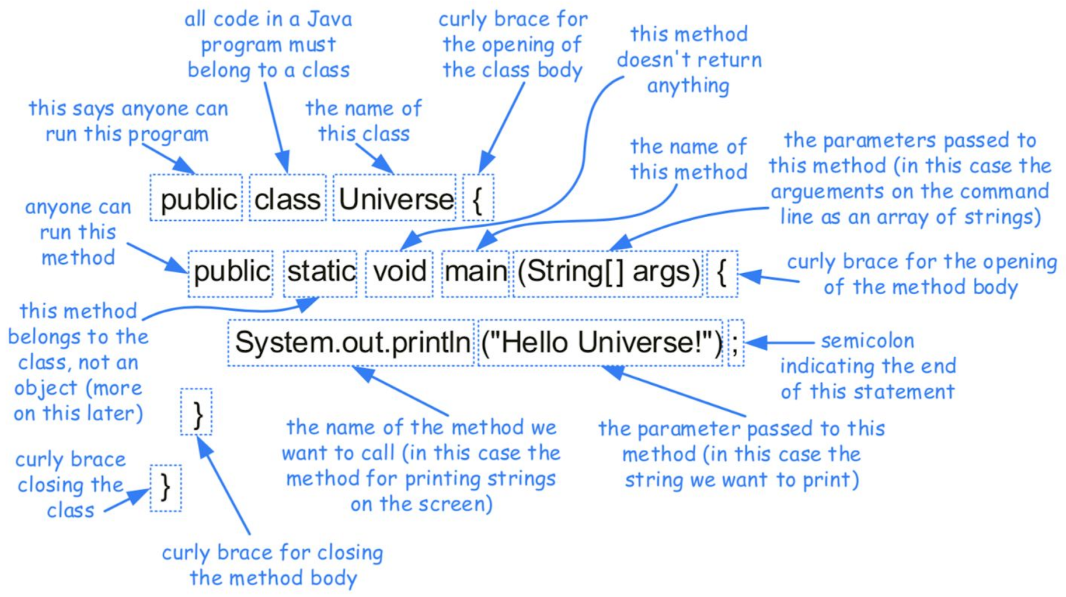
JAVA PRIMER



all code in a Java program must belong to a class
curly brace for the opening of the class body
this method doesn't return anything
this says anyone can run this program
the name of this class
the name of this method
the parameters passed to this method (in this case the arguements on the command line as an array of strings)
public class Universe {
anyone can run this method
public static void main (String[] args) {
curly brace for the opening of the method body
this method belongs to the class, not an object (more on this later)
System.out.println ("Hello Universe!") ;
semicolon indicating the end of this statement
}
curly brace closing the class
}
the name of the method we want to call (in this case the method for printing strings on the screen)
the parameter passed to this method (in this case the string we want to print)
curly brace for closing the method body

Figure 1.1: A “Hello Universe!” program.

Methods -executable statements are placed in functions, known as methods
Methods generally belong to a **class.**
Any set of statements between the braces “{” and “}” define a program ***block***.
The name of a class, method, or variable in Java is called an ***identifier***.
Objects- In java, objects are primary actors.
Block comments that begin with “/**” (note the second asterisk) have a special purpose, allowing a program, called Javadoc, to read these comments and automatically generate software documentation.

Every object is an ***instance*** of a class, which serves as the ***type*** of the object and as a blueprint, defining the data which the object stores and the methods for accessing and modifying that data. The critical ***members*** of a class in Java are the following:

• ***Instance variables***, which are also called ***fields***, represent the data associated with an object of a class. Instance variables must have a ***type***, which can either be a base type (such as int, float, or double) or any class type (also known as a ***reference type*** for reasons we soon explain).

• ***Methods*** in Java are blocks of code that can be called to perform actions (similar to functions and procedures in other high-level languages). Methods can accept parameters as arguments, and their behavior may depend on the object upon which they are invoked and the values of any parameters that are passed. A method that returns information to the caller without changing any instance variables is known as an ***accessor method***, while an ***update method*** is one that may change one or more instance variables when called.

```
1  public class Counter {
2    private int count;                                    // a simple integer instance variable
3    public Counter( ) { }                                 // default constructor (count is 0)
4    public Counter(int initial) { count = initial; }          // an alternate constructor
5    public int getCount( ) { return count; }                     // an accessor method
6    public void increment( ) { count++; }                         // an update method
7    public void increment(int delta) { count += delta; }          // an update method
8    public void reset( ) { count = 0; }                           // an update method
9  }
```

Code Fragment 1.2: A Counter class for a simple counter, which can be queried, incremented, and reset.

This class includes one instance variable, named count, which is declared at line 2. As noted on the previous page, the count will have a default value of zero, unless we otherwise initialize it.

The class includes two special methods known as constructors (lines 3 and 4), one accessor method (line 5), and three update methods (lines 6–8). Unlike the original Universe class from page 2, our Counter class does not have a main method, and so it cannot be run as a complete program. Instead, the purpose of the Counter class is to create instances that might be used as part of a larger program.

<u>Creating and Using Objects</u>

```
public class CounterDemo {
  public static void main(String[ ] args) {
    Counter c;                    // declares a variable; no counter yet constructed
    c = new Counter( );           // constructs a counter; assigns its reference to c
    c.increment( );               // increases its value by one
    c.increment(3);               // increases its value by three more
    int temp = c.getCount( );     // will be 4
    c.reset( );                   // value becomes 0
    Counter d = new Counter(5);// declares and constructs a counter having value 5
    d.increment( );               // value becomes 6
    Counter e = d;                // assigns e to reference the same object as d
    temp = e.getCount( );         // will be 6 (as e and d reference the same counter)
    e.increment(2);               // value of e (also known as d) becomes 8
  }
}
```

Code Fragment 1.3: A demonstration of the use of Counter instances.

At line 3 of our demonstration, a new variable c is declared with the syntax:

Counter c;

This establishes the identifier, c, as a variable of type Counter, but it does not create a Counter instance.
Classes are known as ***reference types*** in Java, and a variable of that type (such as c in our example) is known as a ***reference variable***. A reference variable is capable of storing the location (i.e., ***memory address***) of an object from the declared class.

Creation of a new object in Java
In Java, a new object is created by using the new operator followed by a call to a constructor for the desired class; a constructor is a method that always shares the same name as its class. The new operator returns a ***reference*** to the newly created instance; the returned reference is typically assigned to a variable for further use.
In Code Fragment 1.3, a new Counter is constructed at line 4, with its reference assigned to the variable c. That relies on a form of the constructor, Counter(), that takes no arguments between the parentheses. (Such a zero-parameter constructor is known as a ***default constructor***.)

Three events occur as part of the creation of a new instance of a class:

- A new object is dynamically allocated in memory, and all instance variables are initialized to standard default values. The default values are null for reference variables and 0 for all base types except boolean variables (which are false by default).
- The constructor for the new object is called with the parameters specified.
  The constructor may assign more meaningful values to any of the instance variables, and perform any additional computations that must be done due to the creation of this object.
- After the constructor returns, the new operator returns a reference (that is, a memory address) to the newly created object. If the expression is in the form of an assignment statement, then this address is stored in the object variable, so the object variable ***refers*** to this newly created object.

The Dot Operator

One of the primary uses of an object reference variable is to access the members of the class for this object, an instance of its class. That is, an object reference vari- able is useful for accessing the methods and instance variables associated with an object. This access is performed with the dot (".") operator.
For example, in Code Frag- ment 1.3, we call c.increment() at line 5, c.increment(3) at line 6, c.getCount() at line 7, and c.reset() at line 8. If the dot operator is used on a reference that is currently null, the Java runtime environment will throw a NullPointerException.

Signatures
If there are several methods with this same name defined for a class, then the Java runtime system uses the one that matches the actual number of parameters sent as arguments, as well as their respective types. For example, our Counter class supports two methods named increment: a zero-parameter form and a one- parameter form. Java determines which version to call when evaluating commands such as c.increment( ) versus c.increment(3). A method's name combined with the number and types of its parameters is called a method's ***signature***, for it takes all of these parts to determine the actual method to perform for a certain method call. Note, however, that the signature of a method in Java does not include the type that the method returns, so Java does not allow two methods with the same signature to return different types.

Referencing Objects

There can, in fact, be many references to the same object, and each reference to a specific object can be used to call methods on that object. Such a situation would correspond to our having many remote controls that all work on the same device. Any of the remotes can be used to make a change to the device (like changing a channel on a television). Note that if one remote control is used to change the device, then the (single) object pointed to by all the remotes changes. Likewise, if one object reference variable is used to change the state of the object, then its state changes for all the references to it. This behavior comes from the fact that there are many references, but they all point to the same object.

Returning to our CounterDemo example, the instance constructed at line 9 as Counter d = new Counter(5); is a distinct instance from the one identified as c. However, the command at line 11, Counter e = d;

does not result in the construction of a new Counter instance. This declares a new *reference variable* named e, and assigns that variable a reference to the existing counter instance currently identified as d. At that point, both variables d and e are aliases for the same object, and so the call to d.getCount() behaves just as would e.getCount(). Similarly, the call to update method e.increment(2) is affecting the same object identified by d.

It is worth noting, however, that the aliasing of two reference variables to the same object is not permanent. At any point in time, we may reassign a reference variable to a new instance, to a different existing instance, or to null.

Modifiers

Immediately before the definition of a class, instance variable, or method in Java, keywords known as ***modifiers*** can be placed to convey additional stipulations about that definition.

Access Control Modifiers

The first set of modifiers we discuss are known as ***access control modifiers***, as they control the level of access (also known as ***visibility***) that the defining class grants to other classes in the context of a larger Java program.

The static Modifier

The static modifier in Java can be declared for any variable or method of a class
(or for a nested class, as we will introduce in Section 2.6).

When a variable of a class is declared as static, its value is associated with the class as a whole, rather than with each individual instance of that class. Static variables are used to store “global” information about a class. (For example, a static variable could be used to maintain the total number of instances of that class that have been created.) Static variables exist even if no instance of their class exists.
When a method of a class is declared as static, it too is associated with the class itself, and not with a particular instance of the class. That means that the method is not invoked on a particular instance of the class using the traditional dot notation. Instead, it is typically invoked using the name of the class as a qualifier.

As an example, in the java.lang package, which is part of the standard Java distribution, there is a Math class that provides many static methods, including one named sqrt that computes square roots of numbers. To compute a square root, you do not need to create an instance of the Math class; that method is called using a syntax such as Math.sqrt(2), with the class name Math as the qualifier before the dot operator.
Static methods can be useful for providing utility behaviors related to a class that need not rely on the state of any particular instance of that class.

The final Modifier

A variable that is declared with the final modifier can be initialized as part of that declaration, but can never again be assigned a new value. If it is a base type, then it is a constant. If a reference variable is final, then it will always refer to the same object (even if that object changes its internal state). If a member variable of a class is declared as final, it will typically be declared as static as well, because it would be unnecessarily wasteful to have every instance store the identical value when that value can be shared by the entire class.

Designating a method or an entire class as final has a completely different consequence, only relevant in the context of inheritance. A final method cannot be overridden by a subclass, and a final class cannot even be subclassed.

Declaring Instance Variables

When defining a class, we can declare any number of instance variables. An impor- tant principle of object-orientation is that each instance of a class maintains its own individual set of instance variables (that is, in fact, why they are called *instance* variables). So in the case of the Counter class, each instance will store its own (independent) value of count.

The general syntax for declaring one or more instance variables of a class is as follows (with optional portions bracketed):
[*modifiers*] *type identifier*$_1$[=*initialValue*$_1$], *identifier*$_2$[=*initialValue*$_2$];
In the case of the Counter class, we declared private int count;
where private is the modifier, int is the type, and count is the identifier. Because we did not declare an initial value, it automatically receives the default of zero as a base-type integer.

Java methods can return only one value. To return multiple values in Java, we should instead combine all the values we want to return in a ***compound object****, whose instance variables include all the values we want to return, and then return a reference to that compound object. In addition, we can change the internal state of an object that is passed to a method as another way of "returning" multiple results.*

Parameters
A method's parameters are defined in a comma-separated list enclosed in parentheses after the name of the method. A parameter consists of two parts, the parameter type, and the parameter name. If a method has no parameters, then only an empty pair of parentheses is used.

All parameters in Java are passed ***by value***, that is, any time we pass a parameter to a method, a copy of that parameter is made for use within the method body. So if we pass an int variable to a method, then that variable's integer value is copied. The method can change the copy but not the original. If we pass an object reference as a parameter to a method, then the reference is copied as well. Remember that we can have many different variables that all refer to the same object. Reassigning the internal reference variable inside a method will not change the reference that was passed in.

Defining Constructors

A ***constructor*** is a special kind of method that is used to initialize a newly created instance of the class so that it will be in a consistent and stable initial state. This is typically achieved by initializing each instance variable of the object (unless the default value will suffice), although a constructor can perform more complex computation. The general syntax for declaring a constructor in Java is as follows:

modifiers name(*type*$_0$ *parameter*$_0$, ..., *type*$_{n-1}$ *parameter*$_{n-1}$) { // constructor body . . .

}

Constructors are defined in a very similar way as other methods of a class, but there are a few important distinctions:

1. Constructors cannot be static, abstract, or final, so the only modifiers that are allowed are those that affect visibility (i.e., public, protected, private, or the default package-level visibility).
2. The name of the constructor must be identical to the name of the class it constructs. For example, when defining the Counter class, a constructor must be named Counter as well.

3. We don’t specify a return type for a constructor (not even void). Nor does the body of a constructor explicitly return anything. When a user of a class creates an instance using a syntax such as

   ```
   Counter d = new Counter(5);
   ```

   the new operator is responsible for returning a reference to the new instance to the caller; the responsibility of the constructor method is only to initialize the state of the new instance.

A class can have many constructors, but each must have a different ***signature***, that is, each must be distinguished by the type and number of the parameters it takes. If no constructors are explicitly defined, Java provides an implicit ***default constructor*** for the class, having zero arguments and leaving all instance variables initialized to their default values. However, if a class defines one or more nondefault constructors, no default constructor will be provided.
As an example, our Counter class defines the following pair of constructors:

```
public Counter() { }
public Counter(int initial) { count = initial; }
```

The first of these has a trivial body, { }, as the goal for this default constructor is to create a counter with value zero, and that is already the default value of the integer instance variable, count. However, it is still important that we declared such an explicit constructor, because otherwise none would have been provided, given the existence of the nondefault constructor. In that scenario, a user would have been unable to use the syntax, new Counter().

Out of Bounds Errors

It is a dangerous mistake to attempt to index into an array *a* using a number outside the range from 0 to *a*.length−1. Such a reference is said to be ***out of bounds***. Out of bounds references have been exploited numerous times by hackers using a method called the ***buffer overflow attack*** to compromise the security of computer systems written in languages other than Java. As a safety feature, array indices are always checked in Java to see if they are ever out of bounds. If an array index is out of bounds, the runtime Java environment signals an error condition. The name of this condition is the ArrayIndexOutOfBoundsException. This check helps Java avoid a number of security problems, such as buffer overflow attacks.