Object-Oriented Programming

Java basics I

Tom Lin Jamie Willis

February 1, 2021

Java is an object-oriented, imperative, statically and strongly typed, general- purpose programming language. The Java language itself has multiple implementations, the most prominent one being the *Oracle Java Virtual Machine* (Oracle JVM), other implementations include the *Android Runtime* (ART), *Oracle GraalVM*, and the now-deprecated *GCJ* (GCC once supported Java as a language). The notes below will reference the official *Java Language Specification*, or JLS, which provides strict specifications of the language, should there be any confusion or ambiguity.

Types and references

Java supports two major kinds of types: *primitive type* and *reference type*.

A *primitive type*[1] is not a class and hence cannot have instances. They present a sharp cut from Java's type system and object-oriented philosophy.

[1] §JLS 4.2

A *reference type*[2] could be one of the following:

[2] §JLS 4.3

- Class
- Interface
- Array
- Type variable

Interfaces, arrays, and type variables will be discussed in Java Basics II.

In Java, we make the distinction between a *type* and an *instance* of a type:

Type	Instance

Primitive, e.g `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`	`int a = 42;`
	`double b = 3.14159;`
	`byte c = 0x3b;`

Class, e.g `java.lang.Object` and anything that extends from it, including anonymous classes[a], lambdas[b], and boxed primitives[c].	`Object a = new Object();`
	`Object b = new Object();`
	`Object c = new Object() { };`
	`String d = "foo";`
	`Long e = new Long(42);`
	`Long f = 42L;`
	`Function<Long, Long> g = (x, y) -> x + y;`

[a] A class that is created in-line and has no name
[b] An interface with only one method
[c] `Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, and `Double`

All instances must have a type that is known at compile time. This, however, does not mean the program cannot exhibit dynamic typing behavior, as superclass of all objects `java.lang.Object` can be considered a form of *top type*[1]. Java also supports *casting*[3], which may be used to circumvent type safety.

[3] §JLS 5.5.2

[1] `https://en.wikipedia.org/wiki/Top_type`

Classes

A *class*[4] can contain the following *class members*[5] in no particular order:

```
public class Foo {

   int a = 0; // field(attributes)
   final String s = "a"; // field(attributes)

   static { System.out.println("Class loaded"); } // static initialiser
   { System.out.println("Instance created"); } // instance initialiser

   Foo(int x) { this.a = x; } // constructor
   int b() { return a; } // method
   static Foo c(int y) { return new Foo(y); } // static method
   @Override public String toString() { return "Bar(a=" + a + ")"; } // overridden method

}
```

The class `Foo` can than be instantiated using the `new` keyword and method/fields accessed:

```
Foo foo = new Foo(1);
int a = foo.b(); // a = 1
Foo bar = Foo.c(2);
int b = bar.b(); // b = 2
int c = bar.a; // c = 2
bar.a = 42;
int a1 = bar.a; // a1 = 42;
bar.toString(); // Bar(a=42)
```

Modifiers

There are two major types of modifiers: Access modifiers and non-access modifiers. These are commonly used on class/field/method declarations, but some can be used on parameters and blocks. The main purpose of modifiers is to provide access control or behavior alterations to your code. There is no enforced order of modifiers when multiple distinct modifiers are present, but JSL does specify a *customary* order.

Table 1 shows class modifiers ordered in *ClassModifier* production order[6].

Table 3 shows field modifiers ordered in *FieldModifier* production order[7].

Table 2 shows method modifiers ordered in *MethodModifier* production order[8].

Table 4 shows a list of access modifiers; these can be paired with a class, field, or method, but can only appear once, i.e you cannot mix access modifiers.

Table 5 lists several common modifier combinations.

Modifier	Meaning
`abstract`	Makes itself non-instantiatable and allows for abstract methods
`static`	Decouples dependency from the enclosing class (only meaningful when the class is an inner class)
`final`	Seals the class by prevent subclassing

Table 1: Class modifiers

[4] JLS §8

[5] JLS §8.2

[6] JLS §8.1.1

[7] JLS §8.3.1

[8] JLS §8.4.3

Modifier	Meaning
`abstract`	Allows the method body to be omitted
`static`	Decouples dependency from the enclosing class
`final`	Seals the method to prevent overriding
`synchronized`	Ensures exclusiveness of the invocation when used multi-threaded
`native`	Signifies that there is a backing native implementation of this method

Table 2: Method modifiers

Modifier	Meaning
`static`	Decouples dependency from the enclosing class
`final`	Makes the field non-assignable after initial assignment
`transient`	Makes field transparent to serialisation
`volatile`	Ensures field's data is visible to all threads accessing it

Table 3: Field modifiers

Modifier	same class	same package	subclass	world(other package)
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	✗
no modifier(package private)	✓	✓	✗	✗
`private`	✓	✗	✗	✗

Table 4: Access(visibility) modifiers

Modifier	Meaning
`public static void main(String[] args)/*...*/`	The entry point of a program on the JVM
`public/private static final String CONSTANT = /*...*/;`	A constant declaration
`public/private static double pow(double x, double y) /*...*/`	A synonym to a first class function

Table 5: Commonly used combinations

When to use `static`?

In general, the only use that violates OO principles is static fields. Unless you are defining constants via `static final`, avoid marking fields as `static` where possible.

Control structures

And finally, the usual control structures like in C:

```
boolean value = true;

// conditional
if (true) System.out.println("was true");
else System.out.println("Was false");

System.out.println(value ? "true" : "false"); // ternary
```

```java
// switch, also works with string and enum
switch (1) {
   case 1: System.out.println("num is 1"); break;
   case 2: System.out.println("num is 2"); break;
   default: System.out.println("num is not 1 or 2"); break;
}

int i = 0; while (i < 10) { System.out.println(i); i++; } // while loop
for (int j = 0; j < 10; j++) System.out.println(j); // for
```