
Brief Summary of Java

Java programs are compiled into an intermediate format, known as bytecode, and then run through an interpreter that executes in a Java Virtual Machine (JVM).

The basic syntax of Java is similar to that of C and C++. All white space is treated equally, indent level does not matter, statements end in a semi-colon, and blocks of code are enclosed between { and }.

Comments are enclosed between /* and */, or else begin with //, in which case the rest of the line is a comment.

The integer data types are `byte`, `short`, `int`, and `long`, which correspond to numbers of 8, 16, 32, and 64 bits. The types `float` and `double` store floating-point numbers; `char` stores a 16-bit Unicode character, and `boolean` can hold one of two values, `true` or `false`.

Variables are declared with a type and name, as in:

```
int myint;
```

and can be initialized at the same time:

```
char delimiter = '/';  
boolean finished = false;
```

Variables can be declared anywhere they are used; the scope of a variable usually extends to the end of the code block it was declared in.

Java allows variables to be converted between different numeric types by casting, as in:

```
int a;  
double d = (double)a;
```

You can also cast between objects, but we won't cover that.

Variables can be declared as `final`, which means their value cannot be changed once it is initialized:

```
final int MAX_LEN = 128;
```

Arithmetic expressions in Java are straightforward, with `%` used for modulo:

```
k = a + b;  
remainder = tot % users;
```

The ++ and -- operators exist; if they are used in prefix notation, the expression is evaluated after the operation is done; in postfix notation, the expression is evaluated before the operation is done. So, with the following

```
d = 4;  
e = ++d;  
f = e--;
```

e and f will both get set to 5.

Beyond the basic data types, everything in Java is stored in objects. An object is a grouping of variables and methods (functions that operate on those variables).

The variables and methods included in an object are defined in a class. Technically a class is a description of an object and an instance is an actual object, but the word object is used to refer to both.

You can define your own classes; Java includes many pre-defined ones. One such class is `String`, which is used to store a constant string. Strings in Java are not just arrays of characters; they are a class that has defined methods for accessing and modifying the characters in the string.

We'll use the `String` class to show how Java objects are used. A `String` can be created from an array of characters as follows:

```
char[] myArray = { 'a', 'b', 'c' };  
String myString = new String(myArray);
```

The expression `new String(myArray)` invokes what is called a constructor for the class `String`. Constructors create a new object, optionally taking parameters. How many parameters a constructor takes, and what types those parameters are, is part of the class definition. Multiple constructors can exist, as long as they take different parameter lists. For example, another constructor for `String` is called as

```
String myString = new String(myArray, 2, 1);
```

That is, specifying an offset and count within `myArray`, and you can also call:

```
String myString = new String();
```

which creates an empty string (A `String` cannot be changed once initialized, so it would stay empty).

When Java sees a string in double quotes, it automatically converts it to a `String` object, so you can write:

```
String newString = "text";
```

which is actually an assignment of one `String` to another. This automatic creation of an object from a constant is unique to the `String` class, but it sure is convenient.

There is also a constructor for `String` that takes another `String`:

```
String newString = new String("text");
```

although this would temporarily create an extra object (since "text" automatically becomes a `String`, then is used to construct another one), so it is unnecessary. `String` actually has nine constructors, plus two more that are obsolete. There are no destructors in Java; objects are destroyed after the last reference to them is removed (often because the variables holding that reference goes out of scope). A variable can be assigned a keyword `null` to force its reference to be removed:

```
anotherString = null;
```

Java does not have explicit pointers; in a sense, all variables that refer to objects are pointers. When you assign between two objects of the same type, you are actually assigning a reference to the object on the right-hand side; to create a new object, you need to construct a new one:

```
myObject a, b;  
a = b;           // reference  
a = new myObject(b); // create a new object
```

Classes define methods that can be called on an instance of that class. For example, the `String` class has a method `length()` which returns the length of the string:

```
String j = "abc123";  
x = j.length();
```

As mentioned earlier, a `String` cannot change once initialized. Java has another class, `StringBuffer`, which holds strings that can change. A `StringBuffer` can be constructed from a `String`, or from a length, which specifies how many characters of capacity it should start with:

```
StringBuffer sb1 = new StringBuffer("howdy");  
StringBuffer sb2 = new StringBuffer(100);
```

StringBuffer has a variety of methods on it:

```
sb.append("more data");  
char c = sb.charAt(12);  
sb.reverse();
```

In Java, the + operator can be used to concatenate strings together; a sequence such as

```
String greeting = "Hello";  
greeting = greeting + " there";
```

is legal. Since the original `String` that `greeting` points to cannot be modified, the concatenation actually involves the creation of a new `String`, which `greeting` is then set to point to, thus removing the reference to the original "Hello" string, which will eventually cause it to be destroyed.

(The concatenation statement also involves some more behind-the-scenes magic by the compiler; it creates a temporary `StringBuffer`, then calls the `StringBuffer.append()` method for each expression separated by a + sign, then calls `StringBuffer.toString()` to convert it back to the result `String`. As with the automatic creation of `String` objects from constant strings, this is a special case on the part of Java, but is there because string concatenation is so useful.)

`StringBuffer.append()` is overloaded so it can be passed any primitive type, thus you can call

```
int j = 4;  
String b = "Value is " + j;
```

and `b` will equal "Value is 4"; in fact `StringBuffer.append()` will work for any object at all by appending the result of the object's `toString()` method, which can be overridden as needed by the author of the object's class.

Java likely has a class for almost any standard operation you want to do; the documentation lists constructors and methods. For example, there are classes that wrap all of the primitive types, such as this one that wraps the short primitive in a class called `Short` (note the capital 'S' on the class name), and provides various useful methods:

```
Short s = new Short(12);  
String str = s.toString();
```

I won't go into more detail about specific classes, except as needed in the examples. In order to use a class, you have to import the package that defines it; this will be specified in the documentation of the class. For example, to use the String class, include the following in the code:

```
import java.lang.String;
```

which can include a wildcard:

```
import java.lang.*;
```

Arrays in Java are declared with square brackets:

```
int[] intArray;
```

The array then has to be created:

```
intArray = new int[10];
```

intArray would then be indexed from 0 to 9.

Arrays can also be created at declaration time, if values are specified:

```
int[] array2 = { 5, 4, 3, 2, 1 };
```

You can't explicitly specify the length in that case; it's determined from how many values are provided.

You can get the number of elements in an array:

```
k = array2.length;
```

Note that this is not a method, so there are no parentheses after length.

Arrays can also hold objects, so you can declare:

```
MyObject[] objarray;
```

which would then be created as follows (this could be combined with the declaration):

```
objarray = new MyObject[5];
```

It is important to note that this creates only the array. You still need to create the 5 objects:

```
for (k = 0; k < 5; k++) {  
    objarray[k] = new MyObject();  
}
```

To create sub-arrays, create an array each of whose elements is an array. The first array can be declared and created in one step:

```
int[][] bigArray = new int[6][];
```

and then each of the sub-arrays would need to be created (they can each be different lengths, in fact):

```
for (m = 0; m < 6; m++) {  
    bigArray[m] = new int[20];  
}
```

You can initialize arrays when they are declared:

```
short[][] shortArray = { { 1, 2, 3 }, { 4 }, { 5, 6 } };
```

After that, `shortArray[0]` would be an array of 3 elements, `shortArray[1]` would be an array of 1 element, and `shortArray[2]` would be an array of 2 elements.

Finally, if the entries in the arrays are objects, they have to be constructed as well, as in the following:

```
final int XDIM = 6;  
final int YDIM = 10;  
SomeObj[][] oa;  
oa = new SomeObj[XDIM][];  
for (int i = 0; i < XDIM; i++) {  
    oa[i] = new SomeObj[YDIM];  
    for (int j = 0; j < YDIM; j++) {  
        oa[i][j] = new SomeObj();  
    }  
}
```

Java conditionals use the same `if/else` syntax as C:

```
if (j == 5) {  
    // do something
```

```
} else {  
    //do something else  
}
```

The switch statement is also the same, with explicit break statements required, and a default case:

```
switch (newChar) {  
    case "@":  
        process_at();  
        break;  
    case ".":  
        process_dot();  
        break;  
    default:  
        ignore();  
}
```

Looping is done with for, while, and do/while:

```
while (k > 8) {  
    do_processing();  
}  
  
do {  
    eof = get_line();  
} while (eof != true);
```

break breaks out of a loop, and continue jumps to the next iteration. A label can be added to break or continue to specify which loop it refers to:

```
outerloop:  
for (x = 0; x < 20; x++) {  
    for (y = x; y < 20; y++) {  
        if (something) {  
            break outerloop;  
        }  
    }  
}
```

Note that outerloop: is a label for the loop and the statement break outerloop; breaks out of the labeled loop. It does *not* jump to the point where the outerloop: label exists in the code.

Any Java program requires that the author define at least one class. A class is defined as follows:

```
class MyClass {
    private int a;
    public StringBuffer b;
    public MyClass(int j) {
        a = j;
        b = new StringBuffer(j);
    }
    public MyClass(String s) {
        a = s.length();
        b = new StringBuffer(s);
    }
    public int getLength() {
        return a;
    }
}
```

a and b are member variables in the class. a is defined with an access specifier of `private`, meaning that it is hidden from the view of external code; b is `public`, meaning anyone can access it if they have an instance of `MyClass`. For example:

```
MyClass mc = new MyClass("hello");
String abc = mc.b;    // this is allowed, b is public
int def = mc.a;       // this is NOT allowed, a is private
```

We'll get back to access specifiers in a second. For the moment, note that `MyClass` has two constructors, one of which takes an `int` as a parameter, one of which takes a `String` (the second one is the one we called in the code sample above). Both constructors initialize a and b; variables can also be initialized when they are declared, so b could have been declared as

```
public StringBuffer b = new StringBuffer();
```

although for this class that would not be necessary since every constructor initializes it.

Classes can also inherit from another class. A subclass inherits all the state and behavior of the its superclass (but *not* the constructors!), although it can override variables and methods by providing new ones with the same name (unless those variables and methods were declared with the `final` keyword). If the superclass is declared `abstract`, it cannot itself be instantiated and exists to be a common ancestor to different subclasses. Inheritance is indicated by the `extends` keyword:

```
abstract class Polygon {
    Point[] points;
    abstract int getcount();
}
```

```
}  
  
class Triangle extends Polygon {  
    public Triangle() {  
        points = new Point[3];  
    }  
    int getcount() { return 3 };  
}
```

The access specifier of a class variable can be `public`, `private`, `protected`, or `package` (the default). `public` means any code can access it; `private` means only methods in the class itself can access it, and `package` means any code in the same "package" (which is a way to group classes together) can access it.

A variable marked `protected` can be accessed by the class, subclasses, and all classes in the same package. Actually, to be more precise, subclasses can only access a `protected` member inherited from a superclass when the object is an instance of the subclass (which it usually will be); they can't modify an instance of the superclass itself. If you didn't catch that, don't worry too much about it.

Members of a class (variables or methods) can be declared with the keyword `static`, which makes them "class members," as opposed to "instance members" which is the case we have been describing so far. Class variables and class methods exist just once, as opposed to once per instance. For example, a class could assign unique identifiers to each instance it creates:

```
class ImportantObject {  
    private static int nextcounter = 0;  
    private int counter;  
    public ImportantObject() {  
        counter = nextcounter++;  
    }  
    // continues...  
}
```

Each instance of the class has its own `counter` member, but there is only one global `nextcounter`.

Closely related to classes are interfaces. An interface is like an abstract class, except it can't implement any of its methods:

```
public interface identify {  
    String getName();  
}
```

Other classes can now support an interface using the `implements` keyword. Unlike inheritance, where a class can only inherit from one class, classes can implement as many interfaces as they like, as long as they provide implementations of the methods:

```
class SomeClass implements identify {
    final String name = "SomeClass";
    String getName() { return name };
    // rest of class follows...
}
```

A class with only public member variables, and no methods, can be used to group variables together by name, similar to C structures:

```
class Record {
    public String name;
    public int id;
    public int privilege;
}

Record r = new Record();
r.name = "Joe";
r.id = 12;
r.privilege = 3;
```

Java supports exceptions. Exceptions are objects, which can be caught:

```
try {
    j = my_array[x];
} catch (ArrayIndexOutOfBoundsException e) {
    System.err.println("Exception " + e.getMessage());
} finally {
    // cleanup code
}
```

A try can have multiple catch blocks, each catching a different exception (there is a hierarchy of exception classes, leading back to a class called `Throwable`; a catch block that catches a particular exception will also catch any exceptions that are subclasses of that exception).

If an exception happens and is caught, the catch block executes. The finally block always executes, whether an exception happens or not, and is usually used for cleanup code.

You can create and throw exceptions:

```
if (array.length == 0) {
    throw new IllegalArgumentException();
}
```

Java requires that functions that can throw an exception specify it in the declaration of the function with `throws`:

```
public void process_array(int[] array)
    throws IllegalArgumentException {
    if (array.length == 0) {
        throw new IllegalArgumentException();
    }
}
```

Functions must also specify exceptions that may be thrown by functions they call, unless they catch the exception. Thus, a function that called `process_array()` as defined above would need to either put it in a try block with an associated catch block that caught `IllegalArgumentException`, or specify in its own declaration that it throws `IllegalArgumentException` (this "catch or specify" rule does not apply to a class of exceptions known as runtime exceptions; this is discussed in detail in the Java documentation).

The examples we use will be split between command-line applications and applets designed to run in a web browser. A command-line application has to contain a class that implements a `main()` function, which must be defined as `public static`, return type `void`, and receive the command-line as an array of `Strings`:

```
public class MyApplication {
    public static void main(String[] args) {
        for (int j = 0; j < args.length; j++) {
            System.out.println(args[j]);
        }
    }
}
```

An applet inherits from a class called `Applet`:

```
public class MyApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Testing 123", 10, 10);
    }
}
```

The `paint()` method is overridden (from a superclass a few levels up from `Applet`) and is used to display on the screen; the `Graphics` class has many of the methods used to draw lines and shapes, display text, change color, etc.