

Chapter 2

Introduction to Java

©2002 by Harvey Gould, Jan Tobochnik, and Wolfgang Christian
11 February 2002

We introduce some of the basic structure and syntax of Java including classes and methods, primitive data types, loops, operator overloading, and method overriding. We also investigate the limits that finite precision arithmetic places on computation.

2.1 Introduction

Java, like any language, has a grammar or syntax that borrows from other languages including mathematics. If you are familiar with C or C++, you will find that the expressions, operators, and statements in Java are very similar. In this chapter we will introduce some of the syntax that we will need to write grammatically correct Java statements. As will be the norm, we first give a sample program and then explain the new syntax.

The following program is an example of a Java program that sums the series $\sum_{n=1}^{100} 1/n^2$:

```
public class MyFirstApp { // beginning of class definition

    public static void main(String[] args) {
        // statements within the braces { } form the body of the method
        double sum = 0; // example of declaration and assignment statement
        for (int n = 1; n <= 100; n = n + 1) { // beginning of for loop
            double term = 1.0/(n*n);
            sum = sum + term;
        } // end of for loop
        System.out.println("sum = " + sum); // print result
    } // end of method definition
} // end of class definition
```

As we will see, this program is not really object oriented, but it allows us to introduce much of the syntax of Java.

The first line declares a new class named `MyFirstApp`. The Java convention is to begin class names with an uppercase letter. If a name consists of more than one word, the words are joined together and each succeeding word begins with a uppercase letter (another Java convention). Each class must be in a separate file with the same name as the name of the class, for example, `MyFirstApp.java`. The left brace begins the body of the class definition and the corresponding right brace ends the class definition.

The second statement defines the `main` method. This method has a special status in Java. A Java application (a stand-alone program) consists of one or more class definitions, one of which must define the method `main`. (A Java applet is different insofar as that it runs inside a browser and does not usually contain a `main` method; instead it has other methods such as `init` and `start`.) The `main` method is automatically called first to start the program. We call the class that contains this method the *target* class. The syntax associated with the declaration of the `main` method is convoluted, but understanding this syntax is not really necessary. Note that methods are defined only inside a class definition. Because we will later write programs consisting of many classes, we will adopt our own convention that the filename of the target class ends with `App`. Familiarize yourself with your Java development environment by doing Exercise 2.1.

Exercise 2.1. First Application

- a. Enter the above listing of `MyFirstApp` into a file named `MyFirstApp.java`. Be sure to pay attention to capitalization because Java is case sensitive.
- b. Compile and run `MyFirstApp`. How do you know if the program is working correctly?
- c. What happens if you make a typing mistake? For example, try typing `double sun 0`; and see what happens.

Digital computers can store information only as binary numbers, that is, sequences of ones and zeros. Every one or zero is called a *bit* and a group of 8 bits is referred to as a *Byte*. For example, the integer 362 is stored as 0000000101101011 using two Bytes of memory. It would be difficult to write a program if we had to write numbers such as the speed of light in binary format. High-level computer languages allow us to reference stored numbers using *identifiers* or variable names. A valid identifier is a series of characters consisting of letters, digits, underscores, and dollar signs that does not begin with a digit nor contain any spaces. Because Java distinguishes between upper and lowercase characters, `Sum` and `sum` are different identifiers. The Java convention is that variable names begin with a lowercase letter, except in special cases, and each succeeding word in a variable name begins with an uppercase letter.

A variable can refer to just about anything including objects such as graphs. In a purely object oriented language, all variables would be objects that we would introduce by their class definitions. However, there are certain constructs that are so basic and important that they have a special status and are especially easy to create and access. These variables are called *primitive data types* and represent integer, real, boolean, and character variables. An integer variable, a double variable, and a boolean variable are created and initialized by the following statements:

```
int numberOfParticles = 10;
double sum = 0;
boolean inert = false;
```

The primitive type `char` is used for single characters. A variable must be *declared* before it can be used, and the value of a variable can be initialized at the same time that its type is declared.

There are four types of integers, `byte`, `short`, `int`, and `long`, with the differences being the range that these types can store (see Table 2.3 in Appendix 2A). We will almost always use type `int` because it does not require as much memory as type `long`. There also are two types of floating point numbers, but we will always use type `double` to avoid excessive roundoff error.

Certain combinations of characters are part of the Java language and cannot be used as variable names. These reserved words are known as *keywords*. They are written in lowercase and include the following:

```
false double new public super for main public extends
```

A complete list is given in Appendix xx.

A statement causes Java to perform an action. It contains expressions and is terminated by a semicolon. A statement is like a complete sentence and an expression is similar to a phrase. The simplest expressions are identifiers or variables. More interesting expressions can be created by combining variables using *operators*, such as the following example of the plus operator:

```
x + 3.0
```

An expression can be evaluated to produce a value. The value of the above expression is determined by the current value of the variable `x` and the usual rules of mathematics.

The simplest statement is the *assignment statement*, for example

```
x = 1;
c = 2.99792458e8;
```

The *assignment operator*, the equals sign, stores the value of the expression in the memory location that is associated with a variable, such as `c` in the above statement. Note how powers of 10 are written.

The following example illustrates an important difference between the equals sign in mathematics and in Java.

```
int x = 3;
x = x + 1;
```

What is the value of x ? The equals sign produces a replacement of a value in memory and is not a statement of equality. In fact, the left hand and right hand sides of an equals sign are usually not equal.

Java supports a variety of operators. Table 2.1 shows the operators that we will use most often. A complete listing is given in Appendix xx.

One way to sum a series of terms is to use a *loop* construction. Loops are blocks of code that are executed repeatedly. They typically require the initialization of one or more variables, a test to determine if the loop should continue or stop, and a counter variable that changes as the loop executes in order to bring about the fulfillment of the stopping condition. The `for` statement makes these three steps explicit. The loop in `MyFirstApp` illustrates how a `for` loop can be used to sum the first 100 terms in a series. What does the following example do?

operator	operand	description	sample expression	result
=	any	assignment	$x = 7, y = 3;$	7 in x and 3 in y
+, -	numbers	addition, subtraction	$3.0 + x$	10
*, /	numbers	multiplication, division	$7/2$	3
++, --	number	pre or post increment, decrement	$x++;$	8 stored in x
==	any	test for equality	$x == y$	false
!	boolean	logical complement	$!(x == y)$	true
+=	numbers	$x+= 3;$ equivalent to $x = x + 3;$	$x+ = 3;$	11 stored in x
-=	numbers	$x- = 2;$ equivalent to $x = x - 2;$	$x- = 2;$	9 stored in x
=	numbers	$x= 4;$ equivalent to $x = 4*x;$	$x*= = 4;$	36 stored in x
/=	numbers	$x/ = 2;$ equivalent to $x = x/2;$	$x/ = 2;$	18 stored in x
%=	numbers	$x%= 4;$ equivalent to $x = x \% 4;$	$x%= 4;$	2 stored in x

Table 2.1: Common operators. The operations are done in order. The operator % is the modulus operator. For example, $23 \% 5 = 3$.

```
for (int n = 0; n < 10; n++)
    System.out.println("n = " + n); // only one statement is performed
```

Because the test is performed before the body of the loop is executed and the increment is performed after the loop has executed, the number 0 is printed but the number 10 is not. The increment expression `n++` is equivalent to `n = n + 1` (see Table 2.1). The loop counter, `n`, is declared as part of the initialization of the loop. Declaring variables where they are used is good practice because it is easy to see what the variable does. In this case `n` is a counter, and it is not necessary to document it. Declaring `n` within the loop limits its use to within the body of the loop. The section of code where a variable can be used is referred to as its *scope*.

Note that we indented the statements within the class definition, the statements with the body of the `main` method, and the statements within the `for` loop. Indentation is done to clarify the structure of the program and is ignored by the Java compiler.

Another construction that is used to perform repetitive actions is the `while` loop. The `while` statement starts by checking a condition. If this condition is true, the following statement or group of statements is executed. The following example shows how the `while` loop is used to print the numbers 0 through 9.

```
int n = 0;
while (n < 10) { // beginning of loop
    System.out.println("n = " + n);
    n++;
} // end of loop
```

Note how braces are used to group statements to define the statements within a class, within a method, or within control statements such as the `while` or `for` loop. Also note how single line comment statements are written using `//`. Comments are very important for the user, but are ignored by the Java compiler and do not affect the speed of the program or compilation.

In `MyFirstApp`, the body of the `main` method contains a statement that prints a string that

is defined by the characters in quotes. (Strings are sequences of characters and are discussed in more detail in Chapter ??.) We note that `System.out.println` is used to display output. In brief, this syntax means that we are accessing the object `out`, which is contained in class `System`, and invoking the method `println` of the object `out` by using the *dot operator*.

Note that the text to be printed is created by adding a number to a string. Java converts the numbers into strings before *concatenating* (combining two string variables). This use of the plus operator to concatenate is an example of *operator overloading*. The plus operator acts differently when operating on two strings than when operating on two numbers. The type of variable is clearly important in determining if $4 + 5$ should produce the integer number 9 or the string 45.

Exercise 2.2. While loop

- a. Replace the body of the main method in `MyFirstApp` class by the following code:

```
double sum = 0.0;
double maxChange = 1.0e-4; // maximum difference
double change = 1.0;
int n = 0;
while (change >= maxChange) {
    n++; // equivalent to n = n + 1
    double term = 1.0/(n*n);
    sum = sum + term;
    change = term/sum; // relative change
} // end of while loop
```

This code sums the terms in the series until the relative change in the sum is less than the desired amount.

- b. How many terms are required to achieve a relative change of less than 10^{-2} and less than 10^{-4} ? What do you think is the result of adding an infinite number of terms in the series $\sum_{n=1} 1/n^2$?

2.2 Static Methods

All Java programs are built from *classes*, which define a collection of related *data* and *methods*. Data is stored in variables which can refer to just about anything. However, there are two basic types of data, and these types are used very differently within a program. Primitive data types, such as `x` and `n`, are used to store numeric values and single characters. (There are eight primitive data types.) Composite data types, such `Complex` (see Section 2.4.3), group data in such a way that it can be referred to by a single variable.

Methods are operations on data. They are similar to functions or subroutines in other programming language with the important distinction that they have access to all data within a class. In most cases, classes cannot be used until they are instantiated using the *new* operator as explained in Section 2.3. However, some classes define *static* data and methods that can be used immediately. We discuss static methods first because they are used to define functions that can be used (invoked) anywhere in a Java program and because they are essential to starting a Java application.

Java provides many already defined mathematical methods in the `Math` class. These methods are static. For example, we can calculate the square root of 2.0 and set it equal to a double precision variable by the statement

```
double x = Math.sqrt(2.0); // example of the use of a static method
```

Note how we have used the dot operator to access the `sqrt` method of the `Math` class.

The method `Math.sqrt()` computes the square root of any value passed to it and returns the result. This method is independent of any data stored in the `Math` class and hence the method is static. Hence, we can use `Math.sqrt()` without first instantiating an object from the `Math` class. The distinction between static and non-static methods will become more apparent in Section 2.4, where we will see that we must first create an *object* or an *instance* of a class before we can use its non-static methods. In this section we will use only static methods.

The `Math` class provides many common mathematical methods, including trigonometric, logarithmic, exponential, and rounding operations, and predefined constants. Some examples that use the `Math` class include:

```
double theta = Math.PI/4; // constant pi defined in Math class
double u = Math.sin(theta); // sine of theta
double v = Math.log(0.1); // natural log of 0.1
double w = Math.pow(10,0.4); // 10 to the 0.4 power
double x = Math.atan(3.0); // arc tangent
```

The above statements illustrate that a class is described by its methods. These methods tell us what this class can do; we don't have to know about the details of how the methods are implemented. Note the Java convention that constants such as the value of π are written in uppercase letters, that is, `Math.PI`.

Exercise 2.3 asks you to read the `Math` class documentation and Exercise 2.4 asks you to use a trigonometric identity to test the accuracy of several of the methods in the `Math` class.

Exercise 2.3. The `Math` class

- Read the documentation for the `Math` class. The Java documentation is a part of most development environments and is available online at <http://java.sun.com/j2se/1.3/docs/api/>. It also can be downloaded to your computer.
- How many methods are defined in the `Math` class? How many constants?
- Describe the difference between the two versions of the arctangent method.
- Write a program to verify the output of various methods in the `Math` class.

We now list a program that tests the accuracy of the calculation of the trigonometric identity, $\sin x = \sqrt{1 - \cos^2 x}$. Create a file named `MathApp.java` and enter the following statements:

```
public class MathApp {

    public static void main(String[] args) {
        double theta = Math.PI/4.0; // angle in radians
        double a = Math.sin(theta);
```

```

    double b = Math.sqrt(1 - Math.cos(theta)*Math.cos(theta));
    System.out.println("sin(theta) = " + a);
    System.out.println("value of trigonometric identity = " + b);
    double difference = a - b;
    System.out.println("difference = " + difference);
}
}

```

Exercise 2.4. Trigonometric identity

Use the `MathApp` program to do the following:

- Determine if the trigonometric identity, $\sin x = \sqrt{1 - \cos^2 x}$, is satisfied exactly. Is the difference between the computed values of $\sin x$ and $\sqrt{1 - \cos^2 x}$ zero? If not, why not?
- Change the angle at which the trigonometric identity is being tested to $\pi/4000.0$. Does the difference change? Explain.
- Which variable do you think is most accurate, a or b ?

The `main` method is a static method, but has some important differences from most static methods. Because `main` does not calculate any values, its return value is `void`. In contrast, the return value for `Math.sin` is declared to be `double`. The arguments for the `main` method are also different than the `Math.sin` method. The `sin` method is passed a double variable, in contrast to the `main` method which is passed an array of strings, `String[] args`. (Arrays are defined in Section 3.4.) Although the argument to the `main` method is not used, it must be included.

The arguments of a method are called the *parameter list*. The parameters in the list are separated by commas and the type must be declared for each parameter in the list. If the method does not receive any parameters, the parentheses are still required. The type of data that is passed to a method must match the type of data in the method's *signature*, the combination of the method number, type, and order of arguments in the parameter list. For example, we cannot call the method `Math.sin("two")`, because the `Math.sin` method expects a number not a string.

We next define a new class that defines some hyperbolic functions that are not included in the `Math` class. The methods of `MyMath` can be invoked by statements similar to the following:

```

// test the hyperbolic tangent
double x = 0.5;
double y = MyMath.tanh(x);
System.out.println("x = " + x + " tanh(y) = " + y);

```

These statements should be inserted into the `main` method of the target class `MyMathApp`.

The hyperbolic sine (`sinh`), cosine (`cosh`), and tangent (`tanh`) functions can be defined in terms of the exponential function.

$$\sinh u = \frac{e^u - e^{-u}}{2} \quad (2.1a)$$

$$\cosh u = \frac{e^u + e^{-u}}{2} \quad (2.1b)$$

$$\tanh u = \frac{e^u - e^{-u}}{e^u + e^{-u}}. \quad (2.1c)$$

We implement these definitions in class `MyMath` as follows:

```
public class MyMath {

    private MyMath(){}          // prohibit instantiation

    public static final double MAX_EXPONENT = Math.log(Double.MAX_VALUE);
    public static final double MIN_EXPONENT = Math.log(Double.MIN_VALUE);

    public static double sinh(double u) {
        return (Math.exp(u) - Math.exp(-u))/2;
    }

    public static double cosh(double u) {
        return (Math.exp(u) + Math.exp(-u))/2;
    }

    public static double tanh(double u) {
        if (u > MAX_EXPONENT)
            return 1.0;
        else if (u < MIN_EXPONENT)
            return -1.0;
        else
            return (Math.exp(u) - Math.exp(-u))/(Math.exp(u) + Math.exp(-u));
    }
}
```

The term `public` in the first line means that the class can be used by other classes. The first two statements in the body of the class definition are examples of definitions of *named constants*. For example, we could define the speed of light *c* by

```
public static final double SPEED_OF_LIGHT = 3.0e8;
```

The keyword `static` means that this variable can be accessed using the class name. The keyword `final` means that it is not possible to change the value of `SPEED_OF_LIGHT` after it is given an initial value. By convention, named constants are written in uppercase letters with underscores used to separate the words.

The `Double` class defines the static constants, `Double.MAXIMUM` and `Double.MINIMUM`, as the maximum and minimum allowable value of the double primitive type. The `Double` class is distinct from the `double` primitive type. How would you find the largest and smallest values of type `int`? How would you find the largest and smallest values of type `int`?

We evaluate the constants `MAX_EXPONENT` and `MIN_EXPONENT` using the inverse exponential, the logarithm, to define the largest argument of the tanh function that can be evaluated accurately using exponentials. It is important to avoid using large arguments even though the tanh function is well defined.

The above implementation of the hyperbolic tangent introduces the `if` statement and the operator corresponding to less than. The logical operators evaluate expressions to produce boolean values of `true` or `false` (see Table 2.2). The `if` statement evaluates an expression and decides

which statement to execute. The following example shows how the `if` statement can be used to test for illegal negative numbers before the square root method is invoked.

```
if (x >= 0)
    y = Math.sqrt(x);
```

If the boolean expression produces a value of `true` then the statement is executed. Otherwise, the statement is skipped and the program continues. If there is only one statement to evaluate as in the above example, we will write it as two lines and indent the second line for clarity.

It is possible to evaluate more than one statement by enclosing a block of code in braces. A block of code is known as a *compound statement* and can be used anywhere a single statement is required. For example, we can write

```
if (x >= 0) {
    y = Math.sqrt(x);
    System.out.println("y = " + y);
}
```

The `if/else` statement is similar to the `if` statement except that it chooses between two statements. The first statement is executed if the expression is `true`; the second statement is executed if the expression is `false`.

```
if (x >= 0)
    y = Math.sqrt(x);
else
    y = Math.sqrt(-x);
```

`if` statements can be combined as shown in the definition of the `tanh` method. If the first expression is true, then the method exits and the value returned is 1.0. If the first expression is false, then a second `if` statement is executed. There are again two possibilities each of which exits with a different return value.

operator	description	sample expression	result
>	greater than	7 > 7	false
>=	greater than or equal	7 >= 7	true
<	less than	3 < Math.PI	true
<=	less than or equal	3 <= 7	true
==	equal	7 == 3 + 4	true
!=	not equal	7 != 15/2	true
&&	logical AND	7 > 2 && 5 < 3	false
	logical OR	7 > 2 3 < 5	true

Table 2.2: Common Java relational and logical operators.

So far our programs have consisted of no more than two files, for example, `MyMath.java` and `MyMathApp.java`. The Java compiler will be able to find the two files as long as they are in the same directory. In Section 2.4.2 we will learn how to make *packages* so that the files can be placed anywhere.

Exercise 2.5. MyMath class

- a. Write a program by replacing the body of the `main` method in the `MyMathApp` class with code that tests the three hyperbolic functions. For example, write code such as the following:

```
double x = 0.2;
double y = MyMath.sinh(x);
System.out.println("x = " + x + " sinh(x) = " + y);
```

Check your values using a calculator. Find the maximum and minimum values for the arguments that produce correct results for all three functions.

- b. Remove the exponent range check from the `tanh` method. How does this change effect the domain of the function?
- c. Test the numerical accuracy of the identities $\cosh^2 x - \sinh^2 x = 1$ and $\cosh 2x = \cosh^2 x + \sinh^2 x$.

2.3 Non-Static Methods

There are two general types of methods in Java, *static* and *non-static*. We have already used static methods to define mathematical functions and the static method `main` to start a Java application. The syntax that is used to write static methods is common to most modern programming languages discuss non-static methods.

Assume that we have two non-interacting particles constrained to move in one dimension. Each mass is modeled in a class named `ParticleFree`. Our goal is to write a small Java application that tests this class. Although the program will not run because we have not yet defined the `ParticleFree` class, `ParticleFreeApp` might be written as follows:

```
public class ParticleFreeApp { // ParticleFreeApp class begins

    public static void main(String args[]) { // main method begins
        ParticleFree p1 = new ParticleFree(2.0, 1.0);
        // create particle with y = 2, vy = 1
        ParticleFree p2 = new ParticleFree(5.0, 3.0); // create another particle
        System.out.println(" particle 1:" +p1);        // print state of particle 1
        System.out.println(" particle 2:" +p2);        // print state of particle 2
        double dt = 0.1;                               // time step for particle motion
        p1.move(dt);                                    // move particle 1
        p2.move(dt);                                    // move particle 2
        System.out.println(" particle 1:" +p1);        // print state of particle 1
        System.out.println(" particle 2:" +p2);        // print state of particle 2
    } // main method ends
} // ParticleFreeApp class ends
```

Note the differences in syntax from the `MathApp` program. In particular, note that we did not write

```
ParticleFree .move(dt);
```

but instead wrote

```
ParticleFree p1 = new ParticleFree(2.0, 1.0);
```

followed by

```
p1.move(dt);
```

This syntax change is a manifestation of the difference between static and non-static methods.

One of the most important object oriented programming concepts is the distinction between an object's definition, which is given by a class, and the actual object itself. The creation of an object requires that the class contain a special method known as a *constructor*. Constructors are easy to spot because they have the same name as the class. Unlike other methods, constructors do not have a return type. Constructors are called automatically by Java when an object is created using the `new` operator and allocate computer memory for the object. The following statement invokes the `ParticleFree` constructor and assigns the object to the variable `p1`.

```
p1 = new ParticleFree(2.0, 1.0);
```

Like other methods, constructors can have arguments, which are often used to initialize variables as is the case here. (In this case, the first parameter is the position of the particle and the second parameter is its velocity.)

The `ParticleFreeApp` class *instantiates* two `ParticleFree` objects and allows us to move these objects by invoking the `move` method of `p1` and `p2`. Objects are data as are the primitive data types, such as `int` and `double` which we have already encountered. However, an identifier that refers to an object does not contain the entire object, whereas a variable that refers to a primitive data type contains the actual data. An object refers to an address in memory (called a *pointer* in some languages) to where the object is stored. Java (unlike C) does not allow you to access this address directly, so you will sometimes read that Java does not support pointer arithmetic. The difference between primitive data types and objects is explored in Exercise 2.8.

2.4 Examples of Classes

We now introduce several examples of classes with non-static methods. As before, we introduce this concept by considering several examples.

2.4.1 One-Dimensional Motion

Consider the motion of a particle freely falling near the Earth's surface due to the gravitational attraction of the Earth on the particle. If we neglect air resistance, we know that the acceleration of the particle is constant with magnitude $g \approx 9.8 \text{ m/s}^2$. Let us adopt a coordinate system such that y increases upward. With this choice the acceleration $a = -g$. For this case of constant acceleration, the solutions of Newton's second law of motion can be expressed as

$$v(t) = v_0 - gt \tag{2.2a}$$

and

$$y(t) = y_0 + v_0t - \frac{1}{2}gt^2, \tag{2.2b}$$

where y_0 and v_0 are the initial position and velocity of the particle, respectively.

Clearly, the analytical solution in (2.2) is so simple that we do not need to write a program to compute the position and velocity of the particle. But we have to start somewhere, so let us discuss how we might write a program to compute the position and velocity of a freely falling particle at a particular time.

Our first step is to define a `ParticleFree` class. In this case it is straightforward to think of a particular particle as an object of the class `ParticleFree`. We know that a particle is characterized by its position, velocity, acceleration, and mass. For simplicity, we consider motion in the vertical direction only. Our class looks like the following:

```
public class ParticleFree extends Object {
    double mass = 1.0;
    double y,v;    // position , velocity
    static final double g = 9.8; // acceleration near earth's surface
    double a = -g; // acceleration

    public ParticleFree(double _y, double _v) {
        y = _y;
        v = _v;
    }

    public void move(double dt) {
        v = v + a*dt; // dt is the time step
        y = y + v*dt + 0.5*a*dt*dt;
    }

    public String toString() {
        return "y = " + y + "," + "v = " + v + "," + "a = " + a;
    }
}
```

Because the acceleration due to the earth's gravity is independent of the mass of a particle, we have set the particle's mass equal to unity. Note that we have chosen a coordinate system such that the height y and the velocity v of the particle decrease with time. We have defined `ParticleFree` such that it can "print" itself using the `toString()` method, that is, display its position, velocity, and acceleration. In addition, we have included the method `move` that uses the analytical solution of a freely falling body in one dimension to compute the position and velocity.

The first line of the `ParticleFree` class states that `ParticleFree` is a *subclass* of the `Object` class. The keyword `extends` declares that `ParticleFree` has all the functionality of `Object`. The `extends` declaration is optional in this case because all classes implicitly extend `Object`, but we have included this declaration here to show explicitly that all classes *inherit* from `Object`. We say that `ParticleFree` is a *subclass* of `Object` and the latter is a *superclass*. We will not explicitly extend `Object` in future classes that we write.

Because `ParticleFree` is a subclass of `Object`, it automatically *inherits* the methods of `Object`. One of the methods of `Object` is the `toString()` method, which converts the data stored in an object into a format suitable for displaying. To display the data in `ParticleFree`, we need to *override* the `toString()` method and customize it. Several methods can be defined with the

same name if the methods have a different set of parameters or signature.

The variables `y`, `v`, `a`, and `mass` are part of the class definition and are called *member* or *instance* variables. Their values can be different for every instance of the class. That is, every instance of `ParticleFree` is allocated memory to store its own copy of these variables.

The next group of statements define the *constructor* for the class `ParticleFree`. As explained earlier, a constructor is a special static method that is called to create an instance of a class. A constructor always has the same name as the class that it initializes and does not have a return data type. A constructor always exists. If we do not define at least one constructor, the Java compiler supplies a default constructor with no parameters.

The `ParticleFree` constructor requires two initialization parameters, the position and the velocity. We could define this constructor in different ways so as to unambiguously show the difference between these passed variables and member variables. Two possible ways are as follows:

```
public ParticleFree(double y, double v) {
    this.y = y;
    this.v = v;
}
```

or

```
public ParticleFree(double _y, double _v) {
    y = _y;
    v = _v;
}
```

Instance variables have *class scope* so they are accessible in any non-static method (including constructors) in the class. In this case, the variables `y`, `v`, `a`, and `mass` are defined such that they have class scope. Class scope begins at the opening left brace of a class definition and ends at the closing right brace of the class definition. Class scope means that any (non-static method) in the class can access `y`, `v`, `a`, and `mass` directly. However, local variables have precedence if there is a name conflict. In this case the local variables `y` and `v` in the parameter list have the same name as the class instance variables `y` and `v`. Hence, the constructor `ParticleFree` cannot use the names `y` and `v` to access the instance variables. Instead, the instance variables are referred to as `this.y` and `this.v`. The reference `this` always refers to the current object, which is the object within which the reference appears. Hence, `this.y` refers to the instance variable `y` defined in the current object. To avoid the confusion between instance and local variables with the same names, the second example adds an underscore to the variables in the parameter list. We will adopt the convention that a passed variable that has a name conflict with an instance variable will be renamed with an underscore.

We now write `ParticleFreeApp` so that it will compute the position, velocity, and acceleration of a freely falling particle at different times and display this information on the screen.

```
public class ParticleFreeApp {
    public static void main(String[] args) {
        ParticleFree p;
        p = new ParticleFree(20,0); // create a particle with x = 20 and v = 0
        double t = 0.0;           // elapsed time from instantiation
        double dt = 0.1;
```

```

    while (p.y >= 0) {
        p.move(dt);
        t = t + dt;
        // equivalent to System.out.println(t + "\t" + p.toString());
        System.out.println(t + "\t" + p); // \t is tab
    }
}
}

```

The first statement of the `main` method declares `p` to be an object of type `ParticleFree`. This statement does not create the object, but says only that `p` is a suitable name for an object of type `ParticleFree`. That is, `p` can store a reference to an object of type `ParticleFree` if and when we create one.

The second line with the keyword `new` actually creates an object of type `ParticleFree` and associates it with the name `p`. We can think of `new` as creating the instance variables of the object `p`. We can combine these two statements into the single statement:

```
ParticleFree p = new ParticleFree(20,0); // new particle with x = 20 and v = 0
```

In the definition of class `ParticleFree`, we defined a constructor with two parameters so that the instance variables `y` and `v` are initialized using the values of the appropriate parameters.

Method `move` takes one parameter and uses the analytical solution (2.2) of a freely falling body to compute the position and velocity of the body. The variable `dt` can be thought of as a place holder for the numerical values that will be passed to the method when it is executed. Note that we have defined the return type of method `move` to be `void`.

In our example of class `ParticleFreeApp`, the `main` method creates the object `p` of type `ParticleFree` with initial height 20 (meters) and zero velocity. Because we want to access the variable `y` associated with this instantiation of a `ParticleFree`, we use the dot notation and write `p.y`. The variable `t` is an example of a *local variable*, a variable that is defined and accessible only within the same method. Note the use of the `while` loop and how the coordinates of the particle are displayed.

Exercise 2.6. Freely falling body

- a. Compile and run `ParticleFreeApp` and determine the time it takes for a particle to reach the Earth's surface. Check that the computed result is consistent with the analytical solution.
- b. Write a class that instantiates two particles, `p1` and `p2` with different initial positions and velocities. Suppose that `p1.y = 20`, `p1.v = 0`, and `p2.y = 30`, `p2.v = 5`. Which particle reaches the ground first?

Exercise 2.7. Testing the `ParticleFree` class

As mentioned in the text, it is possible to put a `main` method in any class. Because `main` is a static method, we have to treat it just like it was in another file. That is, within the `main` method we must create an object of type `ParticleFree` before we can call any of its methods. Modify class `ParticleFree` by putting a `main` method in it, and show that it is necessary to create an object of the class before you use any of the methods or access any of the instance variables of `ParticleFree`.

Exercise 2.8. Difference between types of variables

a. Consider the primitive data types in the following code:

```
double x = 2.0;
double y = 3.0;
x = y;
System.out.println("x = " + x);
System.out.println("y = " + y);
```

What are the values of `x` and `y`? What if we then write

```
y = -4;
```

What are the values of `x` and `y` now? Write a short target application that displays the values. In this case, the identifiers `x` and `y` are primitive data types.

b. Now consider the class data types in the following:

```
p1 = new ParticleFree(-2.0, -1.0);
p2 = new ParticleFree(2.0, 1.0);
p1 = p2;
System.out.println(p1);
System.out.println(p2);
```

In this case the identifiers `p1` and `p2` refer to the same object. If we move either `p1` or `p2`, we would move the same particle. In fact, it is impossible to access the original particle 2 unless we saved a reference to this particle in another identifier.

As you use objects in various contexts, you will become aware that defining an object name does not create the object. For example, we could write the following:

```
ParticleFree p3;
```

Particle 3 does not yet exist because Java has created only the variable name. We must first create the object using the `new` operator before particle 3 can be used.

```
p3 = new ParticleFree(-2.0, 1.0);
```

We will also appreciate the fact that *instantiated objects can remember their state*. It is possible to define a static hyperbolic tangent function because each evaluation of the function produces exactly the same result. But if we want multiple particles to have different states, then the particle class must contain non-static variables and methods.

2.4.2 Packages and Access Modifiers

Anyone who has ever used a computer knows how difficult it can be to find a misplaced file on a hard drive. Directories, or folders, can make the job of organizing your work easier, but only if you follow consistent naming conventions. *Packages* are the Java equivalent of directories. Imagine the mess, not to mention naming conflicts, if thousands of Java class files that you and others wrote

were located in a single directory on your hard disk. Packages allow programmers to organize their code into logical units and provide a syntactic structure that enables programmers to use code in other directories. Creating a package is equivalent to creating a subdirectory that stores files containing Java source code.

Packages can have subpackages just as directories can have subdirectories. Various operating systems use slash, backslash, or a colon to designate the path from a directory to a subdirectory to a file. Java uses the dot notation which we have already encountered. For example, the `Math` class is located in the package `java.lang`, which means that the designers of Java can find the code for the `Math.java` class by starting in a directory called `java` and going to a subdirectory called `lang`. Whenever packages are used, Java requires that the first statement of the source code contain a *package declaration*. For example, the first statement inside the `Math.java` file is `package java.lang;`. The package declaration must exactly match the directory names.

The programs for this book is organized by chapter and is located in packages that end with the chapter number. However, because you will use the `MyMath` class in various places throughout the text, we suggest that the code for the `MyMath` class be placed in a package called `org.opensourcephysics.sip.extras`. To shorten the code listings, we will usually omit the package declaration in the narrative.

So far we have declared the methods of the several classes that we have written to be `public`, so that the methods can be used by any other class. However, we have not declared the instance variables of the various classes to be either `private` or `public`. If we do not specify the visibility level of a variable, the default is “package visibility,” that is, the instance variable can be accessed only within the class that defines it and within classes that are part of the same package. In contrast, `private` instance variables and methods can only be accessed from within their defining class. Finally, there is a fourth access modifier that provides an intermediate level of protection which is slightly less restrictive than the default. The *protected* access modifier states that a variable or a method is accessible to all classes in the same package and all subclasses even if the subclasses are not in the same package.

Many Java textbooks stress that it is not good programming practice to make the instance variables of a class `public` and all instance variables should be declared `private`. If an instance variable is `public`, then *any* other class can directly access the instance variable and change its value. Such a change might lead to unforeseen circumstances. For class `ParticleFree`, the mass does not affect the acceleration of a freely falling particle even though the mass of a particle is essential information in general. But for example, if we were to take into account drag resistance (as we do in Chapter 5), we would have to incorporate the dependence of the drag resistance on the mass (and shape) of the falling body. Because it is necessary to recalculate the drag coefficient when the mass changes, we should modify the `ParticleFree` class so that the mass variable is `private`. In that case we should also provide a method to access the mass of the particle and a method to set its mass:

```
public class ParticleFree {
    private double mass;
    double y,v,a;

    public void setMass(double _mass) {
        mass = _mass;
        // code to calculate drag goes here (see Chapter 3)
```

```
    }  
  
    public double getMass() {  
        return mass;  
    }  
}
```

For brevity, we have omitted the other methods. If we want to get/set the value of the mass of the particle, we would write something like

```
ParticleFree p = new ParticleFree();  
p.setMass(3.0);  
double mass = p.getMass();
```

Providing accessor methods such as the `getMass` and `setMass` methods in the above produces much extra code that may obscure the physics. Our approach will be to allow package visibility if there are only a few interrelationships. We assume that the package author knows what is going on inside his or her classes. However, we will in general not use `public` instance variables and will provide accessor methods when necessary. In Chapter 3, we will begin to use *protected* variables, which are accessible to all classes in the same package and all subclasses even if the subclasses are not in the same package.

2.4.3 Complex Numbers

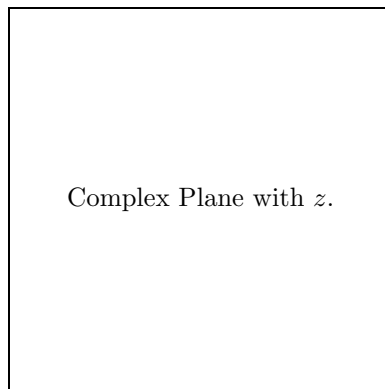


Figure 2.1: A complex number, z , can be defined by its real and imaginary parts, x and y , respectively, or by its magnitude, $|z|$, and phase angle, θ .

Complex numbers are used in physics to represent quantities such as alternating currents and quantum wavefunctions that have an amplitude and phase (see Figure 2.1). Java does not provide a complex number as a primitive data type, so we will build a class that implements some common complex arithmetic operations. This class is an explicit example of the statement that classes are effectively new programmer-defined types. If our new class is called `Complex`, we could test it using code such as the following:

```

public class ComplexApp {

    public static void main(String[] args) {
        Complex a = new Complex(3.0, 1.0); // complex number 3 + i1
        Complex b = new Complex(1.0,-1.0); // complex number 1 - i1
        System.out.println(a);          // print a using a.toString()
        System.out.println(b);          // print b using b.toString()
        a.conjugate();                   // complex conjugate of a
        System.out.println(a);          // print a
        Complex sum = b.add(a);          // add a to b
        System.out.println(sum);         // print sum
        Complex product = b.multiply(a); // multiply b by a
        System.out.println(product);     // print product
    }
}

```

Note that `ComplexApp` is the target class. Because the methods of class `Complex` are not static, we must first *instantiate* a `Complex` object with a statement such as

```
Complex a = new Complex();
```

The variable `a` is an object of class `Complex`. As before, we can think of `new` in the above as creating the instance variables and memory of the object. Compare the form of this statement to the declaration,

```
double x = 3.0;
```

A variable of class type `Complex` is literally more complex than a primitive variable because its definition also involves associated methods and instance variables.

Note that we have first written a class that uses the `Complex` class before we have actually written the latter. Although programming is an iterative process, it is a good idea to think about how the objects of a class are to be used first. Exercise 2.9 encourages you to do so.

Exercise 2.9. Complex Number Test

What will be the output when `ComplexApp` is run? Make reasonable assumptions about how the methods of `Complex` will perform using your knowledge of Java and complex numbers.

Clearly, we need to define methods that add, multiply, and take the conjugate of complex numbers and define a method that prints their value. We next list the code for the `Complex` class.

```

public class Complex {
    double real = 0;
    double imag = 0; // real, imag are instance variables

    public Complex() { } // use default value 0 + i0

    public Complex(double _real, double _imag) {
        real = _real;
        imag = _imag;
    }
}

```

```

public void conjugate() {
    imag = -imag;
}

public Complex add(Complex c) {
    /* result is also complex so need to introduce another
    variable that is of type Complex */
    Complex sum = new Complex();
    sum.real = real + c.real;
    sum.imag = imag + c.imag;
    return sum;
}

public Complex multiply(Complex c) {
    Complex product = new Complex();
    product.real = real*c.real - imag*c.imag;
    product.imag = real*c.imag + imag*c.real;
    return product;
}

public String toString() {
    // note method overriding
    if (imag >= 0)
        return real + " + i" + Math.abs(imag);
    else
        return real + " - i" + Math.abs(imag);
}
}

```

Notice how class `Complex` *encapsulates* (hides) both the data and the methods that characterize a complex number. That is, we can use the `Complex` class without any knowledge of how its methods are implemented or how its data is stored.

The general features of this class definition are as before. The variables `real` and `imag` are the instance variables of class `Complex`. In contrast, the variable `sum` in method `add` is a local variable because it can be accessed only within the method in which it is defined.

In this case there are two constructors that are distinguished by their parameter list. The constructor with two arguments allows us to initialize the values of the instance variables. As mentioned on page 25, if a constructor is not defined, Java defines a default constructor with no parameters. However, once we add at least one constructor to a class, then Java does not create a default constructor.

The most important new feature of the `Complex` class is that we have used the class definition itself to define the return type of methods `add` and `multiply`. One reason we need to return a variable of type `Complex` is that a method returns (at most) a *single* value. For this reason we cannot return both `sum.real` and `sum.imag`. More importantly, we want the sum of two complex numbers to also be of type `Complex` so that we can add a third complex number to the result. Note also that we have defined methods `add` and `multiply` so that they do not change the values of the instance variables of the numbers to be added, but create a new complex number that stores the

sum. Finally, the class definition includes an example of how to write comment statements that span multiple lines.

Another way to represent complex numbers is by their magnitude and phase, $|z|e^{i\theta}$. If $z = a + ib$, then

$$|z| = \sqrt{a^2 + b^2}, \quad (2.3a)$$

and

$$\theta = \arctan \frac{b}{a}. \quad (2.3b)$$

We investigate this alternative representation in Exercise 2.10.

Exercise 2.10. Complex Numbers

- Add methods to get the magnitude and phase of a complex number, `getMagnitude` and `getPhase`, respectively. Add test code to invoke these methods. Be sure to check the phase in all four quadrants.
- Create a new class named `ComplexPolar` that stores a complex number as a magnitude and phase. Define methods for this class so that it behaves the same as the `Complex` class. Test this class using the code for `ComplexApp`.

Exercise 2.11. Final

The structure of the `Math` class defined by Java looks something like

```
public final class Math {

    public static final double PI = 3.14159265358979323846;
    public static final double E = 2.7182818284590452354;

    public static double cos(double a) {
        // code to calculate cos goes here
    }
}
```

Look at the online Java documentation or a Java textbook (see the references for suggestions) and read about the significance of `final` in the `Math` class definition. What is the significance of the keywords `public static final` in the definition of π ? How would you modify the `MyMath` class so that its method definitions cannot be changed?

Exercise 2.12. Static variables

Consider what happens when you make the following changes in the `Complex` class:

```
public class Complex {
    static int counter = 0; // # of complex numbers that have been created
    int index = 0;

    public Complex (double _real, double _imag) {
        index = ++counter;
    }
}
```

Because `counter` is a static variable, there is only *one* value for all objects that are instantiated from the `Complex` class. When a new `Complex` object is created, `counter` is increased by unity and stored in the variable `index`. Change the `toString` method to print the index in addition to the real and imaginary components. Write a program that illustrates these properties.

2.5 Summary

So far we have seen that classes and methods form the basis of *object oriented programming*. A note of encouragement is in order here; it will take time and practice to become a good object oriented programmer. There is much about Java that is slow to reveal itself, and we have only scratched the surface of object oriented programming in this introductory chapter. Object oriented programming is a design philosophy that will shape the structure of the programs that we write. But as with many seemingly straightforward ideas, for example, Galilean and Lorentz invariance, the full implication of object oriented programming may not be immediately obvious.

Appendix 2A

type	contains	default	size	range
boolean	<code>true</code> or <code>false</code>	<code>false</code>	1 bit	NA
char	unicode character	<code>\u0000</code>	16 bits	<code>\u0000</code> to <code>\uFFFF</code>
byte	signed integer	0	8 bits	-128 to 127
short	signed integer	0	16 bits	-32768 to 32767
int	signed integer	0	32 bits	-2147483648 to 2147483647
long	signed integer	0	64 bits	-2^{63} to $2^{63} - 1$
float	floating point	0.0	32 bits	$\pm 1.4\text{E-}45$ to $\pm 3.4028235\text{E}+38$
double	floating point	0.0	64 bits	$\pm 4.9\text{E-}324$ to $\pm 1.7976931348623157\text{E}+308$

Table 2.3: Primitive data types, their memory requirements, and their range.

References and Suggestions for Further Reading

Stephen J. Chapman, *Java for Engineers and Scientists*, Prentice-Hall (2000).

Bruce Eckel, *Thinking in Java*, second edition, Prentice-Hall (2000). This text also discusses the finer points of object-oriented programming. See also <http://www.mindview.net/Books/>.

David Flanagan, *Java in a Nutshell*, third edition, O'Reilly (2000).

Walter Savitch, *Java: An Introduction to Computer Science and Programming*, second edition, Prentice-Hall (2001).