JDK 7

New Features

Language Enhancement

`switch` on String

Before JDK 7, only integral types can be used as selector for switch-case statement. In JDK 7, you can use a `String` object as the selector. For example,

```
String day = "SAT";
switch (day) {
   case "MON": System.out.println("Monday"); break;
   case "TUE": System.out.println("Tuesday"); break;
   case "WED": System.out.println("Wednesday"); break;
   case "THU": System.out.println("Thursday"); break;
   case "FRI": System.out.println("Friday"); break;
   case "SAT": System.out.println("Saturday"); break;
   case "SUN": System.out.println("Sunday"); break;
   default: System.out.println("Invalid");
}
```

`String.equals()` method is used in comparison, which is case-sensitive. Java compiler can generate more efficient code than using nested if-then-else statement.

This feature is handy in handling options specified in command-line arguments, which are `String`s. For example (slightly neater code than using nested if-then-else statement),

```
// This program accepts three command-line options
//   -c : create
//   -v : verbose
//   -d : debug
// More than one options can be specified in any order.
public class SwitchOnString {
   public static void main(String[] args) {
      boolean create = false;
      boolean verbose = false;
      boolean debug = false;

      for (String arg: args) {
         switch (arg) {
            case "-c": create = true; break;
            case "-v": verbose = true; break;
            case "-d": debug = true; break;
            default:
               System.out.println("invalid option");
               System.exit(1);
         }
      }

      System.out.println("create: " + create);
      System.out.println("verbose: " + verbose);
      System.out.println("debug: " + debug);
   }
}
```

Binary Literals with prefix "`0b`"

In JDK 7, you can express literal values in binary with prefix '`0b`' (or '`0B`') for integral types (`byte`, `short`, `int` and `long`), similar to C/C++ language. Before JDK 7, you can only use octal values (with prefix '`0`') or hexadecimal values (with prefix '`0x`' or '`0X`').

You are also permitted to use underscore (_) to break the digits to improve the readability but you must start and end with a digit, e.g.,

```
int number1 = 0b01010000101000101101000010100010;
int number2 = 0b0101_0000_1010_0010_1101_0000_1010_0010;
int number3 = 2_123_456;  // break the digits with underscore
```

For example,

```
public class BinaryLiteralTest {
   public static void main(String[] args) {
      // Some 32-bit 'int' literal values
      int anInt1 = 0b0101_0000_1010_0010_1101_0000_1010_0010;
      int anInt2 = 0b0011_1000;

      // An 8-bit 'byte' literal value. By default, literal values are 'int'.
      // Need to cast to 'byte'
      byte aByte = (byte)0b0110_1101;

      // A 16-bit 'short' literal value
      short aShort = (short)0b0111_0101_0000_0101;

      // A 64-bit 'long' literal value. Long literals requires suffix "L".
      long aLong = 0b1000_0101_0001_0110_1000_0101_0000_1010_0010_1101_0100_0101_1010_0001_0100_0101L;

      // Formatted output: "%d" for integer in decimal, "%x" in hexadecimal, "%o" in octal.
      // Take note that "%b" prints true or false (for null), NOT binary.
      System.out.printf("%d(%x)(%o)(%b)\n", anInt1, anInt1, anInt1, anInt1);
      System.out.printf("%d(%x)(%o)(%b)\n", aByte, aByte, aByte, aByte);
   }
}
```

```
1352847522(50a2d0a2)(12050550242)(true)
109(6d)(155)(true)
```

Underscore for Numeric Literals

In JDK 7, you could insert underscore(s) '_' in between the digits in an numeric literals (integral and floating-point literals) to improve *readability*. For example,

```
int anInt = 0b10101000_01010001_01101000_01010001;
double aDouble = 3.1415_9265;
float  aFloat = 3.14_15_92_65f;
```

Catching Multiple Exception Types

In JDK 7, a single `catch` block can handle more than one exception types.

For example, before JDK 7, you need two `catch` blocks to catch two exception types although both perform identical task:

```
try {
   ......
} catch(ClassNotFoundException ex) {
   ex.printStackTrace();
} catch(SQLException ex) {
   ex.printStackTrace();
}
```

In JDK 7, you could use one single `catch` block, with exception types separated by '`|`'.

```
try {
   ......
} catch(ClassNotFoundException|SQLException ex) {
   ex.printStackTrace();
}
```

[TODO] A complete example on file IO.

The `try`-with-resources Statement

For example, before JDK 7, we need to use a `finally` block, to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly. The code is messy!

```
import java.io.*;
// Copy from one file to another file character by character.
// Pre-JDK 7 requires you to close the resources using a finally block.
public class FileCopyPreJDK7 {
   public static void main(String[] args) {
      BufferedReader in = null;
      BufferedWriter out = null;
      try {
         in  = new BufferedReader(new FileReader("in.txt"));
         out = new BufferedWriter(new FileWriter("out.txt"));
         int charRead;
         while ((charRead = in.read()) != -1) {
            System.out.printf("%c ", (char)charRead);
            out.write(charRead);
         }
      } catch (IOException ex) {
         ex.printStackTrace();
      } finally {            // always close the streams
         try {
            if (in != null) in.close();
            if (out != null) out.close();
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }

      try {
         in.read();   // Trigger IOException: Stream closed
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

JDK 7 introduces a `try`-with-resources statement, which ensures that each of the resources in `try(resourses)` is closed at the end of the statement. This results in cleaner codes.

```
import java.io.*;
// Copy from one file to another file character by character.
// JDK 7 has a try-with-resources statement, which ensures that
// each resource opened in try() is closed at the end of the statement.
public class FileCopyJDK7 {
   public static void main(String[] args) {
      try (BufferedReader in  = new BufferedReader(new FileReader("in.txt"));
           BufferedWriter out = new BufferedWriter(new FileWriter("out.txt"))) {
         int charRead;
         while ((charRead = in.read()) != -1) {
            System.out.printf("%c ", (char)charRead);
            out.write(charRead);
         }
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

Type Inference for Generic Instance Creation

```
import java.util.*;
public class JDK7GenericTest {
   public static void main(String[] args) {
      // Pre-JDK 7
      List<String> lst1 = new ArrayList<String>();
      // JDK 7 supports limited type inference for generic instance creation
      List<String> lst2 = new ArrayList<>();

      lst1.add("Mon");
      lst1.add("Tue");
      lst2.add("Wed");
      lst2.add("Thu");

      for (String item: lst1) {
         System.out.println(item);
      }
```

```
         for (String item: lst2) {
            System.out.println(item);
         }
      }
   }
```

Others

[TODO]

REFERENCES & RESOURCES

- JDK 7 Documentation @ http://download.oracle.com/javase/7/docs/.
- Jeff Friesen, "Exploring JDK 7, Part 1: New Language Features" @ http://www.informit.com/articles/article.aspx?p=1592962&seqNum=3.

Latest version tested: JDK 1.7.0
Last modified: August, 2011