

Tutorial: OOP - Wrapping Up

1 The "static" Variables/Methods

You can apply modifier "static" to variables and methods (and inner classes).

- A static variable/method belongs to the class, and is shared by all instances. Hence, it is also called a *class variable/method*.
- On the other hand, a non-static variable/method (absence of keyword static) belongs to a specific instance of a class, also called an *instance variable/method*.

Each instance maintains its own storage. As the result, each instance variable/method has its own copy in the instances and not shared among different instances. To reference an instance variable/method, you need to identify the instance, and reference it via *anInstanceName.aVariableName* or *anInstanceName.aMethodName()*.

A static variable/method has a **single common memory location** kept in the class and shared by all the instances. The JVM allocates static variable during the class loading. The static variable exists even if no instance is created. A static variable/method can be referenced via *AClassName.aStaticVariableName* or *AClassName.aStaticMethodName()*. It can also be referenced from any of its instances (but not recommended), e.g., *instance1.aStaticVariableName* or *instance2.aStaticVariableName*.

Non-static variables/methods belong to the instances. To use a non-static variable/method, an instance must first be constructed. On the other hand, static variables/methods belong to the class, they are "global" in nature. You need not construct any instance before using them.

The usage of static variables/methods are:

- Provide constants and utility methods, such as `Math.PI`, `Math.sqrt()` and `Integer.parseInt()` which are public static and can be used directly through the class without constructing instances of the class.
- Provide a "global" variable, which is applicable to all the instances of that particular class, for purposes such as counting the number of instances, resource locking among instances, and etc.

(I believe that the keyword "static" is used to signal it does not change among the instances. In C, a static variable in a function block does not change its value across multiple invocation. C++ extends static to denote class variables/methods.)

UML Notation: static variables/methods are underlined in the class diagram.

1.1 Examples

Counting the number of instance created - Instance variable won't work!

Suppose that we want to count the number of instances created. Using an instance variable doesn't work?!

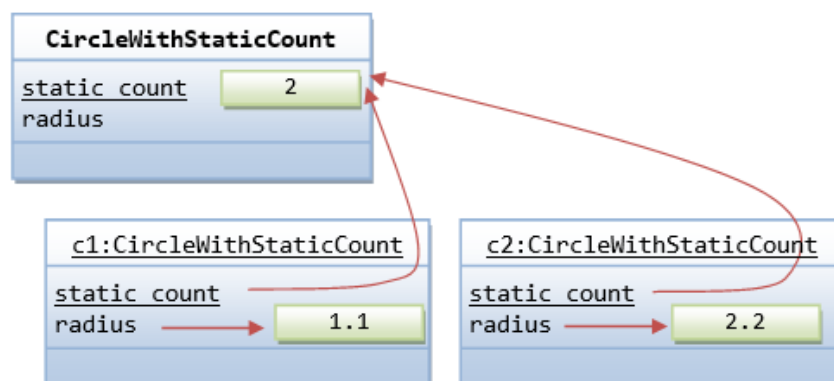


```
1 public class Circle {
    // Member variables
3 public int count = 0; // To count the number of instances created.
  private double radius; // Private member variable
5
6 public Circle(double radius) { // Constructor
7     this.radius = radius;
8     ++count;
9 }
10
11 // Test Driver
12 public static void main(String[] args) {
13     Circle c1 = new Circle(1.1);
14     System.out.println("count is: " + c1.count);
15     //count is: 1
16
17     Circle c2 = new Circle(2.2);
18     System.out.println("count is: " + c2.count);
19     //count is: 1
20 }
21 }
```

This is because count is an instance variable. Each instance maintains its own count. When a new instance is constructed, count is initialized to 0, then increment to 1 in the constructor.

Use a static variable to count the number of instances

We need to use a "static" variable, or class variable which is shared by all instances, to handle the count.



CircleWithStaticCount.java



```
1  /**
   * The CircleWithStaticCount class
   */
3  public class CircleWithStaticCount {
5
   // A static variable to count the number of instances created, shared by all the instances
7  public static int count = 0; // Set to public to simplify access
9
   // An instance variable for each circle to maintain its own radius
10 private double radius;
11
   // constructor
12 public CircleWithStaticCount(double radius) {
13     this.radius = radius;
14     ++count; // one more instance created
15 }
16
   // Test Driver
17 public static void main(String[] args) {
18     CircleWithStaticCount c1 = new CircleWithStaticCount(1.1);
19     System.out.println("count is: " + c1.count);
20     //count is: 1
21     // It is recommended to access static variable via the class instead of instances
22     System.out.println("count is: " + CircleWithStaticCount.count);
23     //count is: 1
24
25     CircleWithStaticCount c2 = new CircleWithStaticCount(2.2);
26     System.out.println("count is: " + CircleWithStaticCount.count);
27     //count is: 2
28     System.out.println("count is: " + c1.count);
29     //count is: 2
30     System.out.println("count is: " + c2.count);
31     //count is: 2
32
33     CircleWithStaticCount c3 = new CircleWithStaticCount(3.3);
34     System.out.println("count is: " + CircleWithStaticCount.count);
35     //count is: 3
36     System.out.println("count is: " + c1.count);
37     //count is: 3
38     System.out.println("count is: " + c2.count);
39     //count is: 3
40     System.out.println("count is: " + c3.count);
41     //count is: 3
42
43 }
44 }
45 }
```

Using static variables/methods as "global" variables and "utility" methods

Another usage of "static" modifier is to provide "global" variables and "utility" methods that are accessible by other classes, without the need to create an instance of that providing class. For example, the class `java.lang.Math` composes purely public static variables and methods. To use the static variable in `Math` class (such as `PI` and `E`) or static methods (such as `random()` or `sqrt()`), you do not have to create an instance of `Math` class. You can invoke them directly via the class name, e.g., `Math.PI`, `Math.E`, `Math.random()`, `Math.sqrt()`.

Non-static (instance) methods: Although from the OOP view point, each instance has its own copy of instance methods. In practice, the instances do not need their own copy, as methods do not have states and the implementation is exactly the same for all the instances. For efficiency, all instances use the copy stored in the class.

Within a class definition, a static method can access only static variables/methods. It cannot access non-static instance variables/methods, because you cannot identify the instance. On the other hand, an instance method can access static and non-static variables/methods. For example,

StaticTest.java



```
1  /**
   * The StaticTest class
   */
3  public class StaticTest {
5      private static String msgStatic = "Hello from static"; // a static variable
7      private String msgInstance = "Hello from non-static"; // a non-static instance
                                                                // variable
9
11     public static void main(String[] args) { // a static method
12         System.out.println(msgStatic);
13         // Hello from static
14         // System.out.println(msgInstance);
15         // compilation error: non-static variable xxx cannot be referenced from a static context
16     }
17 }
```

If a class has only one single instance (known as singleton design pattern), it could be more efficient to use static variable/method for that particular one-instance class?!

static variable or methods cannot be hidden or overridden in the subclass as non-static.

1.2 The Static_INITIALIZER

A *static initializer* is a block of codes labeled `static`. The codes are executed *exactly once*, when the class is loaded. For example,

StaticInitializerTest.java



```
/**
 2  * The StaticInitializerTest class
  */
4  public class StaticInitializerTest {
    static int number; // a static variable
6    static { // a static initializer block - run once when the class is loaded
        number = 88;
8        System.out.println("running static initializer ...");
    }
10
    // Test Driver
12    public static void main(String[] args) {
        System.out.println("running main() ...");
14        System.out.println("number is: " + number);
    }
16    // running static initializer ...
    // running main() ...
18    // number is: 88
}
```

During the class loading, JVM allocates the static variables and then runs the static initializer. The static initializer could be used to initialize static variables or perform an one-time tasks for the class.

1.3 The Class Loader

Every JVM has a built-in *class loader* (of type `java.lang.ClassLoader`) that is responsible for loading classes into the memory of a Java program. Whenever a class is referenced in the program, the class loader searches the classpath for the class file, loads the bytecode into memory, and instantiates a `java.lang.Class` object to maintain the loaded class.

The class loader loads a class only once, so there is only one `java.lang.Class` object for each class that used in the program. This `Class` object stores the static variables and methods.

During the class loading, the class loader also allocates the static variables, and invokes the explicit initializers and static initializers (in the order of appearance).

ClassLoaderTest



```

1  /**
   * The ClassLoaderTest class
   */
3  public class ClassLoaderTest {
5      private static int number1 = 11; // explicit initializer
      static { // static initializer
7          System.out.println("running static initializer ...");
          number1 = 99;
9          number2 = 88;
          System.out.println("number1 is " + number1);
11         // System.out.println("number2 is: " + number2);
          // compilation error: illegal forward reference
13     }

15     private static int number2 = 22; // explicit initializer

17     // Test Driver
     public static void main(String[] args) {
19         System.out.println("running main() ...");
         System.out.println("in main(): number1 is " + number1 + ", number2 is: "
21             + number2);
     }
23     // running static initializer ...
     // number1 is 99
25     // running main() ...
     // in main(): number1 is 99, number2 is: 22
27 }

```

1.4 The Instance_INITIALIZER

Similarly, you could use the so-called instance initializer, which runs during the instantiation process, to initialize an instance. Instance initializer is rarely-used. For example,

InstanceInitializerTest.java



```

1  class Foo {
     int number; // an instance variable
3      { // an instance initializer block – run once per instantiation
         System.out.println("running instance initializer ...");
5         number = 99;
     }

7     public Foo() { // Constructor
9         System.out.println("running constructor ...");
         System.out.println("number is: " + number);

```



```
11     }  
12 }  
13  
14 public class InstanceInitializerTest {  
15     public static void main(String[] args) {  
16         Foo f1 = new Foo();  
17         // running instance initializer ...  
18         // running constructor ...  
19         // number is: 99  
20  
21         Foo f2 = new Foo();  
22         // running instance initializer ...  
23         // running constructor ...  
24         // number is: 99  
25     }  
26 }
```

1.5 The "Instantiation" Process

The sequence of events when a new object is instantiated via the new operator (known as the instantiation process) is as follows:

- JVM allocates memory for the instance in the heap.
- JVM initializes the instance variables to their assigned values or default values.
- JVM invokes the constructor.
- The first statement of the constructor is always a call to its immediate superclass' constructor. JVM invokes the selected superclass' constructor.
- JVM executes the instance initializers in the order of appearance.
- JVM executes the body of the constructor.
- The new operator returns a reference to the new object.

For example,

InstantiationTest.java



```
1 class Foo {  
2     public int number1 = 11; // explicit initializer  
3     { // instance initializer  
4         number1 = 33;  
5         number2 = 44;  
6     }  
7 }
```



```
public int number2 = 22; // explicit initializer
8
public Foo() {
10     System.out.println("running Foo() ...");
12 }
14
public Foo(int number1, int number2) { // Constructor
14     System.out.println("running Foo(int, int) ...");
16     this.number1 = number1; // run after initializers
16     this.number2 = number2;
18 }
18
20 public class InstantiationTest {
20     public static void main(String[] args) {
22         Foo f1 = new Foo();
22         // running Foo() ...
24         System.out.println("number1 is " + f1.number1 + ", number2 is "
24             + f1.number2);
26         // number1 is 33, number2 is 22
28
28         Foo f2 = new Foo(55, 66);
28         //running Foo(int, int) ...
30         System.out.println("number1 is " + f2.number1 + ", number2 is "
30             + f2.number2);
32         // number1 is 55, number2 is 66
34     }
34 }
```

2 The "final" Class/Variable/Method

You can declare a class, a variable or a method final.

- A final class cannot be sub-classed (or extended).
- A final method cannot be overridden in the subclasses.
- A final variable cannot be re-assigned a new value.

final Variables of Primitive Type vs. Reference Type

- A final variable of primitive type is a *constant*, whose value cannot be changed. A "public static final" variable of primitive type is a *global constant*, whose value cannot be changed. For example,



```

2 // class java.lang.Math
  public static final double PI = 3.141592653589793;
  public static final double E = 2.718281828459045;

4

  // class java.lang.Integer
6  public static final int MAX_VALUE = 2147483647;
  public static final int MIN_VALUE = -2147483648;
8  public static final int SIZE = 32;

```

- A final variable of a reference type (e.g., an instance of a class or an array) cannot be re-assigned a new reference. That is, you can modify the content of the instance, but cannot re-assign the variable to another instance. For example,

FinalReferenceTest.java



```

1 public class FinalReferenceTest {
2     public static void main(String[] args) {
3         final StringBuffer sb = new StringBuffer("hello"); // final
4         // ↳ reference of mutable object
5         sb.append(", world"); // can change the contents of the
6         // ↳ referenced object
7         System.out.println(sb); // hello , world
8
9         // sb = new StringBuffer("world peace");
10        // compilation error: cannot assign a value to final variable xxx
11
12        final int[] numbers = {11, 22, 33}; // a final array reference
13        numbers[0] = 44;
14        System.out.println(java.util.Arrays.toString(numbers));
15        // [44, 22, 33]
16    }
17 }

```

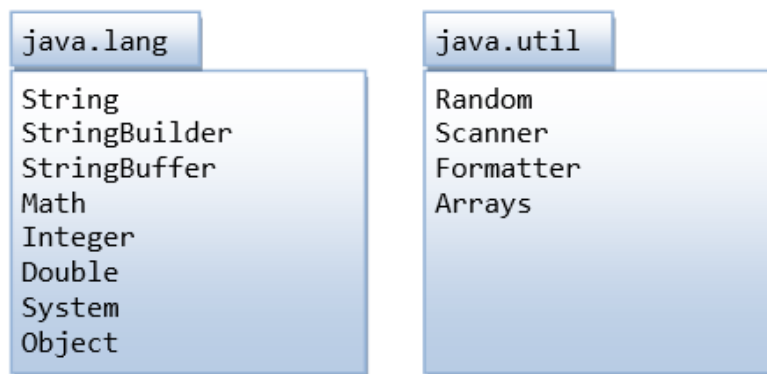
final vs. abstract

final is opposite to abstract. A final class cannot be extended; while an abstract class must be extended and the extended class can then be instantiated. A final method cannot be overridden; while an abstract method must be overridden to complete its implementation. [abstract modifier is applicable to class and method only.]

3 Package, Import, Classpath & JAR

If I have a class called Circle and you also have a class called Circle. Can the two Circle classes co-exist or even be used in the same program? The answer is yes, provided that the two Circle classes are placed in two *different packages*.

A *package*, like a library, is a *collection of classes*, and other related entities such as interfaces, errors, exceptions, annotations, and enums.



UML Notation: Packages are represented in UML notation as tabbed folders, as illustrated.

Package name (e.g., java.util) and classname (e.g., Scanner) together form the so-called *fully-qualified name* in the form of *packagename.classname* (e.g., java.util.Scanner), which unambiguously identifies a class.

Packages are used for:

1. *Organizing* classes and related entities.
2. *Managing namespaces* - Each package is a *namespace*.
3. *Resolving naming conflicts*. For example, com.zzz.Circle and com.yyy.Circle are treated as two distinct classes. Although they share the same classname Circle, they belong to two different packages: com.zzz and com.yyy. These two classes can co-exist and can even be used in the same program via the fully-qualified names.
4. *Access control*: Besides public and private, you can grant access of a class/variable/method to classes within the same package only.
5. *Distributing Java classes*: All entities in a package can be combined and compressed into a single file, known as JAR (Java Archive) file, for distribution.

Package Naming Convention

A package name is made up of the reverse of the domain Name (to ensure uniqueness) plus your own organization's project name *separated by dots*. Package names are in lowercase.

For example, suppose that your Internet Domain Name is "zzz.com", you can name your package as "com.zzz.project1.subproject2".

The prefix "java" and "javax" are reserved for the core Java packages and Java extensions, e.g., java.lang, java.util, and java.net, javax.net.

Package Directory Structure

The "dots" in a package name correspond to the directory structure for storing the class files. For example, the com.zzz.Cat is stored in directory "...\\com\\zzz\\Cat.class" and com.yyy.project1.subproject2.Orange is stored in directory "...\\com\\yyy\\project1\\subproject2\\Orange.class", where "..." denotes the *base directory of the package*.

JVM can locate your class files only if the *package base directory* and the *fully-qualified name* are given. The *package base directory* is provided in the so-called *classpath*.

The "dot" does not mean sub-package (there is no such thing as sub-package). For example, java.awt and java.awt.event are two *distinct packages*. Package java.awt is kept in "...\\java\\awt"; whereas package java.awt.event is stored in "...\\java\\awt\\event".

3.1 The "import" Statement

There are two ways to reference a class in your source codes:

1. Use the *fully-qualified name* in the form of *packagename.classname* (such as java.util.Scanner). For example,



```
public class ScannerNoImport {  
2   public static void main(String[] args) {  
    // Use fully-qualified name in "ALL" the references  
4   java.util.Scanner in = new java.util.Scanner(System.in);  
    System.out.print("Enter a integer: ");  
6   int number = in.nextInt();  
    System.out.println("You have entered: " + number);  
8   }  
}
```

Take note that you need to use the *fully-qualified name* for ALL the references to the class. This is clumpy!

2. Add an "import *fully-qualified-name*" statement at the beginning of the source file. You can then use the classname alone (without the package name) in your source codes. For example,



```

import java.util.Scanner;

2
public class ScannerWithImport {
4     public static void main(String[] args) {
        // Package name can be omitted for an imported class
6        // Java compiler searches the import statements for the fully-qualified name
        Scanner in = new Scanner(System.in); // classname only
8        System.out.print("Enter a integer: ");
        int number = in.nextInt();
10       System.out.println("You have entered: " + number);
    }
12 }

```

The compiler, when encounter a unresolved classname, will search the import statements for the fully-qualified name.

The import statement provides us a *convenient way* for referencing classes without using the fully-qualified name. "Import" does not load the class, which is carried out by the so-called *class loader* at runtime. It merely resolves a classname to its fully-qualified name, or *brings the classname into the namespace*. "Import" is strictly a compile-time activity. The Java compiler replaces the classnames with their fully-qualified names, and removes all the import statements in the compiled bytecode. There is a slight compile-time cost but no runtime cost.

The import statement(s) must be placed after the package statement but before the class declaration. It takes the following syntax:



```

1  import packagename.classname;
   import packagename.*

```

You can import a *single class* in an import statement by providing its fully-qualified name, e.g.,



```

import java.util.Scanner; // import the class Scanner in package java.util
2  import java.awt.Graphics; // import the class Graphics in package java.awt

```

You can also import *all the classes in a package* using the *wildcard* *. The compiler will search the entire package to resolve classes referenced in the program. E.g.,



```
import java.util.*;           // import all classes in package java.util
2 import java.awt.*;          // import all classes in package java.awt
import java.awt.event.*;      // import all classes in package java.awt.event
```

Using wildcard may result in slightly fewer source lines. It has no impact on the resultant bytecode. It is not recommended as it lacks clarity and it may lead to ambiguity if two packages have classes of the same names.

The Java core language package `java.lang` is *implicitly imported to every Java program*. Hence no explicit import statements are needed for classes inside the `java.lang` package, such as `System`, `String`, `Math`, `Integer` and `Object`.

There is also no need for import statements for classes within the *same package*.

Take note that the import statement does not apply to classes in the *default* package.

3.2 The "import static" Statement (JDK 1.5)

Prior to JDK 1.5, only classes can be "imported" - you can omit the *package name* for an imported class. In JDK 1.5, the static variables and methods of a class can also be "imported" via the "import static" declaration - you can omit the *classname* for an imported static variable/method. For example:



```
1 import static java.lang.System.out; // import static variable "out" of "System" class
import static java.lang.Math.*;      // import "ALL" static variables/methods in "Math" class
3
public class TestImportStatic {
5     public static void main(String[] args) {
        // Classname can be omitted for imported static variables/methods
7         out.println("Hello, PI is " + PI);
        out.println("Square root of PI is " + sqrt(PI));
9     }
}
```

The import static statement takes the following syntax:



```
import static packagename.classname.staticVariableName;
2 import static packagename.classname.staticMethodName;
import static packagename.classname.*; // wildcard *denotes all
4                                     // static variables/methods of the class
```

Take note that `import` and `import static` statements does not apply to classes/members in the *default* package.

3.3 Creating Packages

To put a class as part of a package, include a package statement before the class definition (as the FIRST statement in your program). For example,



```
package com.zzz.test;

2
public class HelloPackage {
4     public static void main(String[] args) {
        System.out.println("Hello from a package...");
6     }
}
```

You can create and use package in IDE (such as Eclipse/NetBeans) easily, as the IDE takes care of the details. You can simply create a new package, and then create a new class inside the package.

Compiling Classes in Package

To compile classes in package using JDK, you need to use `-d` flag to specify the *destination package base directory*, for example,

Command window

```
1 // Set the current working directory to the directory containing HelloPackage.java
> javac -d e:\myproject HelloPackage.java
```

The `-d` option instructs the compiler to place the class file in the given package base directory, as well as to create the necessary directory structure for the package. Recall that the dot `'.'` in the package name corresponds to sub-directory structure. The compiled bytecode for `com.zzz.test.HelloPackage` will be placed at `"e:\myproject\com\zzz\test\HelloPackage.class"`

Running Classes in Package

To run the program, you need to set your current working directory at the package base directory (in this case `"e:\myproject"`), and provide the *fully-qualify name*:

Command window

```
2 // Set the current working directory to the package base directory
e:\myproject> java com.zzz.test.HelloPackage
```

It is important to take note that you shall always work in the *package base directory* and issue *fully-qualified name*.

As mentioned, if you use an IDE, you can compile/run the classes as usual. IDE will take care of the details.

3.4 The Default Unnamed Package

So far, all our examples do not use a package statement. These classes belong to a so-called default unnamed package. Use of the default unnamed package is not recommended should be restricted to toy programs only, as they cannot be "imported" into another application. For production, you should place your classes in proper packages.

3.5 Java Archive (JAR)

An Java application typically involves many classes. For ease of distribution, you could bundles all the class files and relevant resources into a single file, called JAR (Java Archive) file.

JAR uses the famous "zip" algorithm for compression. It is modeled after Unix's "tar" (Tape ARchive) utility. You can also include your digital signature (or certificate) in your JAR file for authentication by the recipients.

JDK provides an utility called "jar" to create and manage JAR files. For example, to create a JAR file, issue the following command:

Command window

```
// To create a JAR file from c1 ... cn classes (c:create, v:verbose, f:filename):
2 > jar cvf myjarfile.jar c1.class ... cn.class
```

Example

To place the earlier class `com.zzz.test.HelloPackage` (and possible more related classes and resources) in a JAR file called `hellopackage.jar`:

Command window

```
// Set the current working directory to the package base directory (i.e., e:\myproject)
2 e:\myproject> jar cvf hellopackage.jar com\zzz\test\HelloPackage.class
added manifest
4 adding: com/zzz/test/HelloPackage.class (in = 454) (out = 310)(deflated
  31%)
```

Read "Java Archive (JAR)" for more details.

3.6 Classpath - Locating Java Class Files

Java allows you to store your class files anywhere in your file system. To locate a class, you need to provide the *package base directory* called *classpath* (short for *user class search path*) and the *fully-qualified name*. For example, given that the package base directory is e:\myproject, the class com.zzz.test.HelloPackage can be found in e:\myproject\com\zzz\test\HelloPackage.class.

When the Java compiler or runtime needs a class (given its fully-qualified name), it searches for it from the classpath. You could specify the classpath via the command-line option -cp (or -classpath); or the environment variable CLASSPATH.

A classpath may contain many entries (separated by ';' in Windows or ':' in Unixes/Mac). Each entry shall be a package base directory (which contains many Java classes), or a JAR file (which is a single-file archives of many Java classes).

3.7 Example on Package, Classpath and JAR

In this example, we shall kept the source files and class files in separate directories - "src" and "bin" - for ease of distribution minus the source.

com.zzz.geometry.Circle

Let's create a class called Circle in package com.zzz.geometry. We shall keep the source file as d:\zzzpackages\src\com\zzz\geometry\Circle.java and the class file in package base directory of d:\zzzpackages\bin.



```
package com.zzz.geometry;  
  
2 public class Circle { // save as d:\zzzpackages\src\com\zzz\geometry\Circle.java  
4     public String toString() {  
        return "This is a Circle";  
6     }  
}
```

To compile the Circle class, use javac with -d option to specify the destination package base directory.

Command window

```
1 // Set current working directory to source file (d:\zzzpackages\src\com\zzz\geometry)  
> javac -d d:\zzzpackages\bin Circle.java  
3 // Output class file is d:\zzzpackages\bin\com\zzz\geometry\Circle.class*)
```


com.zzz.geometry.Cylinder

Next, create a class called Cylinder in the same package (com.zzz.geometry) that extends Circle.



```
1 package com.zzz.geometry;  
  
3 // save as d:\zzzpackages\src\com\zzz\geometry\Cylinder.java  
public class Cylinder extends Circle {  
5     public String toString() {  
        return "This is a Cylinder";  
7     }  
}
```

No import statement for Circle is needed in Cylinder, because they are in the same package.

To compile the Cylinder class, we need to provide a classpath to the Circle class via option -cp (or -classpath), because Cylinder class references Circle class.

Command window

```
2 // Set current working directory to source file (d:\zzzpackages\src\com\zzz\geometry)  
> javac -d d:\zzzpackages\bin -cp d:\zzzpackages\bin Cylinder.java  
// Output class file is d:\zzzpackages\bin\com\zzz\geometry\Cylinder.class
```

com.yyy.animal.Cat

Create another class called Cat in another package (com.yyy.animal). We shall keep the source file as d:\yyypackages\src\com\yyy\animal\Cat.java and the class file in package base directory of d:\yyypackages\bin.



```
1 package com.yyy.animal;  
  
3 public class Cat { // save as d:\yyypackages\src\com\yyy\animal\Cat.java  
    public String toString() {  
5        return "This is a Cat!";  
    }  
7 }
```

Again, use -d option to compile the Cat class. No classpath needed as the Cat class does not reference other classes.


```

Command window
1 // Set current working directory to source file (d:\yyypackages\src\com\yyy\animal)
> javac -d d:\yyypackages\bin Cat.java
3 // Output class file is d:\yyypackages\bin\com\yyy\animal\Cat.class

```

myTest.test

We shall write a Test class (in package myTest) to use all the classes. We shall keep the source file as d:\testpackages\src\mytest\Test.java and the class file in package base directory of d:\testpackages\bin.



```

1 package mytest;

3 import com.zzz.geometry.Circle;
import com.zzz.geometry.Cylinder;
5 import com.yyy.animal.Cat;

7 public class Test { // save as d:\testpackages\src\mytest\Test.java
    public static void main(String[] args) {
9         Circle circle = new Circle();
        System.out.println(circle);

11
        Cylinder cylinder = new Cylinder();
13        System.out.println(cylinder);

15        Cat cat = new Cat();
        System.out.println(cat);
17    }
}

```

To compile the Test class, we need -d option to specify the destination and -cp to specify the package base directories of Circle and Cylinder (d:\zzzpackages\bin) and Cat (d:\yyypackages\bin).

```

Command window
2 // Set current working directory to source file (d:\testpackages\src\mytest)
> javac -d d:\testpackages\bin -cp d:\zzzpackages\bin;d:\yyypackages\bin Test.java
// Output class file is d:\testpackages\bin\mytest\Test.class

```

To run the `myTest.Test` class, set the current working directory to the package base directory of `myTest.Test` (`d:\testpackages\bin`) and provide classpath for `Circle` and `Cylinder` (`d:\zzzpackages\bin`), `Cat` (`d:\yyypackages\bin`) and the current directory (for `myTest.Test`).

Command window

```
1 // Set current working directory to package base directory (d:\testpackages\bin)
> java -cp .;d:\zzzpackages\bin;d:\yyypackages\bin myTest.Test
```

Jarring-up `com.zzz.geometry` package

Now, suppose that we decided to jar-up the `com.zzz.geometry` package into a single file called `geometry.jar` (and kept in `d:\jars`):

Command window

```
2 // Set current working directory to package base directory (d:\zzzpackages\bin)
// 'c' to create, 'v' for verbose, 'f' follows by jar filename
> jar cvf d:\jars\geometry.jar com\zzz\geometry\*.class
4 added manifest
adding: com/zzz/geometry/Circle.class (in=300) (out=227)(deflated 24%)
6 adding: com/zzz/geometry/Cylinder.class (in=313) (out=228)(deflated 27%)
// Output is d:\jars\geometry.jar
8
// OR
10 // Set current working directory to package base directory (d:\zzzpackages\bin)
// jar the current directory (.) and its sub-directories
12 > jar cvf d:\jars\geometry.jar .
added manifest
14 adding: com/(in = 0) (out = 0)(stored 0%)
adding: com/zzz/(in = 0) (out = 0)(stored 0%)
16 adding: com/zzz/geometry/(in = 0) (out = 0)(stored 0%)
adding: com/zzz/geometry/Circle.class (in=300) (out=227)(deflated 24%)
18 adding: com/zzz/geometry/Cylinder.class (in=313) (out=228)(deflated 27%)
```

To run `myTest.Test` with the JAR file, set the classpath to the JAR file (classpath accepts both directories and JAR files).

Command window

```
2 // Set current working directory to package base directory (d:\testpackages\bin)
> java -cp .;d:\jars\geometry.jar;d:\yyypackages\bin myTest.Test
```

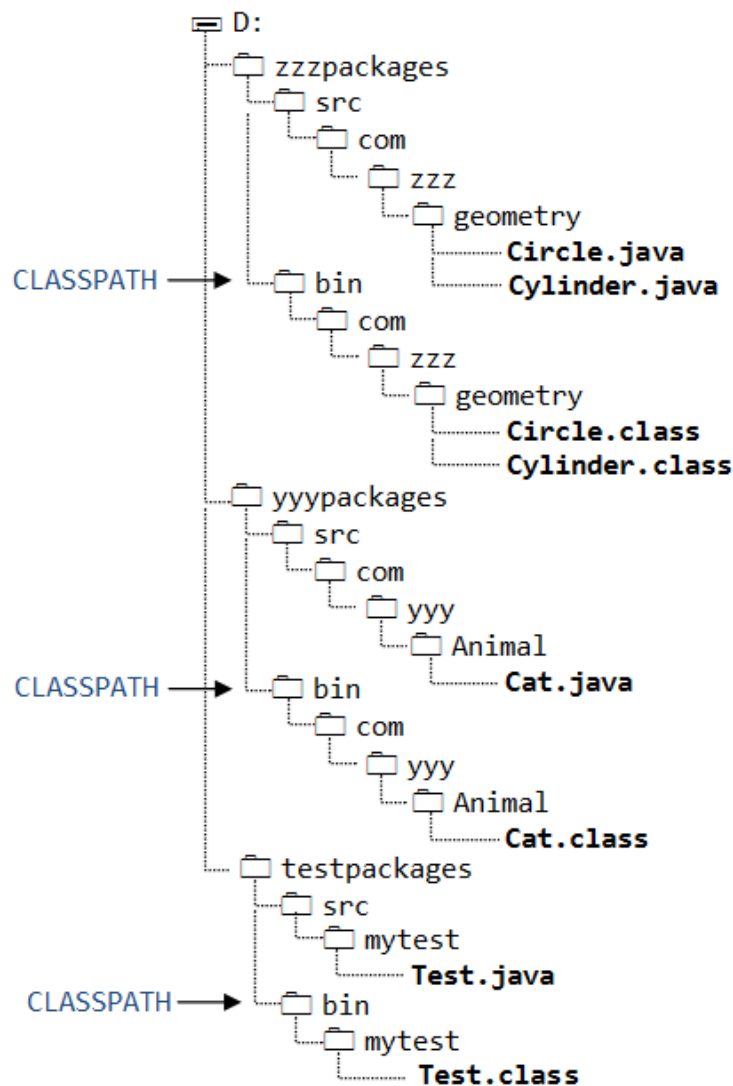
Separating Source Files and Classes

For ease of distribution (without source files), the source files and class files are typically kept in separate directories.

1. Eclipse keeps the source files under "src", class files under "bin", and jar files and native libraries under "lib".
2. NetBeans keeps the source files under "src", class files under "build\classes", jar files and native libraries under "build\lib".

Two Classes of the Same Classname?

Suppose that we have two Circle classes in two different packages, can we use both of them in one program? Yes, however, you need to use fully-qualified name for both of them. Alternatively, you may also import one of the classes, and use fully-qualified name for the other. But you cannot import both, which triggers a compilation error.



3.8 How JVM Find Classes

Reference: JDK documentation on "How classes are found".

To locate a class (given its fully-qualified name), you need to locate the base directory or the JAR file.

The JVM searches for classes in this order:

1. Java Bootstrap classes: such as "rt.jar" (runtime class), "i18n.jar" (internationalization class), charsets.jar, jre/classes, and others.
2. Java Standard Extension classes: JAR files located in "\$JDK_HOME\jre\lib\ext" directory (for Windows and Ubuntu); "/Library/Java/Extensions" and "/System/Library/Java/Extensions" (for Mac). The location of Java's Extension Directories is kept in Java's System Property "java.ext.dirs". User classes.

The user classes are searched in this order:

1. The default ".", i.e., the current working directory.
2. The CLASSPATH environment variable, which overrides the default.
3. The command-line option -cp (or -classpath), which overrides the CLASSPATH environment variable and default.
4. The runtime command-line option -jar, which override all the above.

The JVM puts the classpath is the system property java.class.path. Try running the following line with a -cp option and without -cp (which uses CLASSPATH environment variable) to display the program classpath:



```
System.out.println(System.getProperty("java.class.path"));
```

The CLASSPATH Environment Variable

Alternatively, you could also provide your classpath entries in the CLASSPATH *environment variable*. Take note that if CLASSPATH is not set, the default classpath is the current working directory. However, if you set the CLASSPATH environment variable, you must include the current directory in the CLASSPATH, or else it will not be searched.

It is recommended that you use the -cp (-classpath) command-line option (customized for each of your applications), instead of setting a permanent CLASSPATH environment for all the Java applications. IDE (such as Eclipse/NetBeans) manages -cp (-classpath) for each of the applications and does not rely on the CLASSPATH environment.

3.9 More Access Control Modifiers – protected and default package-private

Java has four *access control modifiers* for class/variable/method. Besides the public (available to all outside classes) and private (available to this class only), they are two modifiers with visibility in between public and private:

- **protected**: available to all classes in the same package and the subclasses derived from it.
- **default (package-private)**: If the access control modifier is omitted, by default, it is available to classes in the same package only. This is also called *package-private* accessibility.

4 Java Source File

A Java source file must have the file type of ".java". It can contain at most one top-level public class, but may contain many non-public classes (not recommended). The file name shall be the same as the top-level public classname.

The source file shall contain statements in this order:

1. Begins with one optional package statement. If the package statement is omitted, the default package (.) is used. Use of default package is not recommended for production.
2. Follows by optional import or import static statement(s).
3. Follows by class, interface or enum definitions.

Each class, interface or enum is compiled into its own ".class" file.

The top-level class must be either public or *default*. It cannot be private (no access to other classes including JVM?!) nor protected (meant for member variables/methods accessible by subclasses), which triggers compilation error "modifier private|protected not allowed here".

5 Dissecting the Hello-world

Let us re-visit the "Hello-world" program, which is reproduced below:



```
1 public class Hello {  
    public static void main(String[] args) {  
3        System.out.println("hello , world");  
    }  
5 }
```

- The class `Hello` is declared `public` so that it is accessible by any other classes. In this case, the JRE needs to access the `Hello` class to run the `main()`.

Try declaring the `Hello` class `private`/`protected`/`package` and run the program.



```
1  private class Hello { ... }  
   // compilation error: modifier private not allowed here  
3  
   protected class Hello { ... }  
5   // compilation error: modifier protected not allowed here  
7  
   class Hello { ... } // default of package access  
   // okay
```

`private` and `protected` are not allowed for outer class. `package` is fine and JRE can also run the program?! What is the use of a `private` class, which is not accessible to others? The usage of `private` class will be explained later in the so-called *inner class*.

- Similarly, the `main()` method is declared `public`, so that JRE can access and invoke the method.

Try declaring the `main()` method `private`/`protected`/`package`.



```
1  private | protected | package static void main(String[] args) { ... }  
   // no compilation error  
3  // runtime error: Main method not found in class Hello
```

You can compile the `main()` with `private`/`protected`/`package`, but cannot run the `main()` method.

- The `main()` method is declared `static`. Remember that a `static` variable/method belongs to the *class* instead of a particular *instance*. There is no need to create an instance to use a `static` variable/method. A `static` method can be invoked via the classname, in the form of `aClassName.aStaticMethodName()`. JRE can invoke the `static main()` method, by calling `Hello.main()` from the class directly. Note that we did not create any instance of the `Hello` class.

Try omitting the `static` keyword and observe/explain the error message.



```

2 public void main(String[] args) { ... } // missing static
                                     // - an instance method
// no compilation error
4 // runtime error: Main method not found in class Hello

```

- The main() method takes an argument of a String array, corresponding to the command-line arguments supplied by the user, performs the program operations, and return void (or nothing) to the JRE.

Try omitting the argument (String[] args) from the main() method.



```

1 public static void main() { // missing String[] argument
                             // - another overloaded version of main()
3     ...
}
5 // no compilation error
  // runtime error: Main method not found in class Hello

```

You can compile, but JRE cannot find the matching main(String[]).

- In C language, the signature of main() function is:



```

1 main(int argc, char *argv[]) { ..... }

```

Two parameters are needed for the command-line argument - int argc to specify the number of arguments and the string-array argv to keep the arguments. In Java, only one parameter - a String array is needed. This is because Java array contains the length internally, and the number of arguments can be retrieved via args.length. Furthermore, in C, the name of the program is passed as the first command-line argument. In Java, the program name is not passed, as the class name is kept within the object. You can retrieve the class name via this.getClass().getName().

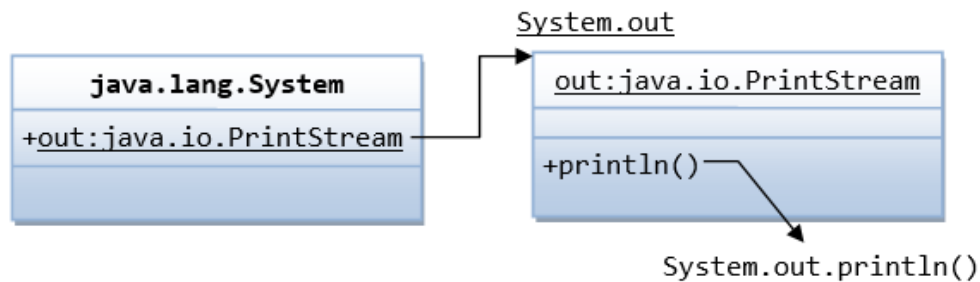
System.out.println()

If you check the JDK API specification, you will find that:

- "System" is a class in the package java.lang.
- "out" is a static public variable of the class java.lang.System.

- "out" is an instance of class "java.io.PrintStream".
- The class java.io.PrintStream provides a public method called "println()".

The figure illustrate the classes involved in System.out.println().



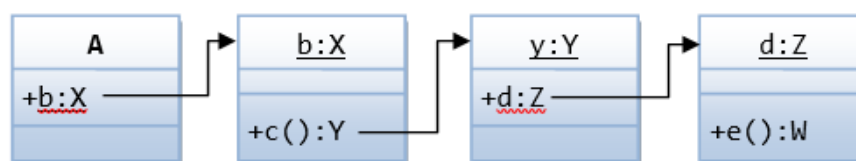
The figure illustrate the classes involved in System.out.println().

Take note that each of the dot (.) opens a 3-compartment box!!!

Example

As an example, the reference "A.b.c().d.e()" can be interpreted as follows:

- "A" is a class.
- "b" is a public static variable of class "A" (because it is referenced via the classname).
- The variable "b" belongs to a class say "X".
- The class "X" provides a public method "c()".
- The "c()" method returns an instance "y" of class say "Y".
- The "Y" class has a variable (static or instance) called "d".
- The variable "d" belongs to a class say "Z".
- The class "Z" provides a public method called "e()".



6 Nested and Inner Classes

It is possible to define a class within another class. Such class is called a nested class in Java terminology. A class that is not a nested class is called a top-level class.

Java has four types of nested classes:

- Static nested classes
- Inner classes
- Local classes
- Anonymous classes

Using nested classes may increase the readability of the code and improve the organization of the code. Inner classes are often used as callbacks in GUI. For example in Java Swing toolkit.

Java static nested classes

A static nested class is a nested class that can be created without the instance of the enclosing class. It has access to the static variables and methods of the enclosing class. For examples,

StaticNestedClassTest.java



```
package com.nestedclass;

2
public class StaticNestedClassTest {

4
    // The static variable, it can be accessed by a static nested class.
6    private static int x = 5;

8    // A static nested class is defined in StaticNestedClassTest.
    // It has one method which prints a message and refers to the static x variable.
10   static class Nested {

12       @Override
        public String toString() {
14           return "This is a static nested class; x:" + x;
16       }
    }

18   public static void main(String[] args) {

20       StaticNestedClassTest.Nested staticNestedClass =
        new StaticNestedClassTest.Nested();
22
        System.out.println(staticNestedClass);
24   }
}
```

The dot operator is used to refer to the nested class.



```
1 StaticNestedClassTest.Nested staticNestedClass =  
   new StaticNestedClassTest.Nested();
```

This is the output of the com.zetcode.StaticNestedClassTest program.

Command window

```
java com.nestedclass.StaticNestedClassTest  
2 This is a static nested class; x:5
```

Java inner classes

An instance of a normal or top-level class can exist on its own. By contrast, an instance of an inner class cannot be instantiated without being bound to a top-level class. Inner classes are also called member classes. They belong to the instance of the enclosing class. Inner classes have access to the members of the enclosing class.



```
package com.nestedclass;  
2  
public class InnerClassTest {  
4  
    private int x = 5;  
6  
    // A nested class is defined in the InnerClassTest class.  
8    // It has access to the member x variable.  
    class Inner {  
10  
        @Override  
12        public String toString() {  
            return "This is Inner class; x:" + x;  
14        }  
    }  
16  
    public static void main(String[] args) {  
18  
        InnerClassTest nc = new InnerClassTest();  
20        InnerClassTest.Inner inner = nc.new Inner();  
22  
        System.out.println(inner);  
24    }  
}
```

First, we need to create an instance of the top-level class. Inner classes cannot exist without an instance of the enclosing class.

Once we have the top-level class instantiated, we can create the instance of the inner class.



```
InnerClassTest.Inner inner = nc.new Inner();
```

This is the output of the com.zetcode.InnerClassTest program.

Command window

```
1 java com.nestedclass.InnerClassTest
   This is Inner class; x:5
```

Java variable shadowing

If a variable in the inner scope has the same name as the variable in the outer scope then it is shadowing it. It is still possible to refer to the variable in the outer scope.



```
package com.nestedclass;

2  /**
   * The Shadowing class,
   * it has x variable in the top-level class, in the inner class, and inside a method.
   */
6  public class Shadowing {
    private int x = 0;

8      class Inner {

10         private int x = 5;

12         void method1(int x) {
14             // We refer to the x variable defined in the local scope of the method.
            System.out.println(x);

16             // Using this keyword, we refer to the x variable defined in the Inner class.
            System.out.println(this.x);

18             // We can refer to the x variable of the Shadowing top-level class.
            System.out.println(Shadowing.this.x);

20         }
22     }

24     public static void main(String[] args) {

26
```



```

28     Shadowing sh = new Shadowing();
        Shadowing.Inner si = sh.new Inner();

30     si.method1(10);
        }
32 }

```

This is example output.

Command window

```

2  java com.nestedclass.Shadowing
   10
   5
4  0

```

Java local classes

A local class is a special case of an inner class. Local classes are classes that are defined in a block. (A block is a group of zero or more statements between braces.) A local class has access to the members of its enclosing class. In addition, a local class has access to local variables if they are declared final. The reason for this is technical. The lifetime of an instance of a local class can be much longer than the execution of the method in which the class is defined. To solve this, the local variables are copied into the local class. To ensure that they are later not changed, they have to be declared final.

Local classes cannot be public, private, protected, or static. They are not allowed for local variable declarations or local class declarations. Except for constants that are declared static and final, local classes cannot contain static fields, methods, or classes.

LocalClassTest.java



```

package com.nestedclass;

2  public class LocalClassTest {

4      public static void main(String[] args) {

6          final int x = 5;

8          // A local class is defined in the body of the main() method.
10         class Local {

12             // A local class can access local variables if they are declared final.

```



```

14      @Override
      public String toString() {
16          return "This is Local class; x:" + x;
      }
18  }

18      Local loc = new Local();
20      System.out.println(loc);
22  }

```

Java anonymous classes

Anonymous classes are local classes that do not have a name. They enable us to declare and instantiate a class at the same time. We can use anonymous classes if we want to use the class only once. An anonymous class is defined and instantiated in a single expression. Anonymous inner classes are also used where the event handling code is only used by one component and therefore does not need a named reference.

An anonymous class must implement an interface or inherit from a class. But the implements and extends keywords are not used. If the name following the new keyword is the name of a class, the anonymous class is a subclass of the named class. If the name following new specifies an interface, the anonymous class implements that interface and extends the Object.

Since an anonymous class has no name, it is not possible to define a constructor for an anonymous class. Inside the body of an anonymous class we cannot define any statements; only methods or members.

We create an anonymous class.

AnonymousClass.java



```

package com.nestedclass;

2  public class AnonymousClass {

4      interface Message {
6          public void send();
      }

8      public void createMessage() {
10         Message msg = new Message() {
12             @Override
14             public void send() {

```



```
16         System.out.println("This is a message");
17     }
18 }
19
20 msg.send();
21
22 public static void main(String[] args) {
23
24     AnonymousClass ac = new AnonymousClass();
25     ac.createMessage();
26 }
}
```

An anonymous class must be either a subclass or must implement an interface. Our anonymous class will implement a Message interface. Otherwise, the type would not be recognized by the compiler.



```
1 interface Message {
2     public void send();
3 }
```

An anonymous class is a local class, hence it is defined in the body of a method. An anonymous class is defined in an expression; therefore, the enclosing right bracket is followed by a semicolon.



```
1 public void createMessage() {
2
3     Message msg = new Message() {
4
5         @Override
6         public void send() {
7             System.out.println("This is a message");
8         }
9     };
10
11     msg.send();
12 }
```

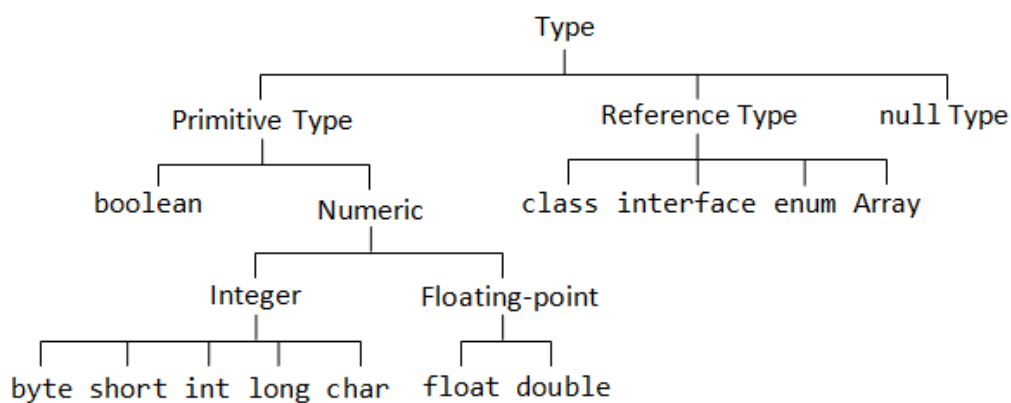
7 More on Variables and References

7.1 Types of Variables

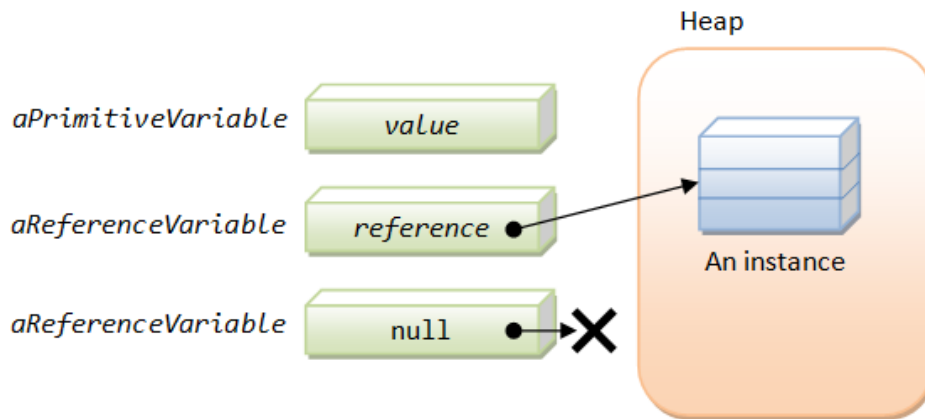
The *type* of a variable determines what kinds of *value* the variable can hold and what operations can be performed on the variable. Java is a "strong-type" language, which means that the type of the variables must be known at compile-time.

Java has three kinds of types:

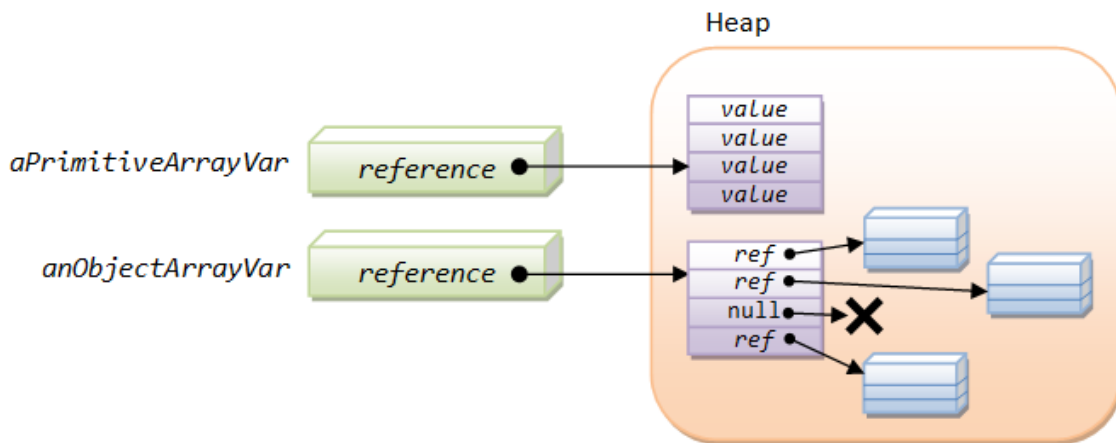
1. Primitive type: There are eight primitive types in Java: byte, short, int, long, float, double, char, and boolean. A primitive-type variable holds a *simple value*.
2. Reference type: Reference types include class, interface, enum and array. A reference-type variable holds a *reference* to an object.
3. A special null type, holding a special null reference. It could be assigned to a reference variable that does not reference any object.



A primitive variable holds a primitive value (in this storage). A reference variable holds a reference to an object or array in the heap, or null. A reference variable can hold a reference of the type or its sub-type (polymorphism). The value null is assigned to a reference variable after it is declared. A reference is assigned after the instance is constructed. An object (instance) resides in the heap. It must be accessed via a reference.



Java implicitly defines a reference type for each possible array type - one for each of the eight primitive types and an object array.



7.2 Scope & Lifetime of Variables

The *scope* of a variable refers to the portion of the codes that the variable can be accessed. The *lifetime* refers to the span the variable is created in the memory until it is destroyed (garbage collected). A variable may exist in memory but not accessible by certain codes.

Java supports three types of variables of different lifetimes:

Automatic Variable (or Local Variable): *Automatic Variables* include method's local variables and method's parameters. Automatic variables are created on entry to the method and are destroyed when the method exits. The scope of automatic variables of a method is inside the *block* where they are defined. Local variable cannot have access modifier (such as `private` or `public`). The only modifier applicable is `final`.

For example,



```

public class AutomaticVariableTest {
2   public static void main(String[] args) {    // scope of method parameter args is
                                                // within this method
4       for (int i = 0; i < 10; ++i) {    // scope of i is within the for-loop
           System.out.println(i);
6       }
           // System.out.println(i);    // i has gone out of scope
           // compilation error: cannot find symbol

10      int j = 0;    // scope of j is within the method, from this statement onwards
                    // (forward reference)
12      for (j = 0; j < 10; ++j) {
           System.out.println(j);
14      }
           System.out.println(j);    // okay

16      int k = 1;
18      do {
           int x = k*k;    // scope of x is within the block
20           ++k;
           System.out.println(k);
22      } while (x < 100);    // compilation error: cannot find symbol
24  }
}

```

Member variable (or Instance variable) of a Class: A member variable of a class is created when an instance is created, and it is destroyed when the object is destroyed (garbage collected).

Static variable (or Class variable) of a Class: A *static variable of a class* is created when the class is loaded (by the JVM's class loader) and is destroyed when the class is unloaded. There is only one copy for a static variable, and it exists regardless of the number of instances created, even if the class is not instantiated. Take note that static variables are created (during class loading) before instance variables (during instantiation).

7.3 Variable Initialization

All class member and static variables that are not explicitly assigned a value upon declaration are assigned a default initial value:

- "zero" for numeric primitive types: 0 for *int*, *byte*, *short* and *long*, 0.0f for *float*, 0.0 for *double*;
- '\u0000' (*null* character) for *char*;
- *false* for *boolean*;
- *null* for reference type (such as array and object).

You can use them without assigning an initial value.

Automatic variables are not initialized, and must be explicitly assigned an initial value before it can be referenced. Failure to do so triggers a compilation error "variable xxx might not have been initialized".

7.4 Array_INITIALIZER

Array's elements are also initialized once the array is allocated (via the new operator). Like member variables, elements of primitive type are initialized to zero or false; while reference type are initialized to null. (Take note that C/C++ does not initialize array's elements.)

For example,



```
String[] strArray = new String[2];
2 for (String str: strArray) {
    System.out.println(str);
4 }
    // null
6    // null
```

You can also use the so-called *array initializer* to initialize the array during declaration. For example,



```
int[] numbers = {11, 22, 33};
2 String[] days = {"Monday", "Tuesday", "Wednesday"};
Circle[] circles = {new Circle(1.1), new Circle(2.2), new Circle(3.3)};
4 float[][] table = {{1.1f, 2.2f, 3.3f}, {4.4f, 5.5f, 6.6f}, {7.7f, 8.8f,
    ↪ 9.9f}};
int[][] data = {{1, 4, 8}, {2, 3}, {4, 8, 1, 5}};
```

7.5 Stack/Heap and Garbage Collector

Where Primitives and Objects Live?

Primitive types, such as int and double, are created in the program's method stack during compiled time for efficiency (less storage and fast access). Java's designer retained primitives in a object-oriented language for its efficiency.

Reference types, such as objects and arrays, are created in the "heap" at runtime (via the new operator), and accessed via a reference. Heap is less efficient as stack, as complex

memory management is required to allocate, manage and release storage.

For automatic variable of reference type: the reference is local (allocated in the method stack), but the object is allocated in the heap.

Stack and heap are typically located at the opposite ends of the data memory, to facilitate expansion.

Object References

When a Java object is constructed via the new operator and constructor, the constructor returns a value, which is a bit pattern that uniquely identifies the object. This value is known as the *object reference*.

In some JVM implementations, this object reference is simply the *address* of the object in the heap. However, the JVM specification does not specify how the object reference shall be implemented as long as it can uniquely identify the object. Many JVM implementations use so-called *double indirection*, where the object reference is the address of an address. This approach facilitates the garbage collector (to be explained next) to relocate objects in the heap to reduce memory fragmentation.

Objects are created via the new operator and the constructor. The new operator:

1. creates a new instance of the given class, and allocate memory dynamically from the heap;
2. calls one of the overloaded constructors to initialize the object created; and
3. returns the reference.

For primitives stored in the stack, compiler can determine how long the item lasts and destroy it once it is out of scope. For object in heap, the compiler has no knowledge of the creation and lifetime of the object.

In C++, you must destroy the heap's objects yourself in your program once the objects are no longer in use (via delete operator). Otherwise, it leads to a common bug known as "memory leak" - the dead objects pile-up and consume all the available storage. On the other hand, destroying an object too early, while it is still in use, causes runtime error. Managing memory explicitly is tedious and error prone, although the programs can be more efficient.

In Java, you don't have to destroy and de-allocate the objects yourself. JVM has a built-in process called *garbage collector* that automatically releases the memory for an object when there is no more reference to that object. The garbage collector runs in a low priority thread.

An object is eligible for garbage collection when there is no more references to that object. Reference that is held in a variable is dropped when the variable has gone out of its scope. You may also explicitly drop an object reference by setting the object reference to null to signal to the garbage collector it is available for collection. However, it may or may not get

garbage collected because there is no guarantee on when the garbage collector will be run or it will be run at all. The garbage collector calls the object's destructor (a method called `finalize()`), if it is defined, before releasing the memory back to the heap for re-use.

If a new reference is assigned to a reference variable (e.g., via `new` and constructor), the previous object will be available for garbage collection if there is no other references.

System.gc() & Runtime.gc()

You can explicitly ask for garbage collection by calling static methods `System.gc()` or `Runtime.gc()`. However, the behavior of these methods is JVM dependent. Some higher priority thread may prevent garbage collector from being run. You cannot rely on the `gc()` methods to perform garbage collection as the JVM specification merely states that "calling this method suggests that the Java Virtual Machine expends effort toward recycling unused objects". So the critical question "When the storage is recovered?" cannot be answered in Java.

Pitfalls of Java

Java's garbage collector frees you from worrying about memory management of objects (no more free or delete like C/C++) so that you can focus on more productive works. It also insures against so called "memory leak" (i.e., used objects were not de-allocated from memory and slowly fill up the precious memory space); or releasing object too early which results in runtime error. These are common problems in C/C++ programs.

However, garbage collector does has its drawbacks:

1. Garbage collector consumes computational resources and resulted in runtime overhead.
2. The rate of execution is not guarantee and can be inconsistent. This is because JVM specification does not spell out when and how long the garbage collector should be run. This may have an impact on real-time programs, when a response is expected within a certain time, which cannot be interrupted by the garbage collector.

Many programmers prefer to use C++ for game programming and animation, as these programs could create millions of objects in a short span. Managing memory efficiently is critical, instead of relying on garbage collector.

There are some (imperfect) solutions to memory management in Java, e.g.,

1. Pre-allocate and re-use the objects, instead of creating new objects. This requires effort from programmers.
2. The author of "jBullet", which is a Java port of the famous Collision Physics library "Bullet Physics", created a library called `jStackAlloc`, which allocates objects on the method's stack instead of program heap. This improves real-time performance by reducing the frequency of garbage collection.

This solution shall remain imperfect until the Java designers decided to allow programmers to manage the storage, which is not likely.

8 More on Methods

8.1 Passing Arguments into Methods - By Value vs. By Reference

Recall that a method receives *arguments* from the caller, performs operations defined in the method body, and returns a piece of result or void to the caller.

To differentiate the parameters inside and outside the method, we have:

- *Actual parameters* (or *arguments*): The actual values passed into the method and used inside the method.
- *Formal parameters* (or *method parameters*): The *placeholders* used in the method definition, which are replaced by the actual parameters when the method is invoked.

For example:



```
1 public static double getArea(double radius) { // radius is called formal
// parameter
3     return radius * radius * Math.PI;
4 }
5
6 public static void main(String[] args) {
7     double r = 1.2;
8     System.out.println(getArea(r)); // these are the actual parameter (or argument)
9     System.out.println(getArea(3.4));
10 }
```

In the above method definition, radius is a parameter placeholder or formal parameter. If we invoke the method with a variable r with value of 1.2, i.e., getArea(r), r = 1.2 is the actual parameter.

Passing Primitive-type Argument into Method - Pass-by-Value

If the argument is a primitive type (e.g., int or double), a copy of identical value is created and passed into the method. The method operates on the cloned copy. It does not have access to the original copy. If the value of the argument is changed inside the method, the original copy is not affected. This is called pass-by-value (passing a cloned value into the method).

For example,



```
1 public class TestPassingPrimitive {
2     public static void main(String[] args) {
3         int number = 10; // primitive type
4     }
5 }
```



```

4      System.out.println("In caller, before calling the method, the value is: "
        + number);
6      aMethodWithPrimitive(number); // invoke method
      System.out.println("In caller, after calling the method, the value is: " + number);
8  }

10     public static void aMethodWithPrimitive(int number) {
11         System.out.println("Inside method, before operation, the value is " + number);
12         ++number; // change the parameter
13         System.out.println("Inside method, after operation, the value is " + number);
14     }
  
```

Command window

```

1  In caller , before calling the method, the value is: 10
   Inside method, before operation , the value is 10
3  Inside method, after operation , the value is 11
   In caller , after calling the method, the value is: 10
  
```

Although the variables are called `number` in the caller as well as in the method's formal parameter, they are two different copies with their own scope.

Passing Reference-Type Argument into Method - Also Pass-by-Value

If the argument is a reference type (e.g., an array or an instance of a class), a copy of the reference is created and passed into the method. Since the caller's object and the method's parameter have the same reference, if the method changes the member variables of the object, the changes are permanent and take effect outside the method.

For example,



```

      public class TestParameterReference {
2      public static void main(String[] args) {
          StringBuffer sb = new StringBuffer("Hello");
4          System.out.println("In caller, before calling the method, the object is \" + sb
              + "\"");
6          aMethodOnReference(sb); // invoke method with side-effect
          System.out.println("In caller, after calling the method, the object is \" + sb
              + "\"");
8      }

10     public static void aMethodOnReference(StringBuffer sb) {
11         System.out.println("Inside method, before change, the object is \" + sb
12             + "\"");
  
```



```

14      sb.append(", world"); // change parameter
      System.out.println("Inside method, after change, the object is \" + sb
16      + "\"");
      }
18  }

```

Command window

```

      In caller , before calling the method, the object is "Hello"
2      Inside method, before change, the object is "Hello"
      Inside method, after change, the object is "world peace"
4      In caller , after calling the method, the object is "Hello"

```

Reference-Type Argument - Pass-by-Reference or Pass-by-value?

As the object parameter can be modified inside the method, some people called it *pass-by-reference*. However, in Java, a *copy of reference* is passed into the method, hence, Java designers called it *pass-by-value*.

Passing a Primitive as a One-Element Array?

Primitive-type parameters are passed-by-value. Hence, the method is not able to modify the caller's copy. If you wish to let the method to modify the caller's copy, you might pass the primitive-type parameter as a one-element array, which is not recommended.

8.2 Method Overloading vs. Overriding

An overriding method must have the same argument list; while an overloading method must have different argument list. You override a method in the subclass. You typically overload a method in the same class, but you can also overload a method in the subclass.

A overriding method:

1. must have the same parameter list as it original.
2. must have the same return-type or sub-type of its original return-type (since JDK 1.5 - called *covariant return-type*).
3. cannot have more restrictive access modifier than its original, but can be less restrictive, e.g., you can override a protected method as a public method.
4. cannot throw more exceptions than that declared in its original, but can throw less exceptions. It can throw exceptions that is declared in its original or their sub-types.
5. overriding a private method does not make sense, as private methods are not really inherited by its subclasses.
6. You cannot override a non-static method as static, and vice versa.

7. Technically, a subclass does not override a static method, but merely hides it. Both the superclass' and subclass' versions can still be accessed via the classnames.
8. A final method cannot be overridden. An abstract method must be overridden in an implementation subclass (otherwise, the subclass remains abstract).

A overloading method:

1. must be differentiated by its parameter list. It shall not be differentiated by return-type, exception list or access modifier (which generates compilation error). It could have any return-type, exception list or access modifier, as long as it has a different parameter list than the others.
2. can exist in the original class or its sub-classes.

9 Frequently-Used Packages in JDK API

JDK API is huge and consists of many packages (refer to JDK API specification). These are the frequently-used packages:

- `java.lang` (the core JDK package): contains classes that are core to the language, e.g., `System`, `String`, `Math`, `Object`, `Integer`, and etc.
- `java.util`: contains utilities such as `Scanner`, `Random`, `Date`, `ArrayList`, `Vector`, `Hashtable`.
- `java.io`: contains input and output classes for reading files and I/O streams, such as `File`.
- `java.net`: contains networking support, such as `Socket` and `URL`.
- `java.awt` (Abstract Windowing Toolkit): contains classes for implementing a graphical user interface, including classes like `Frame`, `Button`, `CheckBox`.
- `java.awt.event`: contains event handling classes, such as key-press, mouse-click etc.
- `java.swing`: Advanced GUI classes, e.g., `JFrame`, `JButton`, `JApplet`, etc.
- `java.applet`: contains classes for implementing Java applets.
- `java.sql`: contains classes for database programming, such as `Connection`, `Statement`, `ResultSet`.
- Many others.

10 Java String

10.1 A Brief Summary of the String Class

A Java String contains an immutable sequence of Unicode characters. Unlike C/C++, where string is simply an array of char, A Java String is an object of the class `java.lang.String`.

Java String is, however, special. Unlike an ordinary class:

- String is associated with *string literal* in the form of double-quoted texts such as "hello, world". You can assign a string literal directly into a String variable, instead of calling the constructor to create a String instance.
- The '+' operator is overloaded to concatenate two String operands. '+' does not work on any other objects such as Point and Circle.
- String is *immutable*. That is, its content cannot be modified once it is created. For example, the method `toUpperCase()` constructs and returns a new String instead of modifying the existing content.

10.1.1 Method Summary

The commonly-used method in the String class are summarized below. Refer to the JDK API for `java.lang.String` a complete listing.



```

// Length
2  int length()           // returns the length of the String
   boolean isEmpty()     // same as str.length() == 0
4  boolean isBlank()     // contains only white spaces (Unicode aware) (JDK 11)

// Comparison
6  boolean equals(String another) // CANNOT use '==' or '!=' to compare two Strings
   // in Java
8  boolean equalsIgnoreCase(String another)
10 int compareTo(String another) // return 0 if this string is the same as another;
   // < 0 if lexicographically less than another; or > 0
12 int compareToIgnoreCase(String another)
   boolean startsWith(String another)
14 boolean startsWith(String another, int fromIdx) // search begins at fromIdx
   boolean endsWith(String another)

16 // Searching: index from 0 to str.length()-1
18 int indexOf(String key)
   int indexOf(String key, int fromIdx)
20 int indexOf(int char)
   int indexOf(int char, int fromIdx) // search forward starting at fromIdx
22 int lastIndexOf(String key)
   int lastIndexOf(String key, int fromIdx) // search backward starting at fromIdx
24 int lastIndexOf(int char)
   int lastIndexOf(int char, int fromIdx)

```



```

26 // Extracting a char or substring, include fromIdx but exclude toIdx
28 char charAt(int idx)
29 String substring(int fromIdx)
30 String substring(int fromIdx, int toIdx)

32 // Creating a new String or char[] from the original - Strings are immutable
33 String toLowerCase()
34 String toUpperCase()
35 String concat(String another) // same as str+another
36 String trim() // creates a new String removing white spaces from front and back
37 String strip() // strips the leading and trailing white spaces (Unicode aware) (JDK 11)
38 String stripLeading() // (JDK 11)
39 String stripTrailing() // (JDK 11)
40 String repeat(int count) // (JDK 11)
41 String indent(int n) // adjusts the indentation by n (JDK 12)
42 char[] toCharArray() // create a char[] from this string

44 // copy into dst char[]
45 void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)

46 // Working with CharSequence (super-interface of String, StringBuffer, StringBuilder)
47 boolean contains(CharSequence cs) // (JDK 5)
48 boolean contentEquals(CharSequence cs) // (JDK 5)
49 boolean contentEquals(StringBuffer sb) // (JDK 4)

51 // (JDK 8)
52 static String join(CharSequence delimiter, CharSequence... elements)
53 static String join(CharSequence delimiter, Iterable<CharSequence>
    ↪ elements)

56 // Text Processing and Regular Expression (JDK 4)
57 boolean matches(String regex)
58 String replace(char old, char new)
59 String replace(CharSequence target, CharSequence replacement) // (JDK 4)
60 String replaceAll(String regex, String replacement)
61 String replaceFirst(String regex, String replacement)

62 // Split the String using regex as delimiter, return a String array
63 String[] split(String regex)
64 String[] split(String regex, int count) // for count times only

66 /*** static methods ***/
67 // Converting primitives to String
68 static String valueOf(type arg) // type can be primitives or char[]
69 // Formatting using format specifiers
70 static String format(String formattingString, Object... args) // same
71 // as printf()

74 /*** Stream and Functional Programming ***/
75 Stream<String> lines() // returns a stream of lines (JDK 11)
76 IntStream chars() // returns a IntStream of characters (JDK 9)

```



```
78 IntStream codePoints()
    R transform(Function<String, R> f) // transforms from String to type R (JDK 12)
```

10.1.2 Examples

static method `String.format()` (JDK 5)

The static method `String.format()` (introduced in JDK 5) can be used to produce a formatted String using C-like `printf()`'s format specifiers. The `format()` method has the same form as `printf()`. For example,



```
String.format("%.1f", 1.234); // returns String "1.2"
```

`String.format()` is useful if you need to produce a simple formatted String for some purposes (e.g., used in method `toString()`). For complex string, use `StringBuffer/StringBuilder` with a `Formatter`. If you simply need to send a simple formatted string to the console, use `System.out.printf()`, e.g.,



```
1 System.out.printf("%.1f", 1.234);
```

10.1.3 New Methods

JDK 9 new methods

- `.chars().codePoints()` → `IntStream`

JDK 11 new methods

- `.repeat(int count)` → `String`
- `.strip().stripLeading().stripTrailing()` → `String` and `.isBlank()` → `boolean` which are unicode white-space aware.
- `.lines()` to produces a `Stream<String>`. (JDK 9 added `.chars()` and `.codePoints()` to produce an `IntStream`.)

JDK 12 new methods

- `.indent(int n)` → `String`
- `.transform(Function<String, R> f)` → `R`
- `.describeConstable()` → `Optional<String>`
- `.resolveConstantDesc()` → `String`

10.2 String is Really Special!

Strings receive special treatment in Java, because they are used frequently in a program. Hence, efficiency (in terms of computation and storage) is crucial.

The designers of Java decided to retain primitive types in an object-oriented language, instead of making everything an object, so as to improve the performance of the language. Primitives are stored in the method stack, which require less storage spaces and are cheaper to manipulate. On the other hand, objects are stored in the program heap, which require complex memory management and more storage spaces.

For performance reason, Java's String is designed to be in between a primitive and an object. The special features in String include:

- The '+' operator, which performs addition on primitives (such as int and double), is overloaded to operate on String objects. '+' performs concatenation for two String operands. Java does not support *operator overloading* for software engineering consideration. In a language that supports operator overloading like C++, you can turn a '+' operator to perform a subtraction, resulted in poor codes. The '+' operator is the only operator that is internally overloaded to support string concatenation in Java. Take note that '+' does not work on any two arbitrary objects, such as Points or Circles.
- A String can be constructed by either:
 - directly assigning a string literal to a String reference - just like a primitive, or
 - via the "new" operator and constructor, similar to any other classes. However, this is not commonly-used and is not recommended.

For example,



```
1 String str1 = "Java is Hot"; // Implicit construction via string literal
   String str2 = new String("I'm cool"); // Explicit construction via new
```

In the first statement, str1 is declared as a String reference and initialized with a string literal "Java is Hot". In the second statement, str2 is declared as a String reference and initialized via the new operator and constructor to contain "I'm cool".

- String literals are stored in a *common pool*. This facilitates *sharing of storage* for strings with the same contents to conserve storage. String objects allocated via new operator are stored in the heap, and there is no sharing of storage for the same contents.

10.2.1 String Literal vs. String Object

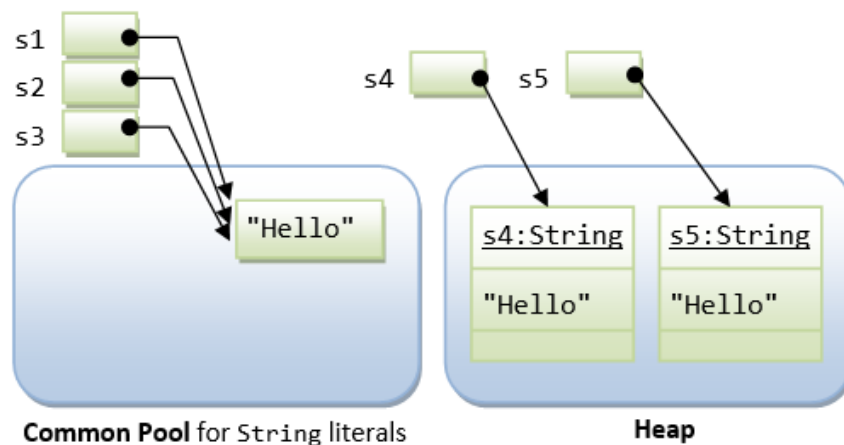
As mentioned, there are two ways to construct a string: implicit construction by assigning a string literal or explicitly creating a String object via the new operator and constructor. For example,



```

1  String s1 = "Hello";           // String literal
   String s2 = "Hello";           // String literal
3  String s3 = s1;                 // same reference
   String s4 = new String("Hello"); // String object
5  String s5 = new String("Hello"); // String object

```



Java has provided a special mechanism for keeping the String literals - in a so-called *string common pool*. If two string literals have the same contents, they will share the same storage inside the common pool. This approach is adopted to *conserve storage* for frequently-used strings. On the other hand, String objects created via the new operator and constructor are kept in the heap. Each String object in the heap has its own storage just like any other object. There is no sharing of storage in heap even if two String objects have the same contents.

You can use the method `equals()` of the String class to compare the contents of two Strings. You can use the relational equality operator `'=='` to compare the references (or pointers) of two objects. Study the following codes:



```

1  s1 == s1;           // true, same pointer
   s1 == s2;           // true, s1 and s2 share storage in common pool
3  s1 == s3;           // true, s3 is assigned same pointer as s1
   s1.equals(s3);      // true, same contents
5  s1 == s4;           // false, different pointers
   s1.equals(s4);      // true, same contents
7  s4 == s5;           // false, different pointers in heap
   s4.equals(s5);      // true, same contents

```

Important Notes:

- In the above example, I used relational equality operator '==' to compare the references of two String objects. This is done to demonstrate the differences between string literals sharing storage in the common pool and String objects created in the heap. *It is a logical error to use (str1 == str2) in your program to compare the contents of two Strings.*
- String can be created by directly assigning a String literal which is shared in a common pool. It is uncommon and not recommended to use the new operator to construct a String object in the heap.

10.2.2 String is Immutable

Since string literals with the same contents share storage in the common pool, Java's String is designed to be immutable. That is, once a String is constructed, its contents cannot be modified. Otherwise, the other String references sharing the same storage location will be affected by the change, which can be unpredictable and therefore is undesirable. Methods such as toUpperCase() might appear to modify the contents of a String object. In fact, a completely new String object is created and returned to the caller. The original String object will be deallocated, once there is no more references, and subsequently garbage-collected.

Because String is immutable, it is not efficient to use String if you need to modify your string frequently (that would create many new Strings occupying new storage areas). For example,



```
// inefficient codes
2 String str = "Hello";
  for (int i = 1; i < 1000; ++i) {
4     str = str + i;
  }
```

If the contents of a String have to be modified frequently, use the StringBuffer or StringBuilder class instead.

10.2.3 String.intern()

When a string is created in Java, it occupies memory in the heap. Also, we know that the String class is immutable. Therefore, whenever we create a string using the new keyword, new memory is allocated in the heap for corresponding string, which is irrespective of the content of the array. Consider the following code snippet.



```
1 String str = new String("Hello");
  String str1 = new String("Hello");
3 System.out.println(str1 == str); // prints false
```

The println statement prints false because separate memory is allocated for each string literal. Thus, two new string objects are created in the memory i.e. str and str1. that holds different references.

We know that creating an object is a costly operation in Java. Therefore, to save time, Java developers came up with the concept of String Constant Pool (SCP). The SCP is an area inside the heap memory. It contains the unique strings. In order to put the strings in the string pool, one needs to call the intern() method. Before creating an object in the string pool, the JVM checks whether the string is already present in the pool or not. If the string is present, its reference is returned.



```
1 String str = new String("Hello").intern(); // statement - 1
  String str1 = new String("Hello").intern(); // statement - 2
3 System.out.println(str1 == str);           // prints true
```

In the above code snippet, the intern() method is invoked on the String objects. Therefore, the memory is allocated in the SCP. For the second statement, no new string object is created as the content of str and str1 are the same. Therefore, the reference of the object created in the first statement is returned for str1. Thus, str and str1 both point to the same memory. Hence, the print statement prints true.

Example:

InternExample.java



```
1 public class InternExample{
  public static void main(String[] args) {
3     String str1 = new String("Hello");
      String str2 = "Hello";
5     String str3 = str1.intern(); // returns string from pool,
                                   // now it will be same as str2
7     System.out.println(str1 == str2); // false because reference variables are
                                         // pointing to different instance
9     System.out.println(str2 == str3); // true because reference variables are
                                         // pointing to same instance
11 }
```




```
Command window
false
2 true
```

Points to Remember

1. A string literal always invokes the intern() method, whether one mention the intern() method along with the string literal or not. For example,



```
String str1 = "d".intern();
2 String str2 = "d"; // compiler treats it as String str2 = "d".intern();
System.out.println(str1 == str2); // prints true
```

2. Whenever we create a String object using the new keyword, two objects are created. For example,



```
String str = new ("Hello World");
```

Here, one object is created in the heap memory outside of the SCP because of the usage of the new keyword. As we have got the string literal too ("Hello World"); therefore, one object is created inside the SCP, provided the literal "Hello World" is already not present in the SCP.

10.3 StringBuffer & StringBuilder

As explained earlier, Strings are immutable because String literals with same content share the same storage in the string common pool. Modifying the content of one String directly may cause adverse side-effects to other Strings sharing the same storage.

JDK provides two classes to support mutable strings: StringBuffer and StringBuilder (in core package java.lang). A StringBuffer or StringBuilder object is just like any ordinary object, which are stored in the heap and not shared, and therefore, can be modified without

causing adverse side-effect to other objects.

StringBuilder class was introduced in JDK 5. It is the same as StringBuffer class, except that StringBuilder is not synchronized for multi-thread operations. However, for single-thread program, StringBuilder, without the synchronization overhead, is more efficient.

10.3.1 java.lang.StringBuffer

Read the JDK API specification for java.lang.StringBuffer.



```

// Constructors
2  StringBuffer()           // an initially-empty StringBuffer
   StringBuffer(int size)   // with the specified initial size
4  StringBuffer(String s)   // with the specified initial content

6  // Length
   int length()

8

// Methods for building up the content
10 StringBuffer append(type arg) // type could be primitives, char[], String,
   StringBuffer, etc
   StringBuffer insert(int offset, arg)

12

// Methods for manipulating the content
14 StringBuffer delete(int fromIdx, int toIdx)
   StringBuffer deleteCharAt(int idx)
16 void setLength(int newSize)
   void setCharAt(int idx, char newChar)
18 StringBuffer replace(int fromIdx, int toIdx, String s)
   StringBuffer reverse()

20

// Methods for extracting whole/part of the content
22 char charAt(int idx)
   String substring(int fromIdx)
24 String substring(int fromIdx, int toIdx)
   String toString()

26

// Indexing
28 int indexOf(String key)
   int indexOf(String key, int fromIdx)
30 int lastIndexOf(String key)
   int lastIndexOf(String key, int fromIdx)

```

Take note that StringBuffer is an ordinary object. You need to use a constructor to create a StringBuffer (instead of assigning to a String literal). Furthermore, '+' operator does not apply to objects, including the StringBuffer. You need to use a proper method such as

append() or insert() to manipulating a StringBuffer.

To create a string from parts, It is more efficient to use StringBuffer (multi-thread) or StringBuilder (single-thread) instead of via String concatenation. For example,



```
1 // Create a string of YYYY-MM-DD HH:MM:SS
  int year = 2010, month = 10, day = 10;
3 int hour = 10, minute = 10, second = 10;
  StringBuffer dateStr = new StringBuffer()
5     .append(year).append("-").append(month).append("-").append(day)
     .append(" ").append(hour).append(":").append(minute).append(":")
7     .append(second).toString();
  System.out.println(dateStr);
9
11 // StringBuilder is more efficient than String concatenation
  String anotherDataStr = year + "-" + month + "-" + day + " " + hour
13     + ":" + minute + ":" + second;
  System.out.println(anotherDataStr);
```

JDK compiler, in fact, uses both String and StringBuffer to handle string concatenation via the '+' operator. For examples,



```
1 String msg = "a" + "b" + "c";
```

will be compiled into the following codes for better efficiency:



```
1 String msg = new StringBuffer().append("a").append("b").append("c")
  .toString();
```

Two objects are created during the process, an intermediate StringBuffer object and the returned String object.

Rule of Thumb: Strings are more efficient if they are not modified (because they are shared in the string common pool). However, if you have to modify the content of a string frequently (such as a status message), you should use the StringBuffer class (or the StringBuilder described below) instead.

10.3.2 java.lang.StringBuilder (JDK 5)

JDK 5 introduced a new `StringBuilder` class (in package `java.lang`), which is almost identical to the `StringBuffer` class, except that it is *not synchronized*. In other words, if multiple threads are accessing a `StringBuilder` instance at the same time, its integrity cannot be guaranteed. However, for a single-thread program (most commonly), doing away with the overhead of synchronization makes the `StringBuilder` faster.

`StringBuilder` is API-compatible with the `StringBuffer` class, i.e., having the same set of constructors and methods, but with no guarantee of synchronization. It can be a drop-in replacement for `StringBuffer` under a *single-thread* environment.

10.3.3 Benchmarking String/StringBuffer/StringBuilder

The following program compare the times taken to reverse a long string via a `String` object and a `StringBuffer`.



```
// Reversing a long String via a String vs. a StringBuffer
2 public class StringsBenchMark {
    public static void main(String[] args) {
4        // Build a long string
        String str = "";
6        int size = 16536;
        char ch = 'a';
8        long beginTime = System.nanoTime(); // Reference time in nanoseconds
        for (int count = 0; count < size; ++count) {
10            str += ch;
            ++ch;
12            if (ch > 'z') {
                ch = 'a';
14            }
        }
16        long elapsedTime = System.nanoTime() - beginTime;
        System.out.println("Elapsed Time is " + elapsedTime/1000
18            + " usec (Build String)");

20        // Reverse a String by building another String character-by-character in the reverse order
        String strReverse = "";
22        beginTime = System.nanoTime();
        for (int pos = str.length() - 1; pos >= 0 ; pos--) {
24            strReverse += str.charAt(pos); // Concatenate
        }
26        elapsedTime = System.nanoTime() - beginTime;
        System.out.println("Elapsed Time is " + elapsedTime/1000
28            + " usec (Using String to reverse)");

30        // Reverse a String via an empty StringBuffer by appending characters in the reverse order
        beginTime = System.nanoTime();
32        StringBuffer sBufferReverse = new StringBuffer(size);
```



```

34     for (int pos = str.length() - 1; pos >= 0 ; pos--) {
        sBufferReverse.append(str.charAt(pos)); // append
    }
36     elapsedTime = System.nanoTime() - beginTime;
    System.out.println("Elapsed Time is " + elapsedTime/1000
38         + " usec (Using StringBuffer to reverse)");

40     // Reverse a String by creating a StringBuffer with the given String and invoke its reverse()
    beginTime = System.nanoTime();
42     StringBuffer sBufferReverseMethod = new StringBuffer(str);
    sBufferReverseMethod.reverse(); // use reverse() method
44     elapsedTime = System.nanoTime() - beginTime;
    System.out.println("Elapsed Time is " + elapsedTime/1000
46         + " usec (Using StringBuffer's reverse() method)");

48     // Reverse a String via an empty StringBuilder by appending characters in the reverse
        order
    beginTime = System.nanoTime();
50     StringBuilder sBuilderReverse = new StringBuilder(size);
    for (int pos = str.length() - 1; pos >= 0 ; pos--) {
52         sBuilderReverse.append(str.charAt(pos));
    }
54     elapsedTime = System.nanoTime() - beginTime;
    System.out.println("Elapsed Time is " + elapsedTime/1000
56         + " usec (Using StringBuilder to reverse)");

58     // Reverse a String by creating a StringBuilder with the given String and invoke its
        reverse()
    beginTime = System.nanoTime();
60     StringBuffer sBuilderReverseMethod = new StringBuffer(str);
    sBuilderReverseMethod.reverse();
62     elapsedTime = System.nanoTime() - beginTime;
    System.out.println("Elapsed Time is " + elapsedTime/1000
64         + " usec (Using StringBuidler's reverse() method)");
    }
66 }

```

Command window

```

Elapsed Time is 332100 usec (Build String)
2 Elapsed Time is 346639 usec (Using String to reverse)
Elapsed Time is 2028 usec (Using StringBuffer to reverse)
4 Elapsed Time is 847 usec (Using StringBuffer's reverse() method)
Elapsed Time is 1092 usec (Using StringBuilder to reverse)
6 Elapsed Time is 836 usec (Using StringBuidler's reverse() method)

```

Observe StringBuilder is 2x faster than StringBuffer, and 300x faster than String. The reverse() method is the fastest, which take about the same time for StringBuilder and StringBuffer.

10.4 java.util.StringTokenizer (Obsoleted by regex)

Very often, you need to break a line of texts into tokens delimited by white spaces. The `java.util.StringTokenizer` class supports this.

For example, the following program reverses the words in a `String`.



```
import java.util.StringTokenizer;
2  /**
   * Reverse the words in a String using StringTokenizer
   */
4  public class StringTokenizerTest {
6      public static void main(String[] args) {
            String str = "Monday Tuesday Wednesday Thursday Friday Saturday Sunday";
8            String strReverse;
            StringBuilder sb = new StringBuilder();
10           StringTokenizer st = new StringTokenizer(str);

12           while (st.hasMoreTokens()) {
                sb.insert(0, st.nextToken());
14                 if (st.hasMoreTokens()) {
                    sb.insert(0, " ");
16                 }
            }
18           strReverse = sb.toString();
            System.out.println(strReverse);
20       }
    }
```



```
1  // Constructors
   StringTokenizer(String s) // Constructs a StringTokenizer for the given string,
3  // using the default delimiter set of " \t\n\r\f"
   // (i.e., blank, tab, newline, carriage-return, and form-feed).
5  // Delimiter characters themselves will not be treated as tokens.
   StrintTokenizer(String s, String delimiterChars) // Use characters in
7                                                    // delimiterSet as delimiters.

9  // Methods
   boolean hasNextToken() // Returns true if next token available
11  String nextToken() // Returns the next token
```

For example,



```
1 // Code Sample
2 StringTokenizer tokenizer = new StringTokenizer(aString);
3 while (tokenizer.hasNextToken()) {
4     String token = tokenizer.nextTokn();
5     .....
6 }
```

The JDK documentation stated that "StringTokenizer is a legacy class that is retained for compatibility reasons although its use is discouraged in new code. It is recommended that anyone seeking this functionality use the split() method of String or the java.util.regex package instead."

For example, the following program uses the split() method of the String class to reverse the words of a String.



```
1 /**
2  * Reverse the words in a String using split() method of the String class
3  */
4 public class StringSplitTest {
5     public static void main(String[] args) {
6         String str = "Monday Tuesday Wednesday Thursday Friday Saturday Sunday";
7         String[] tokens = str.split("\\s"); //white space '\s' as delimiter
8         StringBuilder sb = new StringBuilder();
9         for (int i = 0; i < tokens.length; ++i) {
10             sb.insert(0, tokens[i]);
11             if (i < tokens.length - 1) {
12                 sb.insert(0, " ");
13             }
14         }
15         String strReverse = sb.toString();
16         System.out.println(strReverse);
17     }
18 }
```

10.5 Regular Expression (Regex), Patterns & Matches (JDK 4)

[TODO]

10.6 Super-Interface CharSequence for String, StringBuffer and StringBuilder (since JDK 4)

The interface `java.lang.CharSequence` is implemented by classes `String`, `StringBuffer`, `StringBuilder`, `CharBuffer` and `Segment`.

It defines the common behavior via these abstract methods:



```
// java.lang.CharSequence
2 abstract charAt(int index) -> char
  abstract length() -> int
4 abstract subSequence(int fromIdx, int toIdx) -> CharSequence
  abstract toString() -> String
```

JDK 8 added two default methods into the interface:



```
1 // java.lang.CharSequence (JDK 8)
  default chars() -> IntStream
3 default codePoints() -> IntStream
```

JDK 11 added one static methods into the interface:



```
1 // java.lang.CharSequence (JDK 11)
  static compare(CharSequence cs1, CharSequence cs2) -> int
```

11 Package java.lang Frequently-used Classes

"java.lang" is the Java core language package, which contains classes central to the Java language. It is implicitly "imported" into every Java program. That is, no explicit "import" statement is required for using classes in java.lang.

Frequently-used classes in "java.lang" are:

- `String`, `StringBuffer` and `StringBuilder`: `String` is immutable whereas `StringBuffer`/`StringBuilder` is mutable. `StringBuffer` is thread-safe; while `StringBuilder` is not thread-safe and is meant for single-thread operations.
- `Math`: contains public static final fields `PI` and `E`; and many public static methods such as `random()`, `sqrt()`, `sin()`, `cos()`, `asin()`, `acos()`, `log()`, `exp()`, `floor()`, `ceil()`, `pow()`,

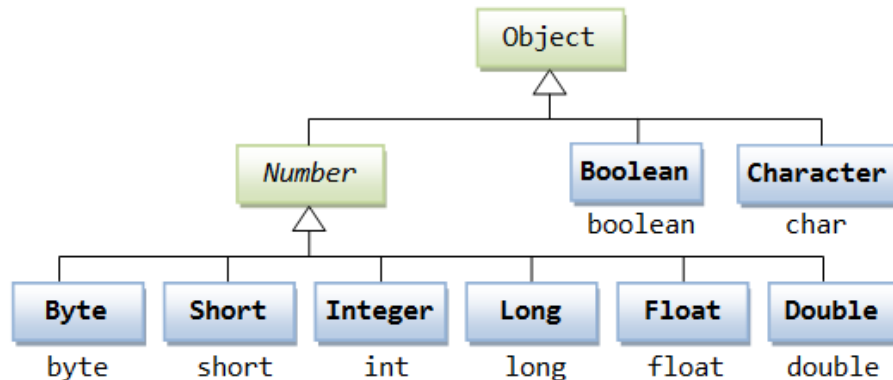
and etc.

- Wrapper class for primitive types: Byte, Integer, Short, Long, Float, Double, Character, and Boolean. The wrapper class is used to wrap a primitive type into a Java class. They are used when a class is needed for purpose such as using multithreading, synchronization and collection. They also contains static utility methods (such as Integer.parseInt()) and static constants (such as Integer.MAX_VALUE).
- System: contains the static variables in, out, and err, corresponds to the standard input, output, and error streams.
- Object: the common root class for all the Java classes. This common root class defines the baseline behaviors needed to support features like multithreading (lock and monitor), synchronization (wait(), notify(), notifyAll()), garbage collection, equals(), hashCode() and toString().

11.1 java.lang.String, StringBuilder & StringBuffer

Read "Java String".

11.2 Wrapper Classes for Primitive Types



The designers of Java language retain the primitive types in an object-oriented language, instead of making everything object, so as to improve the runtime efficiency and performance. However, in some situations, an object is required instead of a primitive value. For examples,

- The data structures in the Collection framework, such as the "dynamic array" ArrayList and Set, stores only objects (reference types) and not primitive types.
- Object is needed to support synchronization in multithreading.
- Objects are needed, if you wish to modify the arguments passed into a method (because primitive types are passed by value).

JDK provides the so-called wrapper classes that wrap primitive values into objects, for each of the eight primitive types - Byte for byte, Short for short, Integer for int, Long for long, Float for float, Double for double, Character for char, and Boolean for boolean, as shown in the class diagram.

Wrapper Classes are Immutable

Each of the wrapper classes contains a private member variable that holds the primitive value it wraps. The wrapped value cannot be changed. In other words, all the wrapper classes are *immutable*.

Wrap via Constructors

Each of the wrapper classes has a constructor that takes in the data type it wraps. For examples:



```
// Wrap an int primitive value into an Integer object
2 Integer aIntObj = new Integer(5566);
// Wrap a double primitive value into a Double object
4 Double aDoubleObj = new Double(55.66);
// Wrap a char primitive value into a Character object
6 Character aCharObj = new Character('z');
// Wrap a boolean primitive value into a Boolean object
8 Boolean aBooleanObj = new Boolean(true);
```

All wrapper classes, except Character, also have a constructor that takes a String, and parse the String into the primitive value to be wrapped.

Static factory method `valueOf()` (JDK 5)

The constructors had been deprecated in JDK 9. You should use static factory method `valueOf()` to construct an instance.

For examples, the following `valueOf()` are defined in the Integer class:



```
// java.lang.Integer
2 public static Integer valueOf(int i)
  public static Integer valueOf(String s)
4 public static Integer valueOf(String s, int radix)
```

For example,



```

public class WrapperClassTest {
2   public static void main(String[] args) {
        // Integer iObj1 = new Integer(11);
4        // warning: [deprecation] Integer(int) in Integer has been deprecated

        Integer iObj2 = Integer.valueOf(22);
        System.out.println(iObj2); //22
        Integer iObj3 = Integer.valueOf("33");
        System.out.println(iObj3); //33
10       Integer iObj4 = Integer.valueOf("1ab", 16); // radix
        System.out.println(iObj4); //427

12       Integer iObj5 = 44; // auto-box from int to Integer
14       int i5 = iObj5; // auto-unbox from Integer to int
        System.out.println(i5); // 44
16   }
}

```

Unwrap via xxxValue() methods

The abstract superclass Number defines the following xxxValue() methods to unwrap, which are implemented in concrete subclasses Byte, Short, Integer, Long, Float, Double. In other words, you can get an int or double value from an Integer object.



```

1 // In classes Byte, Short, Integer, Long, Float and Double
   public byte byteValue() //returns the wrapped "numeric" value as a byte
3   public short shortValue() // ... as a short
   public abstract int intValue() // ... as an int
5   public abstract long longValue() // ... as a long
   public abstract float floatValue() // ... as a float
7   public abstract double doubleValue() // ... as a double

```

Similarly, the Character and Boolean classes have a charValue() and booleanValue(), respectively.



```

1 // In Character class
   public char charValue() // Returns as char
3 // In Boolean class
   public boolean booleanValue() // Returns as boolean

```

Example



```

// Wrap a primitive int into an Integer object
2 Integer intObj = new Integer(556677);
// Unwrap
4 int i = intObj.intValue();
short s = intObj.shortValue(); // truncate
6 byte b = intObj.byteValue(); // truncate

8 // Wrap a primitive double into an Double object
Double doubleObj = new Double(55.66);
10 // Unwrap
double d = doubleObj.doubleValue();
12 int i1 = doubleObj.intValue(); // truncate

14 // Wrap a primitive char into an Character object
Character charObj = new Character('z');
16 // Unwrap
char c = charObj.charValue();

18 // Wrap a primitive boolean into a Boolean object
20 Boolean booleanObj = new Boolean(false);
// Unwrap
22 boolean b1 = booleanObj.booleanValue();

```

Constants - MIN_VALUE, MAX_VALUE and SIZE

All wrapper classes (except Boolean) contain the following constants, which give the minimum, maximum, and bit-length.



```

// All except Boolean
2 public static final type MIN_VALUE // Minimum value
public static final type MAX_VALUE // Maximum value
4 public static final int SIZE // Number of bits
// Float and Double only
6 public static final int MAX_EXPONENT // Maximum exponent
public static final int MIN_EXPONENT // Maximum exponent

```

For examples:



```

1 // Integer class
System.out.println(Integer.MAX+VALUE); // 2147483647
3 System.out.println(Integer.MIN_VALUE); // -2147483648
System.out.println(Integer.SIZE); // 32
5 // Double class

```



```

System.out.println(Double.MAX_VALUE);    // 1.7976931348623157E308
7  System.out.println(Double.MIN_VALUE);  // 4.9E-324
System.out.println(Double.SIZE);         // 64
9  System.out.println(Double.MAX_EXPONENT); // 1023
System.out.println(Double.MIN_EXPONENT); // -1022

```

Static Methods for Parsing Strings

Each of the wrapper classes (except Character) also contain a static method to parse a given String into its respective primitive value:



```

// Byte class
2  public static byte  parseByte(String s) throws NumberFormatException
// Short class
4  public static short parseShort(String s) throws NumberFormatException
// Integer class
6  public static int   parseInt(String s) throws NumberFormatException
// Long class
8  public static long  parseLong(String s) throws NumberFormatException
// Float class
10 public static float  parseFloat(String s) throws NumberFormatException
// Double class
12 public static double parseDouble(String s) throws NumberFormatException
// Boolean class
14 public static boolean parseBoolean(String s)
// returns true for string "true" (case insensitive); returns false otherwise

```

For examples:



```

1  // Parse a String into int. Throw NumberFormatException if the String is not valid
   int i = Integer.parseInt("5566");
3  i = Integer.parseInt("abcd");    // Runtime Error: NumberFormatException
   i = Integer.parseInt("55.66");   // Runtime Error: NumberFormatException
5
   // Parse a String into double
7  double d = Double.parseDouble("55.66");

```

11.3 Auto-Boxing & Auto-Unboxing (JDK 1.5)

Prior to JDK 1.5, the programmers have to explicitly wrap a primitive value into an object, and explicitly unwrap an object to get a primitive value. For example,



```

1 // Pre-JDK 1.5
  Integer intObj = new Integer(5566); // wrap int to Integer
3 int i = intObj.intValue();         // unwrap Integer to int

5 Double doubleObj = new Double(55.66); // wrap double to Double
  double d = doubleObj.doubleValue(); // unwrap Double to double

```

The pre-JDK 1.5 approach involves quite a bit of code to do the wrapping and unwrapping. Why not ask the compiler to do the wrapping and unwrapping automatically? JDK 1.5 introduces a new feature called auto-boxing and unboxing, where the compiler could do the wrapping and unwrapping automatically for you based on their contexts. For example:



```

// Java SE 5.0
2 Integer intObj = 5566; // autobox from int to Integer
  int i = intObj;        // auto-unbox from Integer to int

4
  Double doubleObj = 55.66; // autoboxing from double to Double
6 double d = doubleObj;     // auto-unbox from Double to double

```

With the auto-boxing and unboxing, you can practically ignore the distinction between a primitive and its wrapper object.

11.4 java.lang.Math - Mathematical Functions & Constants

The java.lang.Math class provides mathematical constants (PI and E) and functions (such as random(), sqrt()). A few functions are listed below for references. Check the JDK API specification for details.



```

// static constants
2 public static double Math.PI;
  public static double Math.E;
4 // static methods
  public static double Math.random(); // generate a random number btw 0.0 & 1.0
6 public static double Math.sin(double x); // sine function
  public static double Math.exp(double x); // exponential function
8 public static double Math.log(double x); // natural logarithm of x
  public static double Math.pow(double x, double y); // x raised to power of y
10 public static double Math.sqrt(double x); // square root of x

```

For examples:



```
double radius = 1.1;
2 double area = radius * radius * Math.PI;
int number = (int) Math.pow(2, 3); // int 2 and 3 implicitly promoted to double
4 // invoke pow(double, double) which return a double
// cast the result back to an int
```

Take note that Math class is final - you cannot create subclasses. The constructor of Math class is private - you cannot create instances.

11.5 java.lang.Object - The Common Java Root Class

java.lang.Object is the superclass of all Java classes. In other words, all classes are subclass of Object - directly or indirectly. A reference of class Object can hold any Java object, because all Java classes are subclasses of Object. In other word, every Java class "is a" Object.

Java adopts a *single common root class* approach in its design, to ensure that all Java classes have a set of common baseline properties. The Object class defines and implements all these common attributes and behaviors that are necessary of all the Java objects running under the JVM. For example,

- Ability to compare itself to another object, via equals() and hashCode().
- Provides a text string description, via toString().
- Inter-thread communication, via wait(), notify() and notifyAll().
- Automatic garbage collection.

The Object class has the following public methods:



```
1 // The following methods must be overridden to be used
public boolean equals(Object obj);
3 public int hashCode();

5 // The following methods may be overridden
protected Object clone();
7 protected void finalize();
public String toString();

9
// The following methods are final and cannot be overridden
11 public final Class getClass();
public final void wait(...);
13 public final void notify();
public final void notifyAll();
```

- The method `getClass()` returns a runtime representation of the class in a `Class` object. A `Class` object exists for all the objects in Java. It can be used, for example, to discover the fully-qualified name of a class, its members, its immediate superclass, and the interfaces that it implemented. For example,



```

2  objectName.getClass().getName()    // retrieve the class name
    objectName.getClass().newInstance() // create a new instance

```

- The method `toString()` returns a text string description of the object's current state, which is extremely useful for debugging. The `toString()` is implicitly called by `println()` and the string concatenation operator `'+'`. The default implementation in `Object` returns the classname followed by its hash code (in hexadecimal) (e.g., `java.lang.Object@1e78fc6`). This method is meant to be overridden in the subclasses.
- The method `equals()` defines a notion of object equality, based on the object's contents rather than their references. However, the default implementation in `Object` class use `"=="` which compares the object's references. This method is meant to be overridden in the subclasses to compare the content via "deep" comparison, rather than references. The `equals()` shall be reflective and transitive, i.e., `a.equals(b)` is true, `b.equals(a)` shall be true; if `a.equals(b)` and `b.equals(c)` are true, then `a.equals(c)` shall be true.
- The method `hashCode()` maps an object into a hash value. The same object must always produce the same hash value. Two objects which are `equals()` shall also produce the same hash value.
- The method `clone()` is used to make a duplicate of an object. It creates an object of the same type from an existing object, and initializes the new object's member variables to have the same value as the original object. The object to be cloned must implement `Cloneable` interface. Otherwise, a `CloneNotSupportedException` will be thrown. For reference type variable, only the reference is cloned, not the actual object.
- The methods `wait()`, `notify()`, `notifyAll()` are used in concurrent (multithread) programming. The method `finalize()` is run before an object is destroyed (i.e., destructor). It can be used for cleanup operation before the object is garbage-collected.

11.6 java.lang.System

The `System` class contains three static variables `System.in`, `System.out` and `System.err`, corresponding to the standard input, output and error streams, respectively.

The `System` class also contains many useful static methods, such as:

- `System.exit(returnCode)`: terminate the program with the return code.
- `System.currentTimeMillis()` & `System.nanoTime()`: get the current time in milliseconds and nanoseconds. These methods can be used for accurate timing control.

- `System.getProperties()`: retrieving all the system properties.

11.7 `java.lang.Runtime`

Every Java program is associated with an instance of `Runtime`, which can be obtained via the static method `Runtime.getRuntime()`. You can interface with the operating environment via this `Runtime`, e.g., `exec(String command)` launches the command in a separate process.



```
1 // Call up another program
  import java.io.IOException;
3
  public class ExecTest {
5      public static void main(String[] args) {
          try {
7              Runtime.getRuntime().exec("calc.exe");
          } catch (java.io.IOException ex) {
9              ex.printStackTrace();
          }
11     }
  }
```

12 Package `java.util` Frequently-Used Classes

12.1 `java.util.Random`

Although `Math.random()` method can be used to generate a random double between `[0.0, 1.0)`, the `java.util.Random` class provides more extensive operations on random number, e.g., you can set a random number generator with a initial seed value, to generate the same sequence of random values repeatedly.

Example



```
import java.util.Random;
2 public class TestRandomClass {
    public static void main(String[] args) {
4        // Allocate a pseudo-random number generator with default random seed
        Random random = new Random();
6
        // Generate the next 10 pseudo-random uniformly distributed int
        // value between 0 (inclusive) and 100 (exclusive)
8        for (int i = 0; i < 10; ++i) {
10            System.out.print(random.nextInt(100) + " ");
        }
12    System.out.println();
    }
```



```

14      // Generate the next pseudo-random uniformly distributed
15      // double/float value between 0.0 (inclusive) and 1.0 (exclusive)
16      System.out.println(random.nextDouble());
17      System.out.println(random.nextFloat());
18
19      // Allocate a pseudo-random number generator with the specified seed value
20      Random anotherRandom = new Random(12345);
21      // Generate the "same" sequence of 10 integers
22      for (int i = 0; i < 10; ++i) {
23          System.out.print(anotherRandom.nextInt(100) + " ");
24      }
25      System.out.println();
26  }
  
```

Example: Simulating throw of 3 dice.



```

1  /**
2   * Throw 3 dices and get the total score. Also examine for
3   * - 3-of-a-kind (all 3 dice are the same);
4   * - pair (any two dice are the same);
5   * - special (one dice is more than the sum of the other two)
6   */
7  import java.util.Random;
8
9  public class DiceSimulation {
10     public static void main(String[] args) {
11         Random random = new Random(); // Allocate a random generator
12         int[] diceScores = new int[3]; // Allocate 3 dice
13         int totalScore = 0;
14
15         // Throw the dice
16         for (int i = 0; i < diceScores.length; ++i) {
17             diceScores[i] = random.nextInt(6) + 1; // 1 to 6
18         }
19
20         // Compute total score
21         System.out.print("The dice are:");
22         for (int diceScore : diceScores) {
23             totalScore += diceScore;
24             System.out.print(" " + diceScore);
25         }
26         System.out.println();
27         System.out.println("The total score is " + totalScore);
28
29         // Check for 3-of-a-kind and pair
  
```



```
31     if (diceScores[0] == diceScores[1]) {
32         if (diceScores[0] == diceScores[2]) {
33             System.out.println("It's a 3-of-a-kind");
34         } else {
35             System.out.println("It's a pair");
36         }
37     } else {
38         if (diceScores[0] == diceScores[2] ||
39             diceScores[1] == diceScores[2]) {
40             System.out.println("It's a pair");
41         }
42     }
43     // Check for special
44     if ((diceScores[0] > diceScores[1] + diceScores[2]) ||
45         (diceScores[1] > diceScores[0] + diceScores[2]) ||
46         (diceScores[2] > diceScores[0] + diceScores[1])) {
47         System.out.println("It's a special");
48     }
49 }
```

12.2 java.util.Scanner & java.util.Formatter (JDK 1.5)

[TODO]

12.3 java.util.Arrays

The Arrays class contains various static methods for manipulating arrays, such as comparison, sorting and searching.

For examples,

- The static method boolean Arrays.equals(int[] a, int[] b), compare the contents of two int arrays and return boolean true or false.
- The static method void Arrays.sort(int[] a) sorts the given array in ascending numerical order.
- The static method int binarySearch(int[] a, int key) searches the given array for the specified value using the binary search algorithm. others

12.4 java.util.Collection

[TODO]

13 Package java.text Frequently-Used Classes

The java.text package contains classes and interfaces for handling text, dates, numbers and currencies with locale (internationalization) support.

The NumberFormat/DecimalFormat and DateFormat/SimpleDateFormat supports both output formatting (number/date -> string) and input parsing (string -> number/date) in a locale-sensitive manner for internationalization (i18n).

13.1 java.text.NumberFormat

The NumberFormat class can be used to format numbers and currencies for any locale. To format a number for the current Locale, use one of the static factory methods:



```
String myString = NumberFormat.getInstance().format(myNumber);
```

The available factory methods are:



```
1 // Returns a general-purpose number format
  public static final NumberFormat getInstance();
3
  public static final NumberFormat getInstance(Locale l);
5
  // Returns a general-purpose number format
7  public static final NumberFormat getNumberInstance();
  public static final NumberFormat getNumberInstance(Locale l);
9
  // Returns a integer number format
11 public static final NumberFormat getIntegerInstance();
  public static final NumberFormat getIntegerInstance(Locale l);
13
  // Returns a currency number format
15 public static final NumberFormat getCurrencyInstance();
  public static final NumberFormat getCurrencyInstance(Locale l);
17
  // Returns a percent number format
19 public static final NumberFormat getPercentInstance();
  public static final NumberFormat getPercentInstance(Locale l);
```

The default currency format rounds the number to two decimal places; the default percent format rounds to the nearest integral percent; the default integer format rounds to the nearest integer.



```

import java.text.NumberFormat;
2 import java.util.Locale;

4 public class TestNumberCurrencyFormat {
    public static void main(String[] args) {
6         Locale[] locales = { Locale.US, Locale.FRANCE, Locale.JAPAN };

8         for (Locale loc:locales) {
            NumberFormat formatter = NumberFormat.getInstance(loc);
10            String formattedNumber = formatter.format(123456789.12345);
            System.out.format("%15s: %s\n", loc.getDisplayCountry(),
12                formattedNumber);
        }

14        for (Locale loc:locales) {
            NumberFormat formatter = NumberFormat.getCurrencyInstance(loc);
16            String formattedNumber = formatter.format(123456789.12345);
            System.out.format("%15s: %s\n", loc.getDisplayCountry(),
18                formattedNumber);
        }
20    }
22 }

```

Command window

```

United States: 123,456,789.123
2 France: 123 456 789,123
Japan: 123,456,789.123
4 United States: 123,456,789.12
France: 123 456 789,12
6 Japan: 123,456,789

```

Example 2

In this example, we use static method `NumberFormat.getAvailableLocales()` to retrieve all supported locales, and try out `getInstance()`, `getIntegerInstance()`, `getCurrencyInstance()`, `getPercentInstance()`.



```

import java.util.Locale;
2 import java.text.NumberFormat;

4 public class NumberFormatTest {
    public static void main(String[] args) {
6         // Print a number using the localized number, integer, currency,
        // and percent format for each available locale
8         Locale[] locales = NumberFormat.getAvailableLocales();

```



```

10 double myNumber = -1234.56;
   NumberFormat format;

12 // General Number format
   System.out.println("General Format:");
14 for (Locale locale : locales) {
   if (locale.getCountry().length() == 0) {
16     continue; // Skip language-only locales
   }
   format = NumberFormat.getInstance(locale);
   System.out.printf("%40s -> %s%n", locale.getDisplayName(),
20     format.format(myNumber));
   }

22 // Integer format
24 System.out.println("Integer Format:");
   for (Locale locale : locales) {
26     if (locale.getCountry().length() == 0) {
        continue; // Skip language-only locales
28     }
        format = NumberFormat.getIntegerInstance(locale);
30     System.out.printf("%40s -> %s%n", locale.getDisplayName(),
        format.format(myNumber));
32     }

34 // Currency format
   System.out.println("Currency Format:");
36 for (Locale locale : locales) {
   if (locale.getCountry().length() == 0) {
38     continue; // Skip language-only locales
   }
   format = NumberFormat.getCurrencyInstance(locale);
   System.out.printf("%40s -> %s%n", locale.getDisplayName(),
42     format.format(myNumber));
   }

44 // Percent format
46 System.out.println("Percent Format:");
   for (Locale locale : locales) {
48     if (locale.getCountry().length() == 0) {
        continue; // Skip language-only locales
50     }
        format = NumberFormat.getPercentInstance(locale);
52     System.out.printf("%40s -> %s%n", locale.getDisplayName(),
        format.format(myNumber));
54     }
   }
56 }

```

You can also use the NumberFormat to parse an input string (represent in the locale) to a

Number:



```
public Number parse(String source) throws ParseException
```

13.2 java.text.DecimalFormat

The DecimalFormat class is a subclass of NumberFormat, which adds support for formatting floating-point numbers, such as specifying precision, leading and trailing zeros, and prefixes and suffixes. A DecimalFormat object has a pattern to represent the format of the decimal number, e.g., "#,###,##0.00", where 0 denotes zero padding, and # without the zero-padding.

To use a DecimalFormat with the default locale, invoke its constructor with the pattern, e.g.,



```
1 double d = -12345.789
   DecimalFormat format = new DecimalFormat("$#,###,##0.00"); // default
                        ↪ locale
3   System.out.println(format.format(d));           // -$12,345.79

5   format.applyPattern("#,##0.0#;(#,##0.0#)"); // "positive;negative"
   System.out.println(format.format(d));           // (12,345.79)
```

To use a DecimalFormat with locale, get a NumberFormat by calling the getInstance() and downcast it to DecimalFormat. For example,



```
double d = -12345.789;
2   NumberFormat nf = NumberFormat.getInstance(Locale.GERMAN);
   if (nf instanceof DecimalFormat) {
4       DecimalFormat df = (DecimalFormat)nf;
       df.applyPattern("##,##0.00#");
6       System.out.println(df.format(d)); // -12.345,789
   }
```

13.3 java.text.DateFormat

The DateFormat class can be used to format a date instance with locale.

To format a date/time for the current locale, use one of the static factory methods:



```
1 myString = DateFormat.getDateInstance().format(myDate);
```

The available factory methods for getting a `DateFormat` instance are:



```
1 public static final DateFormat getTimeInstance();  
  
3 // DateFormat.FULL, LONG, MEDIUM and SHORT  
public static final DateFormat getTimeInstance(int timeStyle);  
5 public static final DateFormat getTimeInstance(int timeStyle, Locale l)  
  
7 public static final DateFormat getDateInstance();  
public static final DateFormat getDateInstance(int dateStyle);  
9 public static final DateFormat getDateInstance(int dateStyle, Locale l)  
public static final DateFormat getDateTimeInstance();  
11 public static final DateFormat getDateTimeInstance(int dateStyle,  
    int timeStyle);  
13 public static final DateFormat getDateTimeInstance(int dateStyle,  
    int timeStyle, Locale l);  
  
15 // default DateTime formatter in SHORT style  
17 public static final DateFormat getInstance();
```

The exact display for each style depends on the locales, but in general,

- `DateFormat.SHORT` is completely numeric, such as 12.13.52 or 3:30pm
- `DateFormat.MEDIUM` is longer, such as Jan 12, 1952
- `DateFormat.LONG` is longer, such as January 12, 1952 or 3:30:32pm
- `DateFormat.FULL` is pretty completely specified, such as Tuesday, April 12, 1952 AD or 3:30:42pm PST.

You can also use the `DateFormat` to parse an input string containing a date in the locale to a `Date` object.



```
1 public Date parse(String source) throws ParseException
```


13.4 java.text.SimpleDateFormat

The SimpleDateFormat is a concrete subclass of DateFormat for formatting and parsing dates in a locale-sensitive manner. It supports output formatting (date to string), input parsing (string to date), and normalization.

You can construct a SimpleDateFormat via one of its constructors:



```
1 public SimpleDateFormat(String pattern);  
   public SimpleDateFormat(String pattern, Locale locale);
```

14 Writing Javadoc

A great feature in Java is the documentation can be integrated with the source codes, via the so-called Javadoc (Java Documentation) comments. In other languages, documentation typically is written in another file, which easily gets out-of-sync with the source codes.

Javadoc comments begin with `/**` and end with `*/`. They are meant for providing API documentation to the users of the class. Javadoc comments should be provided to describe the class itself; and the public variables, constructors, and methods.

You can use JDK utility javadoc to extract these comments automatically and produce API documentation in a standard format.

With Javadoc comments, you can keep the program documentation inside the same source file instead of using another documentation file. This provides ease in synchronization.

Javadoc comments and API documentation are important for others to re-use your program. Write Javadoc comments while you are writing the program. Do not leave them as after-thought.

Example

Let's add the Javadoc comments to all the public entities of our Circle class.



```
/**  
2  * The Circle class models a circle with a radius and color.  
   *  
4  * @author CHC  
   */  
6  public class Circle {  
  
8      // private instance variables  
      private double radius;
```



```
10     private String color;

12     /**
13      * Construct a circle with default radius of 1.0 and color of blue.
14     */
15     public Circle() {
16         radius = 1.0;
17         color = "blue";
18     }

19     /**
20      * Construct a circle with the given radius and color.
21      * @param radius The radius of the circle
22      * @param color The color of the circle
23     */
24     public Circle(double radius, String color) {
25         this.radius = radius;
26         this.color = color;
27     }

28     /**
29      * Return the radius of the circle.
30      * @return The radius of the circle.
31     */
32     public double getRadius() {
33         return radius;
34     }

35     /**
36      * Set the radius of the circle.
37      * @param radius The radius of the circle to be set.
38     */
39     public void setRadius(double radius) {
40         this.radius = radius;
41     }

42     /**
43      * Return the color of the circle.
44      * @return The color of the circle.
45     */
46     public String getColor() {
47         return color;
48     }

49     /**
50      * Set the color of the circle.
51      * @param color The color of the circle to be set.
52     */
53     public void setColor(String color) {
54         this.color = color;
55     }
56 }
```



```
62  /**
    * Return the area of the circle.
64  * @return The area of the circle.
    */
66  public double getArea() {
    return radius * radius * Math.PI;
68  }

70  /**
    * Return a short description of this instance.
72  * @return A short string description.
    */
74  public String toString() {
    return "Circle[radius=" + radius + ", color=" + color + "];"
76  }
}
```

You can produce the standard API documentation, via JDK utility javadoc, as follows:



```
1  // cd to the source directory
   javadoc Circle.java
```

Browse the resultant API, by opening the "index.html".