**Highlights of Technology Changes in Java SE 7**

The following list contains links to the the enhancements pages in the [Java SE 7](http://docs.oracle.com/javase/7/docs/) guides documentation. Choose a technology for further information.

* [Swing](http://docs.oracle.com/javase/7/docs/technotes/guides/swing/enhancements-7.html)
* [IO and New IO](http://docs.oracle.com/javase/7/docs/technotes/guides/io/enhancements.html#7)
* [Networking](http://docs.oracle.com/javase/7/docs/technotes/guides/net/enhancements-7.0.html)
* [Security](http://docs.oracle.com/javase/7/docs/technotes/guides/security/enhancements-7.html)
* [Concurrency Utilities](http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/changes7.html)
* [Rich Internet Applications (RIA)/Deployment](http://docs.oracle.com/javase/7/docs/technotes/guides/jweb/clientJRECapabilitiesCheatSheet.html)
  + - [Requesting and Customizing Applet Decoration in Dragg able Applets](http://docs.oracle.com/javase/tutorial/deployment/applet/draggableApplet.html#decoration)
    - [Embedding JNLP File in Applet Tag](http://docs.oracle.com/javase/tutorial/deployment/deploymentInDepth/embeddingJNLPFileInWebPage.html)
    - [Deploying without Codebase](http://docs.oracle.com/javase/tutorial/deployment/deploymentInDepth/deployingWithoutCodebase.html)
    - [Handling Applet Initialization Status with Event Handlers](http://docs.oracle.com/javase/tutorial/deployment/applet/appletStatus.html)
* [Java 2D](http://docs.oracle.com/javase/7/docs/technotes/guides/2d/enhancements70.html)
* [Java XML](http://docs.oracle.com/javase/7/docs/technotes/guides/xml/enhancements.html) - JAXP, JAXB, and JAX-WS
* [Internationalization](http://docs.oracle.com/javase/7/docs/technotes/guides/intl/enhancements.7.html)
* [java.lang Package](http://docs.oracle.com/javase/7/docs/technotes/guides/lang/enhancements.html#7)
  + - [Multithreaded Custom Class Loaders in Java SE 7](http://docs.oracle.com/javase/7/docs/technotes/guides/lang/cl-mt.html)
* [Java Programming Language](http://docs.oracle.com/javase/7/docs/technotes/guides/language/enhancements.html#javase7)
  + - [Binary Literals](http://docs.oracle.com/javase/7/docs/technotes/guides/language/binary-literals.html)
    - [Strings in switch Statements](http://docs.oracle.com/javase/7/docs/technotes/guides/language/strings-switch.html)
    - [The try-with-resources Statement](http://docs.oracle.com/javase/7/docs/technotes/guides/language/try-with-resources.html)
    - [Catching Multiple Exception Types and Rethrowing Exceptions with Improved Type Checking](http://docs.oracle.com/javase/7/docs/technotes/guides/language/catch-multiple.html)
    - [Underscores in Numeric Literals](http://docs.oracle.com/javase/7/docs/technotes/guides/language/underscores-literals.html)
    - [Type Inference for Generic Instance Creation](http://docs.oracle.com/javase/7/docs/technotes/guides/language/type-inference-generic-instance-creation.html)
    - [Improved Compiler Warnings and Errors When Using Non-Reifiable Formal Parameters with Varargs Methods](http://docs.oracle.com/javase/7/docs/technotes/guides/language/non-reifiable-varargs.html)
* [Java Virtual Machine (JVM)](http://docs.oracle.com/javase/7/docs/technotes/guides/vm/)
  + - [Java Virtual Machine Support for Non-Java Languages](http://docs.oracle.com/javase/7/docs/technotes/guides/vm/multiple-language-support.html)
    - [Garbage-First Collector](http://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html)
    - [Java HotSpot Virtual Machine Performance Enhancements](http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html)
* [JDBC](http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/)

**Binary Literals**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

In Java SE 7, the integral types (byte, short, int, and long) can also be expressed using the binary number system. To specify a binary literal, add the prefix 0b or 0B to the number. The following examples show binary literals:

// An 8-bit 'byte' value:

byte aByte = (byte)0b00100001;

// A 16-bit 'short' value:

short aShort = (short)0b1010000101000101;

// Some 32-bit 'int' values:

int anInt1 = 0b10100001010001011010000101000101;

int anInt2 = 0b101;

int anInt3 = 0B101; // The B can be upper or lower case.

// A 64-bit 'long' value. Note the "L" suffix:

long aLong = 0b1010000101000101101000010100010110100001010001011010000101000101L;

Binary literals can make relationships among data more apparent than they would be in hexadecimal or octal. For example, each successive number in the following array is rotated by one bit:

public static final int[] phases = {

0b00110001,

0b01100010,

0b11000100,

0b10001001,

0b00010011,

0b00100110,

0b01001100,

0b10011000

}

In hexadecimal, the relationship among the numbers is not readily apparent:

public static final int[] phases = {

0x31, 0x62, 0xC4, 0x89, 0x13, 0x26, 0x4C, 0x98

}

You can use binary integral constants in code that you can verify against a specifications document, such as a simulator for a hypothetical 8-bit microprocessor:

public State decodeInstruction(int instruction, State state) {

if ((instruction & 0b11100000) == 0b00000000) {

final int register = instruction & 0b00001111;

switch (instruction & 0b11110000) {

case 0b00000000: return state.nop();

case 0b00010000: return state.copyAccumTo(register);

case 0b00100000: return state.addToAccum(register);

case 0b00110000: return state.subFromAccum(register);

case 0b01000000: return state.multiplyAccumBy(register);

case 0b01010000: return state.divideAccumBy(register);

case 0b01100000: return state.setAccumFrom(register);

case 0b01110000: return state.returnFromCall();

default: throw new IllegalArgumentException();

}

} else {

final int address = instruction & 0b00011111;

switch (instruction & 0b11100000) {

case 0b00100000: return state.jumpTo(address);

case 0b01000000: return state.jumpIfAccumZeroTo(address);

case 0b01000000: return state.jumpIfAccumNonzeroTo(address);

case 0b01100000: return state.setAccumFromMemory(address);

case 0b10100000: return state.writeAccumToMemory(address);

case 0b11000000: return state.callTo(address);

default: throw new IllegalArgumentException();

}

}

}

You can use binary literals to make a bitmap more readable:

public static final short[] HAPPY\_FACE = {

(short)0b0000011111100000,

(short)0b0000100000010000,

(short)0b0001000000001000,

(short)0b0010000000000100,

(short)0b0100000000000010,

(short)0b1000011001100001,

(short)0b1000011001100001,

(short)0b1000000000000001,

(short)0b1000000000000001,

(short)0b1001000000001001,

(short)0b1000100000010001,

(short)0b0100011111100010,

(short)0b0010000000000100,

(short)0b0001000000001000,

(short)0b0000100000010000,

(short)0b0000011111100000

}

# **Strings in switch Statements**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

In the JDK 7 release, you can use a String object in the expression of a switch statement:

public String getTypeOfDayWithSwitchStatement(String dayOfWeekArg) {

String typeOfDay;

switch (dayOfWeekArg) {

case "Monday":

typeOfDay = "Start of work week";

break;

case "Tuesday":

case "Wednesday":

case "Thursday":

typeOfDay = "Midweek";

break;

case "Friday":

typeOfDay = "End of work week";

break;

case "Saturday":

case "Sunday":

typeOfDay = "Weekend";

break;

default:

throw new IllegalArgumentException("Invalid day of the week: " + dayOfWeekArg);

}

return typeOfDay;

}

The switch statement compares the String object in its expression with the expressions associated with each case label as if it were using the String.equals method; consequently, the comparison of String objects in switch statements is case sensitive. The Java compiler generates generally more efficient bytecode from switch statements that use String objects than from chained if-then-else statements.

# **The try-with-resources Statement**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

The try-with-resources statement is a try statement that declares one or more resources. A resource is as an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.

public interface java.io.Closeable{

public abstract void close() throws java.io.IOException;

}

The following example reads the first line from a file. It uses an instance of BufferedReader to read data from the file. BufferedReader is a resource that must be closed after the program is finished with it:

static String readFirstLineFromFile(String path) throws IOException {

**try (BufferedReader br = new BufferedReader(new FileReader(path)))** {

return br.readLine();

}

}

In this example, the resource declared in the try-with-resources statement is a BufferedReader. The declaration statement appears within parentheses immediately after the try keyword. The classBufferedReader, in Java SE 7 and later, implements the interface java.lang.AutoCloseable. Because the BufferedReader instance is declared in a try-with-resource statement, it will be closed regardless of whether the try statement completes normally or abruptly (as a result of the method BufferedReader.readLine throwing an IOException).

Prior to Java SE 7, you can use a finally block to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly. The following example uses a finallyblock instead of a try-with-resources statement:

static String readFirstLineFromFileWithFinallyBlock(String path) throws IOException {

BufferedReader br = new BufferedReader(new FileReader(path));

try {

return br.readLine();

} finally {

if (br != null) br.close();

}

}

However, in this example, if the methods readLine and close both throw exceptions, then the method readFirstLineFromFileWithFinallyBlock throws the exception thrown from the finallyblock; the exception thrown from the try block is suppressed. In contrast, in the example readFirstLineFromFile, if exceptions are thrown from both the try block and the try-with-resources statement, then the method readFirstLineFromFile throws the exception thrown from the try block; the exception thrown from the try-with-resources block is suppressed. In Java SE 7 and later, you can retrieve suppressed exceptions; see the section [Suppressed Exceptions](http://docs.oracle.com/javase/7/docs/technotes/guides/language/try-with-resources.html#suppressed-exceptions) for more information.

You may declare one or more resources in a try-with-resources statement. The following example retrieves the names of the files packaged in the zip file zipFileName and creates a text file that contains the names of these files:

public static void writeToFileZipFileContents(String zipFileName, String outputFileName)

throws java.io.IOException {

java.nio.charset.Charset charset = java.nio.charset.Charset.forName("US-ASCII");

java.nio.file.Path outputFilePath = java.nio.file.Paths.get(outputFileName);

// Open zip file and create output file with try-with-resources statement

**try (**

**java.util.zip.ZipFile zf = new java.util.zip.ZipFile(zipFileName);**

**java.io.BufferedWriter writer = java.nio.file.Files.newBufferedWriter(outputFilePath, charset)**

**)** {

// Enumerate each entry

for (java.util.Enumeration entries = zf.entries(); entries.hasMoreElements();) {

// Get the entry name and write it to the output file

String newLine = System.getProperty("line.separator");

String zipEntryName = ((java.util.zip.ZipEntry)entries.nextElement()).getName() + newLine;

writer.write(zipEntryName, 0, zipEntryName.length());

}

}

}

In this example, the try-with-resources statement contains two declarations that are separated by a semicolon: ZipFile and BufferedWriter. When the block of code that directly follows it terminates, either normally or because of an exception, the close methods of the BufferedWriter and ZipFile objects are automatically called in this order. Note that the close methods of resources are called in the opposite order of their creation.

The following example uses a try-with-resources statement to automatically close a java.sql.Statement object:

public static void viewTable(Connection con) throws SQLException {

String query = "select COF\_NAME, SUP\_ID, PRICE, SALES, TOTAL from COFFEES";

**try (Statement stmt = con.createStatement())** {

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String coffeeName = rs.getString("COF\_NAME");

int supplierID = rs.getInt("SUP\_ID");

float price = rs.getFloat("PRICE");

int sales = rs.getInt("SALES");

int total = rs.getInt("TOTAL");

System.out.println(coffeeName + ", " + supplierID + ", " + price +

", " + sales + ", " + total);

}

} catch (SQLException e) {

JDBCTutorialUtilities.printSQLException(e);

}

}

The resource java.sql.Statement used in this example is part of the JDBC 4.1 and later API.

**Note**: A try-with-resources statement can have catch and finally blocks just like an ordinary try statement. In a try-with-resources statement, any catch or finally block is run after the resources declared have been closed.

## Suppressed Exceptions

An exception can be thrown from the block of code associated with the try-with-resources statement. In the example writeToFileZipFileContents, an exception can be thrown from the try block, and up to two exceptions can be thrown from the try-with-resources statement when it tries to close the ZipFile and BufferedWriter objects. If an exception is thrown from the try block and one or more exceptions are thrown from the try-with-resources statement, then those exceptions thrown from the try-with-resources statement are suppressed, and the exception thrown by the block is the one that is thrown by the writeToFileZipFileContents method. You can retrieve these suppressed exceptions by calling the Throwable.getSuppressed method from the exception thrown by thetry block.

## Classes That Implement the AutoCloseable or Closeable Interface

See the Javadoc of the [AutoCloseable](http://docs.oracle.com/javase/7/docs/api/java/lang/AutoCloseable.html) and [Closeable](http://docs.oracle.com/javase/7/docs/api/java/io/Closeable.html) interfaces for a list of classes that implement either of these interfaces. The Closeable interface extends the AutoCloseable interface. Theclose method of the Closeable interface throws exceptions of type IOException while the close method of the AutoCloseable interface throws exceptions of type Exception. Consequently, subclasses of the AutoCloseable interface can override this behavior of the close method to throw specialized exceptions, such as IOException, or no exception at all.

## Handling More Than One Type of Exception

* In Java SE 7 and later, a single catch block can handle more than one type of exception.
* This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

Consider the following example, which contains duplicate code in each of the catch blocks:

catch (IOException ex) {

logger.log(ex);

throw ex;

catch (SQLException ex) {

logger.log(ex);

throw ex;

}

In releases prior to Java SE 7, it is difficult to create a common method to eliminate the duplicated code because the variable ex has different types.

The following example, which is valid in Java SE 7 and later, eliminates the duplicated code:

catch (IOException|SQLException ex) {

logger.log(ex);

throw ex;

}

The catch clause specifies the types of exceptions that the block can handle, and each exception type is separated with a vertical bar (|).

**Note**: If a catch block handles more than one exception type, then the catch parameter is implicitly final. In this example, the catch parameter ex is final and therefore you cannot assign any values to it within the catch block.

Bytecode generated by compiling a catch block that handles multiple exception types will be smaller (and thus superior) than compiling many catch blocks that handle only one exception type each. Acatch block that handles multiple exception types creates no duplication in the bytecode generated by the compiler; the bytecode has no replication of exception handlers.

**Rethrowing Exceptions with More Inclusive Type Checking**

The Java SE 7 compiler performs more precise analysis of rethrown exceptions than earlier releases of Java SE. This enables you to specify more specific exception types in the throws clause of a method declaration.

Consider the following example:

static class FirstException extends Exception { }

static class SecondException extends Exception { }

public void rethrowException(String exceptionName) throws Exception {

try {

if (exceptionName.equals("First")) {

throw new FirstException();

} else {

throw new SecondException();

}

} catch (Exception e) {

throw e;

}

}

This examples's try block could throw either FirstException or SecondException. Suppose you want to specify these exception types in the throws clause of the rethrowException method declaration. In releases prior to Java SE 7, you cannot do so. Because the exception parameter of the catch clause, e, is type Exception, and the catch block rethrows the exception parameter e, you can only specify the exception type Exception in the throws clause of the rethrowException method declaration.

However, in Java SE 7, you can specify the exception types FirstException and SecondException in the throws clause in the rethrowException method declaration. The Java SE 7 compiler can determine that the exception thrown by the statement throw e must have come from the try block, and the only exceptions thrown by the try block can be FirstException andSecondException. Even though the exception parameter of the catch clause, e, is type Exception, the compiler can determine that it is an instance of either FirstException orSecondException:

public void rethrowException(String exceptionName)

throws **FirstException, SecondException** {

try {

// ...

}

catch (Exception e) {

throw e;

}

}

This analysis is disabled if the catch parameter is assigned to another value in the catch block. However, if the catch parameter is assigned to another value, you must specify the exception typeException in the throws clause of the method declaration.

In detail, in Java SE 7 and later, when you declare one or more exception types in a catch clause, and rethrow the exception handled by this catch block, the compiler verifies that the type of the rethrown exception meets the following conditions:

* The try block is able to throw it.
* There are no other preceding catch blocks that can handle it.
* It is a subtype or supertype of one of the catch clause's exception parameters.

The Java SE 7 compiler allows you to specify the exception types FirstException and SecondException in the throws clause in the rethrowException method declaration because you can rethrow an exception that is a supertype of any of the types declared in the throws.

In releases prior to Java SE 7, you cannot throw an exception that is a supertype of one of the catch clause's exception parameters. A compiler from a release prior to Java SE 7 generates the error, "unreported exception Exception; must be caught or declared to be thrown" at the statement throw e. The compiler checks if the type of the exception thrown is assignable to any of the types declared in the throws clause of the rethrowException method declaration. However, the type of the catch parameter e is Exception, which is a supertype, not a subtype, of FirstExceptionandSecondException.

# Underscores in Numeric Literals

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

In Java SE 7 and later, any number of underscore characters (\_) can appear anywhere between digits in a numerical literal. This feature enables you, for example, to separate groups of digits in numeric literals, which can improve the readability of your code.

For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator.

The following example shows other ways you can use the underscore in numeric literals:

long creditCardNumber = 1234\_5678\_9012\_3456L;

long socialSecurityNumber = 999\_99\_9999L;

float pi = 3.14\_15F;

long hexBytes = 0xFF\_EC\_DE\_5E;

long hexWords = 0xCAFE\_BABE;

long maxLong = 0x7fff\_ffff\_ffff\_ffffL;

byte nybbles = 0b0010\_0101;

long bytes = 0b11010010\_01101001\_10010100\_10010010;

You can place underscores only between digits; you cannot place underscores in the following places:

* At the beginning or end of a number
* Adjacent to a decimal point in a floating point literal
* Prior to an F or L suffix
* In positions where a string of digits is expected

The following examples demonstrate valid and invalid underscore placements (which are highlighted) in numeric literals:

**float pi1 = 3\_.1415F; // Invalid; cannot put underscores adjacent to a decimal point**

**float pi2 = 3.\_1415F; // Invalid; cannot put underscores adjacent to a decimal point**

**long socialSecurityNumber1**

**= 999\_99\_9999\_L; // Invalid; cannot put underscores prior to an L suffix**

int x1 = \_52; // This is an identifier, not a numeric literal

int x2 = 5\_2; // OK (decimal literal)

**int x3 = 52\_; // Invalid; cannot put underscores at the end of a literal**

int x4 = 5\_\_\_\_\_\_\_2; // OK (decimal literal)

**int x5 = 0\_x52; // Invalid; cannot put underscores in the 0x radix prefix**

**int x6 = 0x\_52; // Invalid; cannot put underscores at the beginning of a number**

int x7 = 0x5\_2; // OK (hexadecimal literal)

**int x8 = 0x52\_; // Invalid; cannot put underscores at the end of a number**

int x9 = 0\_52; // OK (octal literal)

int x10 = 05\_2; // OK (octal literal)

**int x11 = 052\_; // Invalid; cannot put underscores at the end of a number**

# Type Inference for Generic Instance Creation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context. This pair of angle brackets is informally called the diamond.

For example, consider the following variable declaration:

Map<String, List<String>> myMap = new HashMap<String, List<String>>();

In Java SE 7, you can substitute the parameterized type of the constructor with an empty set of type parameters (<>):

Map<String, List<String>> myMap = new HashMap<>();

Note that to take advantage of automatic type inference during generic class instantiation, you must specify the diamond. In the following example, the compiler generates an unchecked conversion warning because the HashMap() constructor refers to the HashMap raw type, not the Map<String, List<String>> type:

Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning

Java SE 7 supports limited type inference for generic instance creation; you can only use type inference if the parameterized type of the constructor is obvious from the context. For example, the following example does not compile:

List<String> list = new ArrayList<>();

list.add("A");

// The following statement should fail since addAll expects

// Collection<? extends String>

list.addAll(new ArrayList<>());

Note that the diamond often works in method calls; however, it is suggested that you use the diamond primarily for variable declarations.

In comparison, the following example compiles:

// The following statements compile:

List<? extends String> list2 = new ArrayList<>();

list.addAll(list2);

## Type Inference and Generic Constructors of Generic and Non-Generic Classes

Note that constructors can be generic (in other words, declare their own formal type parameters) in both generic and non-generic classes. Consider the following example:

class MyClass<X> {

<T> MyClass(T t) {

// ...

}

}

Consider the following instantiation of the class MyClass, which is valid in Java SE 7 and prior releases:

new MyClass<Integer>("")

This statement creates an instance of the parameterized type MyClass<Integer>; the statement explicitly specifies the type Integer for the formal type parameter, X, of the generic class MyClass<X>. Note that the constructor for this generic class contains a formal type parameter, T. The compiler infers the type String for the formal type parameter, T, of the constructor of this generic class (because the actual parameter of this constructor is a String object).

Compilers from releases prior to Java SE 7 are able to infer the actual type parameters of generic constructors, similar to generic methods. However, the compiler in Java SE 7 can infer the actual type parameters of the generic class being instantiated if you use the diamond (<>). Consider the following example, which is valid for Java SE 7 and later:

MyClass<Integer> myObject = new MyClass<>("");

In this example, the compiler infers the type Integer for the formal type parameter, X, of the generic class MyClass<X>. It infers the type String for the formal type parameter, T, of the constructor of this generic class.