The Java™ Tutorials

Trail: Learning the Java Language
Lesson: Classes and Objects
Section: Objects

Sun Microsystems created Java. The language was heavily influenced by Objective-C, which was popularized by NeXT Computer and later Apple Computer. Apple has mostly transitioned to a successor to Objective-C called Swift.

Sun Microsystems no longer exists. Oracle now owns Sun's Java implementation. Google developed its own implementation for use in Android. Oracle and Google have had extended lawsuits about whether Google can have its own implementation without paying royalties. Google won, and most people following the matter breathed a sigh of relief, because if Oracle had won it would have had many other consequences that were anti-competitive and therefore bad for the end user.

The most authoritative documentation for the version of Java we are using is on Oracle's website. The documentation in this PDF is at:

https://docs.oracle.com/javase/tutorial/java/javaOO/usingobject.html

Processing 4.0b2 is using Java 11 (released by Oracle in 2018). Processing 4.0b3 (which we might switch to at some point) just came out on January 15, 2022, and is using Java 17 (released by Oracle in 2021). Generally-speaking, Java 8 documentation is plenty adequate for later Java versions. In particular, this Java tutorial documentation on objects remains completely relevant.

> *The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.*
> *See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.*
> *See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.*

Using Objects

(using the new keyword and a constructor from the desired class)

Once you've created an object, you probably want to use it for something. You may need to use the value of one of its fields, change one of its fields, or call one of its methods to perform an action.

Referencing an Object's Fields

A synonym for "field" in this context is "instance variable." Another synonym is "member." The latter synonym goes back to some of the original implementation of objects as C structures.

Object fields are accessed by their name. You must use a name that is unambiguous.

You may use a simple name for a field within its own class. For example, we can add a statement *within* the `Rectangle` class that prints the `width` and `height`:

```
System.out.println("Width and height are: " + width + ", " + height);
```

In this case, `width` and `height` are simple names.

Code that is outside the object's class must use an object reference or expression, followed by the dot (.) operator, followed by a simple field name, as in:

```
objectReference.fieldName
```

For example, the code in the `CreateObjectDemo` class is outside the code for the `Rectangle` class. So to refer to the `origin`, `width`, and `height` fields within the `Rectangle` object named `rectOne`, the `CreateObjectDemo` class must use the names `rectOne.origin`, `rectOne.width`, and `rectOne.height`, respectively. The program uses two of these names to display the `width` and the `height` of `rectOne`:

```
System.out.println("Width of rectOne: "  + rectOne.width);
System.out.println("Height of rectOne: " + rectOne.height);
```

Attempting to use the simple names `width` and `height` from the code in the `CreateObjectDemo` class doesn't make sense — those fields exist only within an object — and results in a compiler error.

Later, the program uses similar code to display information about `rectTwo`. Objects of the same type have their own copy of the same instance fields. Thus, each `Rectangle` object has fields named `origin`, `width`, and `height`. When you access an instance field through an object reference, you reference that particular object's field. The two objects `rectOne` and `rectTwo` in the `CreateObjectDemo` program have different `origin`, `width`, and `height` fields.

To access a field, you can use a named reference to an object, as in the previous examples, or you can use any expression that returns an object reference. Recall that the `new` operator returns a reference to an object. So you could use the value returned from new to access a new object's fields:

```
int height = new Rectangle().height;
```

This statement creates a new `Rectangle` object and immediately gets its height. In essence, the statement calculates the default height of a `Rectangle`. Note that after this statement has been executed, the program no longer has a reference to the created `Rectangle`, because the program never stored the reference anywhere. The object is unreferenced, and its resources are free to be recycled by the Java Virtual Machine.

Calling an Object's Methods

You also use an object reference to invoke an object's method. You append the method's simple name to the object reference, with an intervening dot operator (.). Also, you provide, within enclosing parentheses, any arguments to the method. If the method does not require any arguments, use empty parentheses.

```
    objectReference.methodName(argumentList);
```

or:

```
    objectReference.methodName();
```

A synonym for "method" in the present context is "member function." Too many synonyms makes for imprecision. I usually say "instance variable" rather than "field," and "method" rather than "member function," and I am not going to be able to get out of that habit, regardless of whatever Shiffman does because I learned object-oriented programming a very long time ago from the Objective-C crowd where "method" and "instance variable" are the standard terms. I haven't told you about "class variables" yet. If there is time and an occasion perhaps we will discuss them.

The `Rectangle` class has two methods: `getArea()` to compute the rectangle's area and `move()` to change the rectangle's origin. Here's the `CreateObjectDemo` code that invokes these two methods:

```
    System.out.println("Area of rectOne: " + rectOne.getArea());
    ...
    rectTwo.move(40, 72);
```

The first statement invokes `rectOne`'s `getArea()` method and displays the results. The second line moves `rectTwo` because the `move()` method assigns new values to the object's `origin.x` and `origin.y`.

As with instance fields, *objectReference* must be a reference to an object. You can use a variable name, but you also can use any expression that returns an object reference. The `new` operator returns an object reference, so you can use the value returned from new to invoke a new object's methods:

```
    new Rectangle(100, 50).getArea()
```

The expression `new Rectangle(100, 50)` returns an object reference that refers to a `Rectangle` object. As shown, you can use the dot notation to invoke the new `Rectangle`'s `getArea()` method to compute the area of the new rectangle.

Some methods, such as `getArea()`, return a value. For methods that return a value, you can use the method invocation in expressions. You can assign the return value to a variable, use it to make decisions, or control a loop. This code assigns the value returned by `getArea()` to the variable `areaOfRectangle`:

```
    int areaOfRectangle = new Rectangle(100, 50).getArea();
```

Remember, invoking a method on a particular object is the same as sending a message to that object. In this case, the object that `getArea()` is invoked on is the rectangle returned by the constructor.

*Garbage collection is desirable because objects are passed by reference (recall our pass-by-reference vs. pass-by-value discussion). ***The upshot of the fact that the Java runtime has a garbage collector is that you don't have to worry about de-allocating objects.*** This puts them on par with the primitive types (which you declare, and which are usable in scope, but which you don't have to destroy). The "Post-Its" on which primitive types are written are destroyed for you when the code block in which they were declared goes out of scope (otherwise, the Post-Its would just accumulate).*

The Garbage Collector

Some object-oriented languages require that you keep track of all the objects you create and that you explicitly destroy them when they are no longer needed. Managing memory explicitly is tedious and error-prone. The Java platform allows you to create as many objects as you want (limited, of course, by what your system can handle), and you don't have to worry about destroying them. The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called *garbage collection*.

An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are usually dropped when the variable goes out of scope. Or, you can explicitly drop an object reference by setting the variable to the special value `null`. Remember that a program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection.

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically when it determines that the time is right.