Java 7 new features

**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

}

Example is given below.

**int** anInt1 = 0b10100001010001011010000101000101;  
**int** anInt2 = 0b101;  
**int** anInt3 = 0B101; *// The B can be upper or lower case.*System.***out***.println(anInt1); *// -1589272251*System.***out***.println(anInt2); *// 5*System.***out***.println(anInt3); *// 5*

# 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.

**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;  
  
System.***out***.println(creditCardNumber); *// 1234567890123456*System.***out***.println(pi); *// 3.1415*System.***out***.println(maxLong); *// 9223372036854775807*

# 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 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.

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 class BufferedReader, 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 methodBufferedReader.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 finally block 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 finally block; 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 closemethods 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.

# Catching Multiple Exception Types and Rethrowing Exceptions with Improved Type Checking

## 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. A catch 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 statementthrow e must have come from the try block, and the only exceptions thrown by the try block can be FirstException and SecondException. 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 or SecondException:

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 type Exception 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 isException, which is a supertype, not a subtype, of FirstException andSecondException.

# 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 theHashMap 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.

**Java NIO.2 –File Navigation Helpers**

**//Make a reference to a File**

**Path src = Paths.get(“/home/fred/readme.txt”);**

**Path dst = Paths.get(“/home/fred/copy\_readme.txt”);**

**//Make a reference to a path**

**Path src = Paths.get(“/home/fredSRC/”);**

**Path dst = Paths.get(“/home/fredDST/”);**

**//Navigation /home/fredSRC -> /home/fredSRC/tmp**

**Path tmpPath = src.resolve(“tmp”);**

**//Create a relative path from src -> ..**

**Path relativePath = tmpPath.relativize(src);**

**// Convert to old File Format for your legacy apps**

**File file = aPathPath.toFile();**

**Java NIO.2 Features –Files Helper Class**

Class java.nio.file.FilesExclusively static methods to operate on files, directories and other types of files

* 1. Files helper class is feature rich:Copy
  2. Create Directories
  3. Create Files
  4. Create Links
  5. Use of system “temp” directory
  6. Delete
  7. Attributes –Modified/Owner/Permissions/Size, etc.
  8. Read/Write

**Files.move(src, dst, StandardCopyOption.ATOMIC\_MOVE);**

**Files.copy(src, dst, StandardCopyOption.COPY\_ATTRIBUTES, StandardCopyOption.REPLACE\_EXISTING);**

Java NIO.2 Directories

DirectoryStream iterate over entriesScales to large directories

* 1. Uses less resources
  2. Smooth out response time for remote file systems
  3. Implements **Iterable**and **Closeable**for productivity
  4. Filtering supportBuild-in support for glob, regex and custom filters

**Path srcPath = Paths.get(“/home/fred/src”);**

**try (DirectoryStream<Path> dir =**

**srcPath.newDirectoryStream(“\*.java”)) {**

**for (Path file : dir)**

**System.out.println(file.getName());**

**}**

Concurrency APIs JSR 166y -Phasers

* 1. **Phaser**Barrier similar to **CyclicBarrier**and **CountDownLatch**
  2. Used for many threads to wait at common barrier pointFor example, use this to create N threads that you want to do something simultaneously –“start gun” metaphore
  3. How is Phaseran improvement?Dynamic add/remove “parties” to be sync’d
  4. Better deadlock avoidance
  5. Arrival “counting” and phase advance options, etc
  6. Termination api’s
  7. Tiering(tree structure)Rather than sync 100 threads, sync 2x50 then 2x.

**TransferQueue**interfaceExtension to **BlockingQueue**

* 1. Implemented by **LinkedTransferQueue**
  2. Additional Benefits:Adds methods:transfer(E e), tryTransfer(E e), tryTransfer(E e, long timeout), hasWaitingConsumer(), getWaitingConsumerCount()
  3. Allows for smarter queues to be built –sidestep the data structure if it’s known there are consumers waiting.
  4. **Fork Join Framework -JSR 166y**
  5. **ForkJoinPool**Service for running **ForkJoinTasks**
  6. **aFjp.execute(aTask); // async**
  7. **aFjp.invoke(aTask); // wait**
  8. **aFjp.submit(aTask); // async + future**
  9. **ForkJoinPool(); // default to platform**
  10. **ForkJoinPool(int n); // # concurrent threads**
  11. **ForJoinPool(n,aThreadFactory,exHandler,FIFOtasks); // Create your own thread handler, exception handler, and boolean on task ordering (default LIFO)**

**ForkJoinPool p = new ForkJoinPool();**

**MyTask mt = new MyTask(n); // implements compute**

**p.submit(mt);**

**while (!mt.isDone()) {/\*THUMPER!\*/ }**

**System.out.println(mt.get());**