap size, and runtime compiler were introduced to better match the needs of different types of applications while requiring less command-line tuning. New tuning flags were also introduced to allow users to specify a desired behavior which in turn enabled the garbage collector to dynamically tune the size of the heap to meet the specified behavior. In Java SE 6, the default selections have been further enhanced to improve application runtime performance and garbage collector efficiency.

The chart below compares out-of-the-box SPECjbb2005™ performance between Java SE 5 and Java SE 6 Update 2. This test was conducted on a Sun Fire V890 with 24 x 1.5 GHz UltraSparc CPU's and 64 GB RAM running Solaris 10:



In each case the benchmarks were ran without any performance flags. Please see the SPECjbb 2005 Benchmark Disclosure

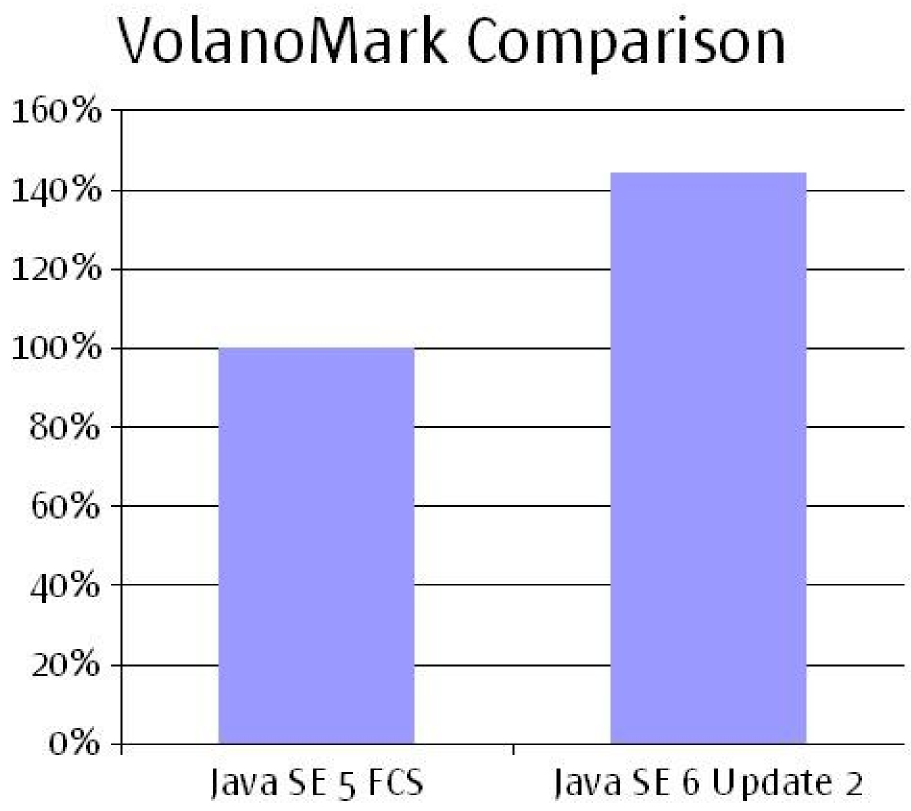
We also compared I/O performance between Java SE 5 and Java SE 6 Update 2. This test was conducted on a Sun Fire V890 with 24 x 1.5 GHz UltraSparc CPU's and 64 GB RAM running Solaris 10:



In each case the benchmarks were ran without any performance flags.

We compared VolanoMark™ 2.5 performance between Java SE 5 and Java SE 6. VolanoMark is a pure Java benchmark that measures both (a) raw server performance and (b) server network scalability performance. In this benchmark, the client side simulates up to 4,000 concurrent socket connections. Only those VMs that successfully scale up to 4,000 connections pass the test. In both the raw performance and network scalability tests, the higher the score, the better the result.

This test was conducted on a Sun Fire V890 with 24 x 1.5 GHz UltraSparc CPU's and 64 GB RAM running Solaris 10:



In each case we ran the benchmark in loopback mode without any performance flags. The result shown is based upon relative throughput (messages per second with 400 loopback connections).

The full Java version for Java SE 5 is:

```
java version "1.5.0"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-b64)
Java HotSpot(TM) Client VM (build 1.5.0-b64, mixed mode)
```

The full Java version for Java SE 6 is:

```
java version "1.6.0_02"
Java(TM) SE Runtime Environment (build 1.6.0_02-b05)
Java HotSpot(TM) Client VM (build 1.6.0_02-b05, mixed mode)
```

Please see the VolanoMark™ 2.5 Benchmark Disclosure

Some other improvements in Java SE 6 include:

- On server-class machines, a specified maximum pause time goal of less than or equal to 1 second will enable the Concurrent Mark Sweep Collector.
- The garbage collector is allowed to move the boundary between the tenured generation and the young generation as needed (within prescribed limits) to better achieve performance goals. This mechanism is off by default; to activate it add this to the command line: option `-XX:+UseAdaptiveGCBoundary` .
- Promotion failure handling is turned on by default for the serial (-XX:+UseSerialGC) and Parallel Young Generation (-XX:+ParNewGC) collectors. This feature allows the collector to start a minor collection and then back out of it if there is not enough space in the tenured generation to promote all the objects that need to be promoted.
- An alternative order for copying objects from the young to the tenured generation in the parallel scavenge collector has been implemented. The intent of this feature is to decrease cache misses for objects accessed in the tenured generation.This feature is on by default. To disable it, please add this to the command line -XX:-UseDepthFirstScavengeOrder
- The default young generation size has been increased to 1MB on x86 platforms
- The Concurrent Mark Sweep Collector's default Young Generation size has been increased.
- The minimum young generation size was increased from 4MB to 16MB.
- The proportion of the overall heap used for the young generation was increased from 1/15 to 1/7.
- The CMS collector is now using the survivor spaces by default, and their default size was increased. The primary effect of these changes is to improve application performance by reducing garbage collection overhead. However, because the default young generation size is larger, applications may also see larger young generation pause times and a larger memory footprint.

2.4 Client-side Performance Features and Improvements

2.4.1 New class list for Class Data Sharing

To reduce application startup time and footprint, Java SE 5.0 introduced a feature called "class data sharing" (CDS). On 32-bit platforms, this mechanism works as follows: the Sun provided installer loads a set of classes from the system jar (the jar file containing all the Java class library, called rt.jar) file into a private internal representation, and dumps that representation to a file, called a "shared archive". On subsequent JVM invocations, the shared archive is memory-mapped in, saving the cost of loading those classes and allowing much of the Java Virtual Machine's metadata for these classes to be shared among multiple JVM processes.

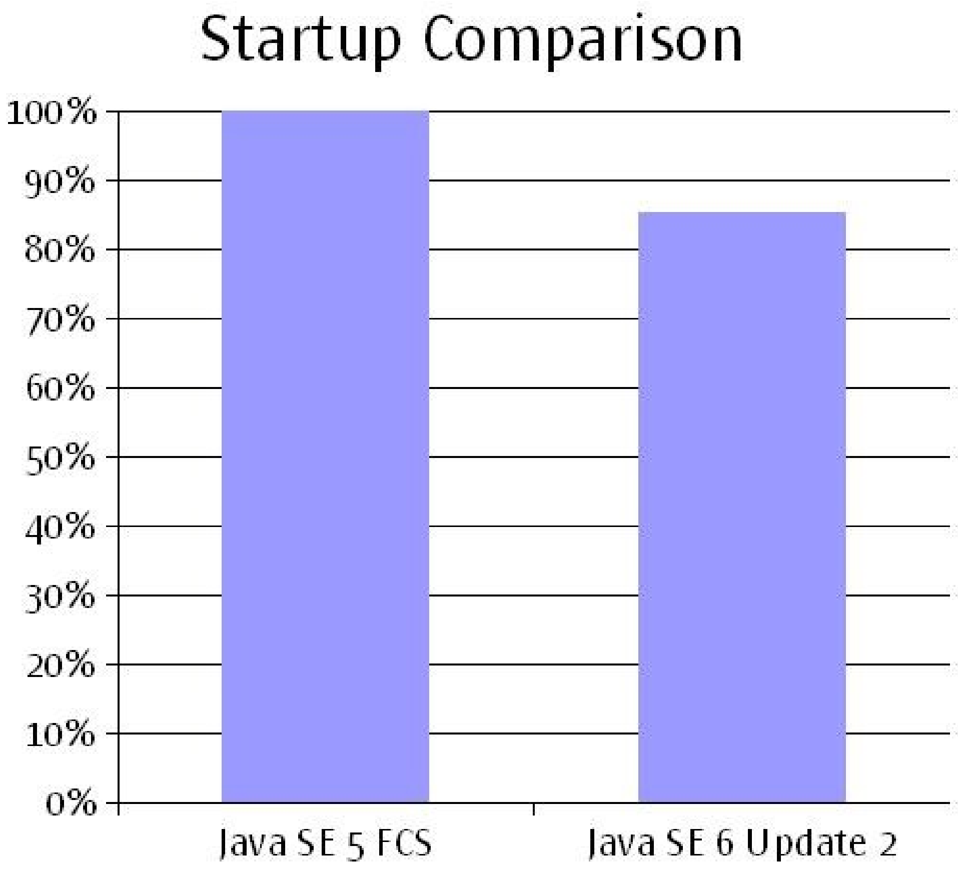
In Java SE 6.0, the list of classes in the "shared archive" has been updated to better reflect the changes to the system jar file.

2.4.2 Improvements to the boot class loader

The Java Virtual Machine's boot and extension class loaders have been enhanced to improve the cold-start time of Java applications. Prior to Java SE 6, opening the system jar file caused the Java Virtual Machine to read a one-megabyte ZIP index file that translated into a lot of disk seek activity when the file was not in the disk cache. With "class data sharing" enabled, the Java Virtual Machine is now provided with a "meta-index" file (located in jre/lib) that contains high-level information about which packages (or package prefixes) are contained in which jar files.

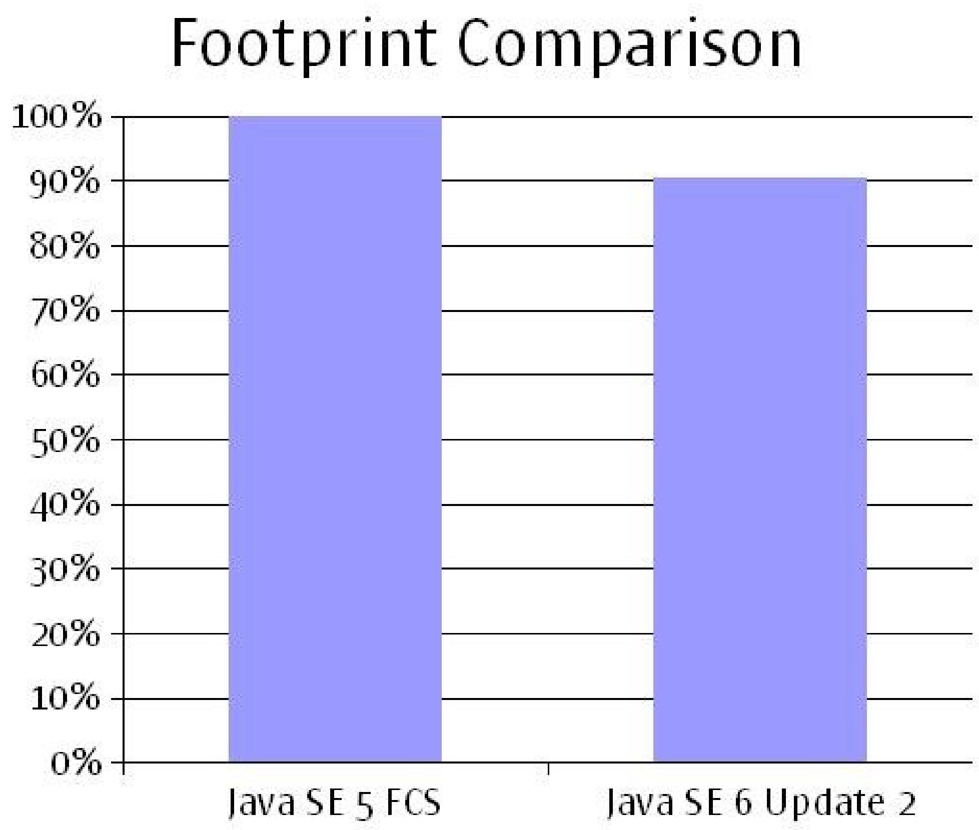
This helps the JVM avoid opening all of the jar files on the boot and extension class paths when a Java application class is loaded. Check bug 6278968} for more details.

Below we show a chart comparing application start-up time performance between Java SE 5 and Java SE 6 Update 2. This test was conducted on an Intel Core 2 Duo 2.66GHz desktop machine with 1GB of memory:



The application start-up comparison above shows relative performance (smaller is better) and in each case the benchmarks were ran without any performance flags.

We also compared memory footprint size required between Java SE 5 and Java SE 6 Update 2. This test was conducted on
an Intel Core 2 Duo 2.66GHz desktop machine with 1GB of memory:



The footprint comparison above shows relative performance (smaller is better) and in each case the benchmarks were run without any performance flags.

Despite the addition of many new features, the Java Virtual Machine's core memory usage has been pared down to make the actual memory impact on your system even lower than with Java SE 5

2.4.3 Splash Screen Functionality

Java SE 6 provides a solution that allows an application to show a splash screen before the virtual machine starts. Now, a Java application launcher is able to decode an image and display it in a simple non-decorated window.

2.4.4 Swing's true double buffering

Swing's true double buffering has now been enabled. Swing used to provide double buffering on an application basis, it now provides it on a per-window basis and native exposed events are copied directly from the double buffer. This significantly improves Swing performance, especially on remote servers.

Please see the Scott Violet's Blog for full details.

2.4.5 Improving rendering on windows systems

The UxTheme API, which allows standard Look&Feel rendering of windows controls on Microsoft Windows systems, has been adopted to improve the fidelity of Swing Systems Look & Feels.

3 New Platform Support

Please see the Supported System Configurations chart for full details.

3.1 Operating Environments

3.1.1 Windows Vista

Java SE 6 is supported on Windows Vista Ultimate Edition, Home Premium Edition, Home Basic Edition, Enterprise Edition and Business Edition in addition to Windows XP Home and Professional, 2000 Professional, 2000 Server, and 2003 Server.