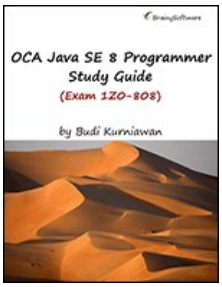

OCA Java SE 8 Programmer Study Guide (Exam 1Z0-808)

by Budi Kurniawan

Chapter 4: Core Classes

Overview

Before discussing other features of object-oriented programming (OOP), let's examine several important classes that are commonly used in Java. These classes are included in the Java core libraries that come with the JDK. Mastering them will help you understand the examples that accompany the next OOP lessons.

The most prominent class of all is definitely **java.lang.Object**. However, it is hard to talk about this class without first covering inheritance, which I will do in Chapter 6, "Inheritance." Therefore, **java.lang.Object** is only discussed briefly in this chapter. Right now I will concentrate on classes that you can use in your programs. I will start with **java.lang.String** and other types of strings: **java.lang.StringBuffer** and **java.lang.StringBuilder**. Then, I will discuss the **java.lang.System** class.

Note When describing a method in a Java class, presenting the method signature always helps. A method often takes as parameters objects whose classes belong to different packages than the method's class. Or, it may return a type from a different package than its class. For clarity, fully qualified names will be used for classes from different packages. For example, here is the signature of the **toString** method of **java.lang.Object:**

```
public String toString()
```

A fully qualified name for the return type is not necessary because the return type **String** is part of the same package as **java.lang.Object**. On the other hand, the signature of the **toString** method in **java.util.Scanner** uses a fully qualified name because the **Scanner** class is part of a different package (**java.util**).

```
public java.lang.String toString()
```

4.1 java.lang.Object

The **java.lang.Object** class represents a Java object. In fact, all classes are direct or indirect descendants of this class. Since we have not learned inheritance (which is only given in Chapter 6, "Inheritance"), the word descendant probably makes no sense to you. Therefore, we will briefly discuss the method in this class and revisit this class in Chapter 6.

Table 4.1 shows the methods in the **Object** class.

Table 4.1: java.lang.Object methods

Method	Description
clone	Creates and returns a copy of this object. A class implements this method to support object cloning.
equals	Compares this object with the passed-in object. A class must implement this method to provide a means to compare the contents of its instances.
finalize	Called by the garbage collector on an object that is about to be garbage-collected. In theory a subclass can override this method to dispose of system resources or to perform other cleanup. However, performing the aforesaid operations should be done somewhere else and you should not touch this method.
getClass	Returns a **java.lang.Class** object of this object. See the section "java.lang.Class" for more information on the **Class** class.
hashCode	Returns a hash code value for this object.
toString	Returns the description of this object.
wait, notify, notifyAll	Used in multithreaded programming in pre-5 Java. Should not be used directly in Java 5 or later. Instead, use the Java concurrency utilities.

4.2 java.lang.String

I have not seen a serious Java program that does not use the **java.lang.String** class. It is one of the most often used classes and definitely one of the most important.

A **String** object represents a string, i.e. a piece of text. You can also think of a **String** as a sequence of Unicode characters. A **String** object can consists of any number of characters. A **String** that has zero character is called an empty **String. String** objects are constant. Once they are created, their values cannot be changed. Because of this, **String** instances are said to be immutable. And, because they are immutable, they can be safely shared.

You could construct a **String** object using the **new** keyword, but this is not a common way to create a **String**. Most often, you assign a string literal to a **String** reference variable. Here is an example:

```
String s = "Java is cool";
```

This produces a **String** object containing "Java is cool" and assigns a reference to it to **s**. It is the same as the following.

```
String message = new String("Java is cool");
```

However, assigning a string literal to a reference variable works differently from using the **new** keyword. If you use the **new** keyword, the JVM will always create a new instance of **String**. With a string literal, you get an identical **String** object, but the object is not always new. It may come from a pool if the string "Java is cool" has been created before.

Thus, using a string literal is better because the JVM can save some CPU cycles spent on constructing a new instance. Because of this, you seldom use the **new** keyword when creating a **String** object. The **String** class's constructors can be used if you have specific needs, such as converting a character array into a **String**.

Comparing Two Strings

String comparison is one of the most useful operations in Java programming. Consider the following code.

```
String s1 = "Java";
String s2 = "Java";
if (s1 == s2) {
    ...
}
```

Here, **(s1 == s2)** evaluates to **true** because **s1** and **s2** reference the same instance. On the other hand, in the following code **(s1 == s2)** evaluates to **false** because **s1** and **s2** reference different instances:

```
String s1 = new String("Java");
String s2 = new String("Java");
if (s1 == s2) {
    ...
}
```

This shows the difference between creating **String** objects by writing a string literal and by using the **new** keyword.

Comparing two **String** objects using the **==** operator is of little use because you are comparing the addresses referenced by two variables. Most of the time, when comparing two **String** objects, you want to know whether the values of the two objects are the same. In this case, you need to use the **String** class's **equals** method.

```
String s1 = "Java";
if (s1.equals("Java")) // returns true.
```

And, sometimes you see this style.

```
if ("Java".equals(s1))
```

In **(s1.equals("Java"))**, the **equals** method on **s1** is called. If **s1** is null, the expression will generate a runtime error. To be safe, you have to make sure that **s1** is not null, by first checking if the reference variable is null.

```
if (s1 != null && s1.equals("Java"))
```

If **s1** is null, the **if** statement will return **false** without evaluating the second expression because the AND operator **&&** will not try to evaluate the right hand operand if the left hand operand evaluates to **false**.

In **("Java".equals(s1))**, the JVM creates or takes from the pool a **String** object containing "Java" and calls its **equals** method. No nullity checking is required here because "Java" is obviously not null. If **s1** is null, the expression simply returns **false**. Therefore, these two lines of code have the same effect.

```
if (s1 != null && s1.equals("Java"))

if ("Java".equals(s1))
```

String Literals

Because you always work with **String** objects, it is important to understand the rules for working with string literals.

First of all, a string literal starts and ends with a double quote ("). Second, it is a compile error to change line before the closing double quote. For example, this code snippet will raise a compile error.

```
String s2 = "This is an important
        point to note";
```

You can compose long string literals by using the plus sign to concatenate two string literals.

```
String s1 = "Java strings " + "are important";
String s2 = "This is an important " +
        "point to note";
```

You can concatenate a String with a primitive or another object. For instance, this line of code concatenates a **String** and an integer.

```
String s3 = "String number " + 3;
```

If an object is concatenated with a String, the **toString** method of the former will be called and the result used in the concatenation.

Escaping Certain Characters

You sometimes need to use special characters in your strings such as carriage return (CR) and linefeed (LF). In other occasions, you may want to have a double quote character in your string. In the case of CR and LF, it is not possible to input these characters because pressing Enter changes lines. A way to include special characters is to escape them, i.e. use the character replacement for them.

Here are some escape sequences:

```
      \u          /* a Unicode character
      \b          /* \u0008: backspace BS */
      \t          /* \u0009: horizontal tab HT */
      \n          /* \u000a: linefeed LF */
      \f          /* \u000c: form feed FF */
      \r          /* \u000d: carriage return CR */
      \"          /* \u0022: double quote " */
      \'          /* \u0027: single quote ' */
      \\          /* \u005c: backslash \ */
```

For example, the following code includes the Unicode character 0122 at the end of the string.

```
String s = "Please type this character \u0122";
```

To obtain a **String** object whose value is John "The Great" Monroe, you escape the double quotes:

```
String s = "John \"The Great\" Monroe";
```

Switching on A String

Starting from Java 7 you can use the **switch** statement with a String. Recall the syntax of the **switch** statement given in Chapter 2, "Statements."

```
switch(expression) {
case value_1 :
    statement(s);
    break;
case value_2 :
    statement(s);
    break;
  .
  .
  .
case value_n :
    statement(s);
    break;
default:
    statement(s);
}
```

Here is an example of using the **switch** statement on a String.

```
String input = ...;
switch (input) {
```

```
case "one" :
    System.out.println("You entered 1.");
    break;
case "two" :
    System.out.println("You entered 2.");
    break;
default:
    System.out.println("Invalid value.");
}
```

The String Class's Constructors

The **String** class provides a number of constructors. These constructors allow you to create an empty string, a copy of another string, and a **String** from an array of chars or bytes. Use the constructors with caution as they always create a new instance of **String**.

Note Arrays are discussed in Chapter 5, "Arrays."

`public String()`

- Creates an empty string.

`public String(String original)`

- Creates a copy of the original string.

`public String(char[] value)`

- Creates a **String** object from an array of chars.

`public String(byte[] bytes)`

- Creates a **String** object by decoding the bytes in the array using the computer's default encoding.

`public String(byte[] bytes, String encoding)`

- Creates a **String** object by decoding the bytes in the array using the specified encoding.

The String Class's Methods

The **String** class provides methods for manipulating the value of a **String**. However, since **String** objects are immutable, the result of the manipulation is always a new **String** object.

Here are some of the more useful methods.

`public char charAt(int index)`

- Returns the char at the specified index. For example, the following code returns 'J'.

```
  "Java is cool".charAt(0)
```

`public String concat(String s)`

- Concatenates the specified string to the end of this **String** and return the result. For example, **"Java ".concat("is cool")** returns "Java is cool".

`public boolean equals(String anotherString)`

- Compares the value of this **String** and *anotherString* and returns **true** if the values match.

`public boolean endsWith(String suffix)`

- Tests if this **String** ends with the specified suffix.

`public int indexOf(String substring)`

- Returns the index of the first occurrence of the specified substring. If no match is found, returns -1. For instance, the following code returns 8.

```
"Java is cool".indexOf("cool")
```

```
public int indexOf(String substring, int fromIndex)
```

- Returns the index of the first occurrence of the specified substring starting from the specified index. If no match is found, returns -1.

```
public int lastIndexOf(String substring)
```

- Returns the index of the last occurrence of the specified substring. If no match is found, returns -1.

```
public int lastIndexOf(String substring, int fromIndex)
```

- Returns the index of the last occurrence of the specified substring starting from the specified index. If no match is found, returns -1. For example, the following expression returns 3.

```
"Java is cool".lastIndexOf("a")
```

```
public String substring(int beginIndex)
```

- Returns a substring of the current string starting from the specified index. For instance, **"Java is cool".substring(8)** returns "cool".

```
public String substring(int beginIndex, int endIndex)
```

- Returns a substring of the current string starting from *beginIndex* to *endIndex*. For example, the following code returns "is":

```
"Java is cool".substring(5, 7)
```

```
public String replace(char oldChar, char newChar)
```

- Replaces every occurrence of *oldChar* with *newChar* in the current string and returns the new **String. "dingdong".replace('d', 'k')** returns "kingkong".

```
public int length()
```

- Returns the number of characters in this **String**. For example, **"Java is cool".length()** returns 12. Prior to Java 6, this method was often used to test if a **String** was empty. However, the **isEmpty** method is preferred because it's more descriptive.

```
public boolean isEmpty()
```

- Returns true is the string is empty (contains no characters).

```
public String[] split(String regEx)
```

- Splits this **String** around matches of the specified regular expression. For example, **"Java is cool".split(" ")** returns an array of three **String**s. The first array element is "Java", the second "is", and the third "cool".

```
public boolean startsWith(String prefix)
```

- Tests if the current string starts with the specified prefix.

```
public char[] toCharArray()
```

- Converts this string to an array of chars.

```
public String toLowerCase()
```

- Converts all the characters in the current string to lower case. For instance, **"Java is cool".toLowerCase()** returns "java is cool".

```
public String toUpperCase()
```

- Converts all the characters in the current string to upper case. For instance, **"Java is cool".toUpperCase()** returns "JAVA IS COOL".

```
public String trim()
```

- Trims the trailing and leading white spaces and returns a new string. For example, **" Java ".trim()** returns "Java".

In addition, there are static methods such as **valueOf** and **format**. The **valueOf** method converts a primitive, a char array, or an instance of **Object** into a string representation and there are nine overloads of this method.

```
public static String valueOf(boolean value)
public static String valueOf(char value)
public static String valueOf(char[] value)
public static String valueOf(char[] value, int offset, int length)
public static String valueOf(double value)
public static String valueOf(float value)
public static String valueOf(int value)
public static String valueOf(long value)
public static String valueOf(Object value)
```

For example, the following code returns the string "23"

```
String.valueOf(23);
```

The **format** method allows you to pass an arbitrary number of parameters. Here is its signature.

```
public static String format(String format, Object... args)
```

This method returns a **String** formatted using the specified format string and arguments. The format pattern must follow the rules specified in the **java.util.Formatter** class and you can read them in the JavaDoc for the **Formatter** class. A brief description of these rules are as follows.

To specify an argument, use the notation **%s**, which denotes the next argument in the array. For example, the following is a method call to the **printf** method.

```
String firstName = "John";
String lastName = "Adams";
System.out.format("First name: %s. Last name: %s",
        firstName, lastName);
```

This prints the following string to the console:

```
First name: John. Last name: Adams
```

Without varargs, you have to do it in a more cumbersome way.

```
String firstName = "John";
String lastName = "Adams";
System.out.println("First name: " + firstName +
        ". Last name: " + lastName);
```

Note The **printf** method in **java.io.PrintStream** is an alias for **format**.

The formatting example described here is only the tip of the iceberg. The formatting feature is much more powerful than that and you are encouraged to explore it by reading the Javadoc for the **Formatter** class.

4.3 java.lang.StringBuffer and java.lang.StringBuilder

String objects are immutable and are not suitable to use if you need to append or insert characters into them because string operations on **String** always create a new **String** object. For append and insert, you'd be better off using the **java.lang.StringBuffer** or **java.lang.StringBuilder** class. Once you're finished manipulating the string, you can convert a **StringBuffer** or **StringBuilder** object to a **String**.

Until JDK 1.4, the **StringBuffer** class was solely used for mutable strings. Methods in **StringBuffer** are synchronized, making **StringBuffer** suitable for use in multithreaded environments. However, the price for synchronization is performance. JDK 5 added the **StringBuilder** class, which is the unsynchronized version of **StringBuffer. StringBuilder** should be chosen over **StringBuffer** if you do not need synchronization.

The rest of this section will use **StringBuilder**. However, the discussion is also applicable to **StringBuffer** as both **StringBuilder** and **StringBuffer** shares similar constructors and methods.

StringBuilder Class's Constructors

The **StringBuilder** class has four constructors. You can pass a **java.lang.CharSequence**, a **String**, or an **int**.

```
public StringBuilder()
public StringBuilder(CharSequence seq)
public StringBuilder(int capacity)
public StringBuilder(String string)
```

If you create a **StringBuilder** object without specifying the capacity, the object will have a capacity of 16 characters. If its content exceeds 16 characters, it will grow automatically. If you know that your string will be longer than 16 characters, it is a good idea to allocate enough capacity as it takes time to increase a **StringBuilder**'s capacity.

StringBuilder Class's Methods

The **StringBuilder** class has several methods. The main ones are **capacity, length, append**, and **insert**.

```
public int capacity()
```

- Returns the capacity of the **StringBuilder** object.

```
public int length()
```

- Returns the length of the string the **StringBuilder** object stores. The value is less than or equal to the capacity of the **StringBuilder**.

```
public StringBuilder append(String string)
```

- Appends the specified **String** to the end of the contained string. In addition, **append** has various overloads that allow you to pass a primitive, a char array, and an **java.lang.Object** instance.
- For example, examine the following code.

  ```
  StringBuilder sb = new StringBuilder(100);
  sb.append("Matrix ");
  sb.append(2);
  ```

  After the last line, the content of **sb** is "Matrix 2".

- An important point to note is that the **append** methods return the **StringBuilder** object itself, the same object on which **append** is invoked. As a result, you can chain calls to **append**.

  ```
  sb.append("Matrix ").append(2);
  ```

```
public StringBuilder insert(int offset, String string)
```

- Inserts the specified string at the position indicated by *offset*. In addition, **insert** has various overloads that allow you to pass primitives and a **java.lang.Object** instance. For example,

  ```
  StringBuilder sb2 = new StringBuilder(100);
  sb2.append("night");
  sb2.insert(0, 'k'); // value = "knight"
  ```

- Like **append, insert** also returns the current **StringBuilder** object, so chaining **insert** is also permitted.

```
public String toString()
```

- Returns a **String** object representing the value of the **StringBuilder**.

4.4 Primitive Wrappers

For the sake of performance, not everything in Java is an object. There are also primitives, such as **int, long, float, double**, etc. When working with both primitives and objects, there are often circumstances that necessitate primitive to object conversions and vice versa. For example, a **java.util.Collection** object can be used to store objects, not primitives. If you want to store primitive values in a **Collection**, they must be converted to objects first.

The **java.lang** package has several classes that function as primitive wrappers. They are **Boolean, Character, Byte, Double, Float, Integer, Long**, and **Short. Byte, Double, Float, Integer, Long**, and **Short** share similar methods, therefore only **Integer** will be discussed here. You should consult the Javadoc for information on the others.

The following sections discuss the wrapper classes in detail.

java.lang.Integer

The **java.lang.Integer** class wraps an **int**. The **Integer** class has two static final fields of type **int: MIN_VALUE** and **MAX_VALUE. MIN_VALUE** contains the minimum possible value for an **int** (-2^{31}) and **MAX_VALUE** the maximum possible value for an **int** ($2^{31} - 1$).

The **Integer** class has two constructors:

```
public Integer(int value)
public Integer(String value)
```

For example, this code constructs two **Integer** objects.

```
Integer i1 = new Integer(12);
Integer i2 = new Integer("123");
```

Integer has the no-arg **byteValue, doubleValue, floatValue, intValue, longValue**, and **shortValue** methods that convert the wrapped value to a **byte, double, float, int, long**, and **short**, respectively. In addition, the **toString** method converts the value to a **String**.

There are also static methods that you can use to parse a **String** to an **int (parseInt)** and convert an **int** to a **String (toString)**. The signatures of the methods are as follows.

```
public static int parseInt(String string)
public static String toString(int i)
```

java.lang.Boolean

The **java.lang.Boolean** class wraps a **boolean**. Its static final fields **FALSE** and **TRUE** represents a **Boolean** object that wraps the primitive value **false** and a **Boolean** object wrapping the primitive value **true**, respectively.

You can construct a **Boolean** object from a **boolean** or a **String**, using one of these constructors.

```
public Boolean(boolean value)
public Boolean(String value)
```

For example:

```
Boolean b1 = new Boolean(false);
Boolean b2 = new Boolean("true");
```

To convert a **Boolean** to a **boolean**, use its **booleanValue** method:

```
public boolean booleanValue()
```

In addition, the static method **valueOf** parses a **String** to a **Boolean** object.

```
public static Boolean valueOf(String string)
```

And, the static method **toString** returns the string representation of a **boolean**.

```
public static String toString(boolean boolean)
```

java.lang. Character

The **Character** class wraps a **char**. There is only one constructor in this class:

```
public Character(char value)
```

To convert a **Character** object to a **char**, use its **charValue** method.

```
public char charValue()
```

There are also a number of static methods that can be used to manipulate characters.

```
public static boolean isDigit(char ch)
```

- Determines if the specified argument is one of these: '1', '2', '3', '4', '5', '6', '7', '8', '9', '0'.

```
public static char toLowerCase(char ch)
```

- Converts the specified char argument to its lower case.

```
public static char toUpperCase(char ch)
```

- Converts the specified char argument to its upper case.

4.5 java.lang.Class

One of the members of the **java.lang** package is a class named **Class**. Every time the JVM creates an object, it also creates a **java.lang.Class** object that describes the type of the object. All instances of the same class share the same **Class** object. You can obtain the **Class** object by calling the **getClass** method of the object. This method is inherited from **java.lang.Object**.

For example, the following code creates a **String** object, invokes the **getClass** method on the **String** instance, and then invokes the **getName** method on the **Class** object.

```
String country = "Fiji";
Class myClass = country.getClass();
System.out.println(myClass.getName()); // prints java.lang.String
```

As it turns out, the **getName** method returns the fully qualified name of the class represented by a **Class** object.

The **Class** class also brings the possibility of creating an object without using the **new** keyword. You achieve this by using the two methods of the **Class** class, **forName** and **newInstance**.

```
public static Class forName(String className)
public Object newInstance()
```

The static **forName** method creates a **Class** object of the given class name. The **newInstance** method creates a new instance of a class.

The **ClassDemo** in Listing 4.1 uses **forName** to create a **Class** object of the **app04.Test** class and create an instance of the **Test** class. Since **newInstance** returns a **java.lang.Object** object, you need to downcast it to its original type.

Listing 4.1: The ClassDemo class

```
package app04;
public class ClassDemo {
    public static void main(String[] args) {
        String country = "Fiji";
        Class myClass = country.getClass();
        System.out.println(myClass.getName());
        Class klass = null;
        try {
            klass = Class.forName("app04.Test");
        } catch (ClassNotFoundException e) {
        }

        if (klass != null) {
            try {
                Test test = (Test) klass.newInstance();
                test.print();
            } catch (IllegalAccessException e) {
            } catch (InstantiationException e) {
            }
        }
    }
}
```

Do not worry about the **try … catch** blocks as they will be explained in Chapter 8, "Error Handling."

You might want to ask this question, though. Why would you want to create an instance of a class using **forName** and **newInstance**, when using the **new** keyword is shorter and easier? The answer is because there are circumstances whereby

the name of the class is not known when you are writing the program.

4.6 java.lang.System

The **System** class is a final class that exposes useful static fields and static methods that can help you with common tasks.

The three fields of the **System** class are **out, in**, and **err:**

```
public static final java.io.PrintStream out;
public static final java.io.InputStream in;
public static final java.io.PrintStream err;
```

The **out** field represents the standard output stream which by default is the same console used to run the running Java application. Note that you can use the **out** field to write messages to the console. You will often write the following line of code:

```
System.out.print(message);
```

where *message* is a **String** object. However, **PrintStream** has many **print** method overloads that accept different types, so you can pass any primitive type to the **print** method:

```
System.out.print(12);
System.out.print('g');
```

In addition, there are **println** methods that are equivalent to **print**, except that **println** adds a line terminator at the end of the argument.

Note also that because **out** is static, you can access it by using this notation: **System.out**, which returns a **java.io.PrintStream** object. You can then access the many methods on the **PrintStream** object as you would methods of other objects: **System.out.print, System.out.format**, etc.

The **err** field also represents a **PrintStream** object, and by default the output is channeled to the console from where the current Java program was invoked. However, its purpose is to display error messages that should get immediate attention of the user.

For example, here is how you can use **err:**

```
System.err.println("You have a runtime error.");
```

The **in** field represents the standard input stream. You can use it to accept keyboard input. For example, the **getUserInput** method in Listing 4.2 accepts the user input and returns it as a String:

Listing 4.2: The InputDemo class

```
package app04;
import java.io.IOException;

public class InputDemo {
    public String getUserInput() {
        StringBuilder sb = new StringBuilder();
        try {
            char c = (char) System.in.read();
            while (c != '\r' && c != '\n') {
                sb.append(c);
                c = (char) System.in.read();
            }
        } catch (IOException e) {
        }
        return sb.toString();
    }

    public static void main(String[] args) {
        InputDemo demo = new InputDemo();
        String input = demo.getUserInput();
        System.out.println(input);
    }
}
```

However, an easier way to receive keyboard input is to use the **java.util.Scanner** class.

The **System** class has many useful methods, all of which are static. Some of the more important ones are listed here.

```
public static void arraycopy(Object source, int sourcePos,
        Object destination, int destPos, int length)
```

- This method copies the content of an array *(source)* to another array *(destination)*, beginning at the specified position, to the specified position of the destination array. For example, the following code uses **arraycopy** to copy the contents of **array1** to **array2**.

```
int[] array1 = {1, 2, 3, 4};
int[] array2 = new int[array1.length];
System.arraycopy(array1, 0, array2, 0, array1.length);
```

```
public static void exit(int status)
```

- Terminates the running program and the current JVM. You normally pass 0 to indicate that a normal exit and a nonzero to indicate there has been an error in the program prior to calling this method.

```
public static long currentTimeMillis()
```

- Returns the computer time in milliseconds. The value represents the number of milliseconds that has elapsed since January 1, 1970 UTC.
  Prior to Java 8, **currentTimeMillis** was used to time an operation. In Java 8 and later, you can use the **java.time.Instant** class, instead.

```
public static long nanoTime()
```

- This method is similar to **currentTimeMillis**, but with nanosecond precision.

```
public static String getProperty(String key)
```

- This method returns the value of the specified property. It returns **null** if the specified property does not exist. There are system properties and there are user-defined properties. When a Java program runs, the JVM provides values that may be used by the program as properties.

  Each property comes as a key/value pair. For example, the **os.name** system property provides the name of the operating system running the JVM. Also, the directory name from which the application was invoked is provided by the JVM as a property named **user.dir**. To get the value of the **user.dir** property, you use:

```
System.getProperty("user.dir");
```

Table 4.2 lists the system properties.

Table 4.2: Java system properties

System property	Description
java.version	Java Runtime Environment version
java.vendor	Java Runtime Environment vendor
java.vendor.url	Java vendor URL
java.home	Java installation directory
java.vm.specification.version	Java Virtual Machine specification version
java.vm.specification.vendor	Java Virtual Machine specification vendor
java.vm.specification.name	Java Virtual Machine specification name
java.vm.version	Java Virtual Machine implementation version
java.vm.vendor	Java Virtual Machine implementation vendor
java.vm.name	Java Virtual Machine implementation name
java.specification.version	Java Runtime Environment specification version
java.specification.vendor	Java Runtime Environment specification vendor
java.specification.name	Java Runtime Environment specification name
java.class.version	Java class format version number
java.class.path	Java class path
java.library.path	List of paths to search when loading libraries
java.io.tmpdir	Default temp file path
java.compiler	Name of JIT compiler to use

Table 4.2: Java system properties

System property	Description
java.ext.dirs	Path of extension directory or directories
os.name	Operating system name
os.arch	Operating system architecture
os.version	Operating system version
file.separator	File separator ("/" on UNIX)
path.separator	Path separator (":" on UNIX)
line.separator	Line separator ("\n" on UNIX)
user.name	User's account name
user.home	User's home directory
user.dir	User's current working directory

```
public static void setProperty(String property, String newValue)
```

- You use **setProperty** to create a user-defined property or change the value of the current property. For instance, you can use this code to create a property named **password:**

```
System.setProperty("password", "tarzan");
```

- And, you can retrieve it by using **getProperty**:

```
System.getProperty("password")
```

- For instance, here is how you change the **user.name** property.

```
  System.setProperty("user.name", "tarzan");
```

```
public static String getProperty(String key, String default)
```

- This method is similar to the single argument **getProperty** method, but returns a default value if the specified property does not exist.

```
public static java.util.Properties getProperties()
```

- This method returns all system properties. The return value is a **java.util.Properties** object. The **Properties** class is a subclass of **j ava.util.Hashtable**.

- For example, the following code uses the **list** method of the **Properties** class to iterate and display all system properties on the console.

```
java.util.Properties properties = System.getProperties();
properties.list(System.out);
```

Self Test

1 Question

Consider the following code snippet.

```
public static void main(String[] args) {
    String s1 = "start";
    String s2 = "start";
    System.out.print(s1 == s2);
    System.out.print(" ... ");

    String s3 = new String("finish");
    String s4 = new String("finish");
    System.out.println(s3 == s4);
}
```

What is printed on the console when the **main** method is invoked?

A. true … false

B. true … true

C. false … false

D. false … true

2 Question

The following code showcases two different ways of splitting a string.

```
String input = "Windows,Linux,,Mac OSX";
StringTokenizer tokenizer = new StringTokenizer(input, ",");
System.out.print(tokenizer.countTokens() + " ");
String[] tokens = input.split(",");
System.out.print(tokens.length);
```

What does this code print on the console?

A. 3 4

B. 4 4

C. 3 3

D. 4 3

3 Question

Given

```
String input = "\tContango ";
System.out.println(input.trim().length());
```

What does this code print on the console?

A. 8

B. 9

C. 10

D. 11

4 Question

Given

```
class FileUtil {
    public static String getFileExtension(String fileName) {
        int index = fileName.lastIndexOf(".");
        return fileName.substring(index);
    }
    public static void main(String[] args) {
        String fileName = "market.pdf";
        System.out.println(getFileExtension(fileName));
    }
}
```

What does the code print?

A. pdf

B. .pdf

C. market

D. market.

5 Question

Consider this code snippet:

```
StringBuilder sb = new StringBuilder("Storage ");
sb.insert(0, "File");
sb.append("almost full");
System.out.print(sb);
```

What will be printed if the code is executed?

A. File Storage almost full

B. FileStorage almost full

C. java.lang.StringBuilder

D. The fully-qualified class name for StringBuilder followed by a random string.

6 Question ?

Given

```
1.  public class Calculator {
2.      public static void main(String[] args) {
3.          Integer radius = (Integer) 123;
4.          Integer height = 234;
5.          System.out.println(radius + ", " + height);
6.      }
7.  }
```

What happens if you try to compile and run the code?

A. It will not compile because of a compile error on line 3

B. It will not compile because you are trying to convert an int to an Integer on line 4.

C. It will compile and print 123, 234

D. It will compile but it will throw a runtime exception when executed

7 Question ?

Given

```
public class VideoGame {
    public static void main(String[] args) {
        switch (args[1]) {
        case "1":
            System.out.println("One player");
            break;
        case "2":
            System.out.println("Two players");
            break;
        default:
            System.out.println("Unknown");
        }
    }
}
```

What will happen if the VideoGame class is invoked using this command:

```
java VideoGame 1 2
```

A. "One player" will be printed on the console

B. "Two players" will be printed on the console

C. An ArrayIndexOutOfBoundsException will be thrown

D. "Unknown" will be printed on the console

8 Question ?

The code below is an incomplete method named **digitsOnly** that returns true if the string argument contains only digits.

```
1. public static boolean digitsOnly(String s) {
2.     String reference = "0123456789";
3.     for (int i = 0; i < s.length(); i++) {
4.        ...
5.             return false;
6.         }
7.     }
8.     return true;
9. }
```

Which line of code needs to be inserted into line 4 for the method to work correctly? (Choose all that apply)

A. if (s.charAt(i) < 48 || s.charAt(i) > 57) {

B. if (!reference.contains(s.charAt(i))) {

C. if (!reference.contains("" + s.charAt(i))) {

D. if (s.charAt(i) <= 48 || s.charAt(i) >= 57) {

9 Question

Given

```
public class SalaryCalculator {
    public static void main(String[] args) {
        Long i = new Long(100L);
        Boolean b = null;
        String s = "1000" + i + (b? 1 : 0);
        System.out.println(s);
    }
}
```

What happens if you try to compile and run the code?

A. It will compile and "10001000" will be printed on the console

B. It will compile and "10001001" will be printed on the console

C. It will compile and a NullPointerException will be thrown when the code is executed.

D. It will not compile because you are trying to concatenate a String with an long and a boolean.

10 Question

Given the code below

```
String s = "kingkong";
System.out.println(s.replace("d", "k"));
```

What will be printed on the console if the code is executed?

A. dingdong

B. dingkong

C. kingkong

D. kingdong

Answers

1 **A**.

String objects created using the same string literal are shared. Thus, s1 and s2 reference the same object and s1==s2 evaluates to true. String objects containing the same string but created using the String constructor are different objects. Therefore, s3 and s4 point to two different objects and s3==s4 evaluates to false.

2 **A**.

Using **StringTokenizer** is an old technique for splitting a string. It does not consider an empty string a token. By contrast, the **split** method returns an empty string as a token so you know if there are empty tokens in a string. Use **split** instead of **StringTokenizer**.

3 **A**.

The **trim** method removes all white spaces, including tabs.

4 **B**.

The **lastIndexOf** method return the last occurrence of the given substring, which in this case is a dot. The **substring** method returns a substring starting from the given index.

5 **B**.

Passing an object to **System.out.print** will call the object's **toString** method and print the returned String.

6 **C**.

Thanks to boxing and unboxing, conversion from a primitive to a wrapper class and vice versa happen automatically. In this case, an int will be converted to Integer with or without type casting.

7 **B**.

args[1] refers to the second argument, so "Two Players" will be printed on the console.

8 **A, C**.

A is correct because the ASCII code for 0 is 48 and the ASCII code for 9 is 57. B causes a compile error because the contains method expects a String and charAt returns a char. C is correct because the argument is implicitly converted to a String by appending the returned char with an empty string. D is incorrect since this means characters 0 and 9 will be rejected.

9 **C**.

Inquiring the value of a **Boolean** is equivalent to calling its **booleanValue** method. Since **b** is null, this will thrown a **NullPointerException**.

10 **C**.

The **replace** method replaces all occurrences of the first argument with the second argument.