Chapter 2: Object-oriented programming

This chapter will introduce you to object-oriented programming, a central feature of Java programming. We'll begin with an example illustrating why objects are a helpful addition to a programming language. Then we'll see how to construct and use objects. After that, we'll make a quick study of some of the objects that Java provides for us.

1. A Motivating Example

In the first chapter, we saw that Java provides some simple data types, such as `int` and `double`, that are useful for arithmetic operations. Mathematicians will wish to think of these as integers and real numbers, recognizing that they are only approximations as they are represented by a finite amount of memory in a computer.

Now suppose we wish to do arithmetic with the Gaussian integers. These are, you will recall, the set of complex numbers whose real and imaginary parts are integers:

$$Z[i] = \{\, a + bi \mid a, b \in Z \,\}.$$

This set of numbers is similar to the integers in that addition, subtraction and multiplication are allowed and, though the set is not closed under division, there is a Division Algorithm analagous to that for the integers.

In thinking about how to represent a Gaussian integer inside a Java program, we can try to use two `int`s or perhaps an array `int[]` of two integers. Here is a fragment of code that multiplies $2 + i$ and $-3 + 2i$ and stores the result in an array called `product`:

```
int[] gi1 = new int[2];
int[] gi2 = new int[2];
gi1[0] = 2;  gi1[1] = 1;
gi2[0] = -3;  gi2[1] = 2;
int[] product = new int[2];
product[0] = gi1[0]*gi2[0] - gi1[1]*gi2[1];
product[1] = gi1[0]*gi2[1] + gi1[1]*gi2[0];
```

This feels cumbersome. We think of a Gaussian integer as a monolithic entity yet the program forces us to juggle the real and imaginary parts.

2. Objects to the rescue

Java objects give us a way of defining new data types. In particular, we can define a new data type that corresponds to our Gaussian integers. What would we want from such a data type? Certainly, we would like to be able to store the numerical value of a Gaussian integer. This means we should have a location to store the real and imaginary parts. Also, we would like some methods that enable us to do arithmetic with Gaussian integers. This is exactly what an object is: a collection of data (called the *state* of the object) and some methods that can change or examine the data (called the *behavior* of the object).

Here is a Java class file that we will use to represent Gaussian integers.

```
public class GaussianInteger {
    int real, imag;
    public GaussianInteger(int a, int b) {
        real = a;  imag = b;
    }
    public GaussianInteger conjugate() {
        return new GaussianInteger(real, -imag);
    }
    public String toString() {
        return real + " + " + imag + " i";
    }
    public static final GaussianInteger ZERO = new GaussianInteger(0, 0);
    public static GaussianInteger multiply(GaussianInteger gi1,
                                           GaussianInteger gi2) {
        return new GaussianInteger(gi1.real*gi2.real - gi1.imag*gi2.imag,
                                   gi1.real*gi2.imag + gi1.imag*gi2.real);
    }
}
```

At first glance, this does not look too unfamiliar. This file contains the definition of a class called `GaussianInteger` and we see that a few variables are declared and a few methods defined. Something is a little strange, however, since there is no `main` method. This means that we are not able to execute this class directly.

3. How do we use it?

When we use existing data types, such as `int`s, we can create a variable and initialize its value as

```
int x = 2;
```

To achieve the same result with a `GaussianInteger`, we can use a statement such as

```
GaussianInteger a = new GaussianInteger(2, 1);
```

We may think of the result as a representation of the Gaussian integer $2 + i$ stored in a variable `a`. We say that `a` is an *instance* of a `GaussianInteger` object through the process of *instantiation*.

The syntax of this expression, especially the use of the keyword `new`, is, well, new to us. It causes several things to happen. First, we may think of it as creating a little box somewhere in the computer's memory. Inside this box are two variables given by the declation

```
int real, imag;
```

Second, it causes the execution of the method

```
public GaussianInteger(int a, int b) {
    real = a;  imag = b;
}
```

This method, with its slightly unusual syntax, is called a *constructor*. In this example, it has the effect of initializing the two variables, `real` and `imag`, inside the box. Finally, the causes the variable `a` to contain the memory address of the box containing these two variables.

Besides these two variables, the box contains a copy of each method *not* modified by the word `static`. In this case, the box representing the Gaussian integer $a = 2 + i$ has the methods `conjugate` and `toString`. Remember that I said in Chapter 1 that a variable is accessible anywhere within the smallest block of code containing its declartion. The variables `real` and `imag` are therefore accessible from methods in the instance in which they reside.

The definition of the method `conjugate` should now be clear to you. We begin with an instance of a `GaussianInteger` and create a new instance with the same real part and opposite imaginary part, which we then return as the result of the method call.

You might ask now how we can access the variables and methods stored in this box. It's quite simple really. For instance, if we would like to know the value of the variable `real` in the instance `a`, we simply use the syntax `a.real`. You will see this usage in the definition of `multiply`. In the same way, to gain access to one of `a`'s methods, we say, for instance, `a.conjugate()`. In this way, you see that the `GaussianInteger` data type does indeed give us a means of representing a Gaussian integer as a monolithic entity together with some methods to operate on it.

A class may provide more than one constructor. For instance, we may wish to give a programmer a convenient way to define real Gaussian integers through a second constructor:

```
public GaussianInteger(int a) {
    real = a;  imag = 0;
}
```

The following program defines two Gaussian integers a and b and prints the conjugate of a and the product ab.

```
public class GaussianArithmetic {
    public static void main(String[] args) {
        GaussianInteger a = new GaussianInteger(2, 1);
        GaussianInteger b = new GaussianInteger(-3, 2);
        GaussianInteger aConjugate = a.conjugate();
        GaussianInteger product = GaussianInteger.multiply(a, b);
        System.out.println("The conjugate of " + a + " is " + aConjugate);
        System.out.println(a + " times " + b + " = " + product);
    }
}
```

Let's look a little more closely at the line

```
        GaussianInteger product = GaussianInteger.multiply(a, b);
```

since it has a slightly different format. You should notice in the definition of the class `GaussianInteger` that there was one variable, `ZERO`, and one method, `multiply`, modified by the word `static`. We call such a variable a *class field* and a method a *class method*. These are variables and methods

that are not attached to any one instance of a `GaussianInteger`. Instead, they are variables and methods useful for working with general `GaussianIntegers`. For instance, the additive identity is an important special instance of a `GaussianInteger` so it is included as a class field and may be accessed by `GaussianInteger.ZERO`. Moreoever, the method `multiply` does not naturally belong to a single instance of a `GaussianInteger`. For this reason, it is made a class method and accessed as shown above.

To use a loose analogy, "class" refers to fields or methods that apply to a set, in this case, the Gaussian integers, whereas "instance" refers to fields or methods that apply to an individual elements of a set.

One more word about the modifier `final` for the class field `ZERO`. This means that the value of this variable is not allowed to be modified. In this case, this modifier makes sense. As the author of this class, I am not allowing you to modify the value of the additive identity of the Gaussian integers.

A class may use the same name for different methods provided the argument lists of the two methods are different. For example, we may add an instance method `multiply` as follows:

```
public GaussianInteger times(GaussianInteger gi) {
    return GaussianInteger.multiply(this, gi);
}
```

The keyword `this` refers to the instance of `GaussianInteger` in which the method resides. To obtain the result of `a` multiplied by `b`, we may now say `a.multiply(b)`.

You may remember that we met the `new` keyword earlier when we instantiated an array

```
int[] previousRow = new int[3];
```

This seems to imply that arrays are objects and indeed they are. For example, an instance of an array has a field `length` that is set to the number of elements in the array. For several reasons, however, arrays are somewhat exceptional examples of objects and do not always obey the syntax expected of typical objects.

Exercise 1: Add class methods

```
public static GaussianInteger add(GaussianInteger a, GaussianInteger b)
```

and

```
public static GaussianInteger subtract(GaussianInteger a, GaussianInteger b)
```

Test your methods by modifying `GaussianArithmetic` to compute the sum and difference of a and b.

Exercise 2: Add an instance method

```
public double getModulus()
```

that returns the modulus $|a|$ of a Gaussian integer a. You will need to use the method `Math.sqrt(double x)` to compute the square root. (You should be able to understand this syntax now: it refers to a `static` method called `sqrt` in a class `Math`.)

Exercise 3: There is a Division Algorithm for the Gaussian integers: that is, given Gaussian integers a and b where $b \neq 0$, there are (not necessarily unique) Gaussian integers q and r such that

$$a = bq + r$$
$$|r| < |b|$$

In fact, to find q you may view b as a complex number and find the closest Gaussian integer to the complex number a/b. This shows that r may be chosen to satisfy

$$|r| \leq \frac{|b|}{\sqrt{2}}.$$

Add an instance method

```
public static GaussianInteger[] dividedBy(GaussianInteger b)
```

that returns an array with two `GaussianIntegers`, q and r, that result from dividing the Gaussian integer represented by the instance by b. You will want to use the expression `(int) Math.round(double x)` to find the closest integer to a real number.

Exercise 4: Implement the Euclidean Algorithm to find the greatest common divisor of two Gaussian integers by creating a class method

```
public static GaussianInteger gcd(GaussianInteger a, GaussianInteger b)
```

Exercise 5: Remember that a Gaussian integer p is prime if it is not a unit and if $p = ab$ then one of a and b is a unit. Write an instance method

```
public boolean isPrime()
```

that determines whether a `GaussianInteger` represents a prime. You may find it convenient to add a class field

```
public static final GaussianInteger[] UNITS
```

where the array contains the four units ± 1 and $\pm i$.

Exercise 6: Modify `GaussianArithmetic` to test the methods you have constructed.

Exercise 7: Design a class called `Rational` that represents a rational number and provides methods for doing arithmetic with rationals, determining when one `Rational` is equal to another and comparing whether one `Rational` is larger than another.

You may want to be careful about dividing by zero. While Java has an especially elegant way to deal with situations like this using `Exceptions`, I prefer that you not investigate it fully at this time. Instead, you may wish to run this code:

```
public class Divide {
    public static void main(String[] args) {
        double r = 1/0;
        System.out.println(r);
    }
}
```

and see how you can adapt it to detect division by a zero `Rational` object and stop the execution of the program.

Exercise 8: Design a class that represents polynomials in one variable with rational coefficients. Include an instance method that reports the degree of the polynomial and class methods to implement the Division Algorithm and the Euclidean Algorithm.

4. What have we gained?

There are several advantages to organizing code in this way. First, it is extremely easy to reuse code created for one project in a later one. For instance, a class can be used without understanding the details of how the methods work, only what methods are available and how they are accessed. In fact, Java provides a convenient way to construct HTML documentation for a given class that gives exactly this information through a program called `javadoc`. We will explore this further in the next section.

Second, different pieces of a program can be separated from one another. In particular, variables may be "localized" so that they are only accessible from the smallest possible portion of the code for which they are needed. The result is that variable names can be chosen to most accurately reflect their meaning and the chance that a variable is accidently modified is minimized. This makes it easier to write and maintain the code.

Finally, you will likely find that it is extremely natural to think about code in this way. The words *class* and *object* are well chosen for a class represents, loosely speaking, a collection of things and an object represents a particular element in the collection. One might argue that this is a fundamental way of organizing the phenomenological world: to note what is common about a collection of related things and then describe abstractly the properties that make such a thing.

5. Java's pre-defined objects

Now that we have constructed our class `GaussianInteger`, we are free to use it as often as we like and to share it with others. In the same way, the Java language provides many classes for us to use directly and documentation to help use them effectively.

When Java 1.0, the first public version of the language, appeared, it contained 212 classes, a modest number when seen from today's perspective. While many of these classes are still used, in one form or another, Java 1.0 is now considered obsolete. Many new features have been added and several portions of the language, most notably the graphical interfaces, have been redesigned according to a different paradigm.

A few years later, Java 1.1 appeared bringing with it 504 classes. The performance of the virtual machine was improved and the way in which events (see Chapter 6) are handled was redesigned. While this version is now considered outdated, it is still used widely since many web browsers support, by default, Java 1.1.

Tripling the number of classes, Java 1.2 represented a significant step forward for the language. For us, this release is notable for it introduced the powerful graphical tools known as Java2D that we will study in the next chapter. Due to the range of new features it introduced, Java 1.2 is now commonly referred to as Java 2. Once the language grew to this size, it was impractical for web browsers to try to keep up with the rapid pace of developments in the language. Sun, the creator of Java, has provided a convenient way around this through the Java Plug-in (see Appendix A).

The most recent version of Java, Java 1.4, defines 2991 classes (!), a huge increase from Java 1.0's 212. This release mainly added new features rather than modifying the way in which existing classes worked.

With such a daunting number of classes, we need to determine quickly whether there is a class already defined that we can use in a project or if we will need to write it ourselves. For this, we have two useful tools. First, classes may be grouped together into *packages*, collections of classes that work toward some common aim. For instance, the package `java.io` provides classes to help with input and output, `java.net` handles networking issues and `java.awt` provides graphical capabilities. One package, `java.lang`, provides the basic classes in the language. Java 1.4 gives us 135 packages, which again seems overwhelming. However, many of these are subpackages of others; for example, `java.awt.geom` gives some geometrical tools to use in graphics.

Documentation for all 2991 classes exists in HTML format and may be viewed by going to

```
http://java.sun.com/
```

and following the link labelled APIs. The first page of the API has hypertext links to documentation for each package. The documentation for a package contains hypertext links to documentation for each class in the package. The documentation for each class contains a complete listing of each class field, instance field, constructor, class method and instance method. Generally, there is a description of how to use a method if it is not entirely clear from its syntax.

Exercise 9: Explore the `java.lang` package and, in particular, the `String` class. This class represents an array of characters as, for instance, in a piece of text. As it is so commonly used, the `String` class is somewhat exceptional: an instance can be created, without using the `new` keyword, by enclosing the characters in double quotes as we have seen. The API also shows a few other ways to instantiate `String`s.

Find a method that returns the number of characters in a `String`.

How could you extract the first three characters from a `String`?

The `char` data type is a 16 bit number used to represent characters. For instance, a typical assignment looks like

```
char c = 'A';
```

Exercise 10: How would you replace every occurrence of "A" in a `String` by "a"?

Exercise 11: Write a method that prints the second character in a `String` after checking that the `String` has at least two characters. You will need to be careful here since the fragment

```
char c = 'A';
System.out.println(c);
```

prints the numerical equivalent of "A" in the ASCII character set. You will need to convert `c` to a `String` before printing it.

Exercise 12: Write a method that converts a `String` into a new one with the first character upper case and each remaining character lower case.

Exercise 13: Read about the `char` primitive data type. Write a method that implements the Caesar cipher in which the letters of the alphabet are cyclically shifted by three characters.

Exercise 14: A Java variable name is valid if it begins with a letter or underscore and is followed by any number of letters, digits or underscores. Write a method that determines whether a `String` satisfies this condition.

Exercise 15: Explain why there are only class methods in the class `Math`.

Exercise 16: Explain the syntax of the instruction

```
System.out.println("Hello");
```

When processing a class file, the Java compiler needs to look up references to any classes used in the class file. Since there is such a large number of classes and packages, the compiler asks that we provide some guidance on where to find the classes we are using. For instance, if we give no guidance, the compiler will only look for classes in the package `java.lang`. However, we can use a class in, say, `java.util` by including an `import` statement in the class file before the class declaration. For example, we might use something like this:

```
import java.util.*;
public class A {
```

in the definition of class `A` so that the compiler will search through classes in `java.util` if it encounters a class it does not recognize. This is similar to loading libraries into a Maple worksheet using the `with` command.

We will discuss later how to assign a Java class we develop to a package.

6. Is everything an object?

Remember that every piece of Java code that you write is contained in a class file. Indeed, objects lie at the heart of the Java programming language. However, not every data type in a piece of Java code is an object: these are the *primitive* data types such as `boolean`, `int` and `double`. There are important distinctions between primitive data types and instances of objects.

Remember that an instance of a class may be thought of a box containing instance fields and methods inside the computer's memory. When we define a variable using an instruction such as

```
GaussianInteger x = new GaussianInteger(2, 1);
```

the content of `x` is actually the address in memory of this box. This is important:

`x` *contains the memory address of the* `GaussianInteger` *object.*

This is similar to the notion of a pointer in the C programming language. However, when we work with a primitive data type, defining an `int`, say, by the instruction

```
int a = 1;
```

the contents of `a` is actually its *value*, in this case 1. This produces interesting effects when copying the contents of one variable into another and when passing arguments into methods.

Exercise 17: Predict the result of the following piece of code:

```
int a = 1;
int b = 2;
a = b;
b = 3;
System.out.println(a);
GaussianInteger x = new GaussianInteger(2, 1);
GaussianInteger y = new GaussianInteger(-3, 2);
x = y;
y.real = 4;
System.out.println(x);
```

and explain its behavior.

Exercise 18: Predict the result of the following piece of code:

```
public class Test {
    public static void main(String[] args) {
        int a = 1;
        intMethod(a);
        System.out.println(a);
        GaussianInteger x = new GaussianInteger(2, 1);
        giMethod(x);
        System.out.println(x);
    }
    public void intMethod(int n) {
        n = 10;
    }
    public void giMethod(GaussianInteger gi) {
        gi.real = 0;
    }
}
```

and explain its behavior.

Exercise 19: What is the result of the following code:

```
GaussianInteger x = new GaussianInteger(2, 1);
GaussianInteger y = new GaussianInteger(2, 1);
if (x == y) System.out.println(‘‘x equals y’’);
else System.out.println(‘‘x does not equal y’’);
```

7. Inheritance

It often happens that we have a class closely related to a class we would like to construct. Maybe we would like to modify one method or perhaps add a few methods to add some new kind of functionality. Java provides a means for doing this through *inheritance*.

Here is an example of how it works. Suppose we have a class `A` as follows

```
public class A {
    int a;
    public A(int i) {
        a = i;
    }
    public void printPhrase(String s) {
        System.out.println(s);
    }
    public void identify() {
        printPhrase("Class A");
    }
}
```

We can define a new class that inherits all of `A`'s fields and methods as follows:

```
public class B extends A {
    public B(int i) {
        super(i);
    }
    public void identify() {
        printPhrase("Class B");
    }
    public int instanceField() {
        return a;
    }
}
```

The phrase `extends A` in the class declaration means that `B` is a *subclass* of `A`. Therefore, `B` contains every field and every member that `A` has. In this example, this means that `B` has an instance field `a` and methods `printPhrase` and `identify`. In addition, `B` defines a new method `instanceField` that `A` does not have.

Notice that `B` defines its own version of the method `identify` that is different from the method of the same name inherited from `A`. When a method is invoked on an instance of `B`, the Java interpreter first looks in `B` to find the method. If it finds it, it is executed. If not, it looks in the superclass `A` for the method. When the subclass defines a new version of a superclass method, we say the subclass *overrides* the superclass method.

The constructor of `B` appears to be new as well. The keyword `super` causes the constructor of the superclass, in this case `A`, to be run. Here, an instance of `B` is created by its constructor calling the constructor of `A`.

Exercise 20: What is the result of the following fragment?

```
B b = new B(2);
b.identify();
System.out.println(b.instanceField());
```

Let's look at an example of how we might use this. First, you might take a moment to study the class `Vector` in the package `java.util`. In some sense, this is very much like an array except for two fundamental differences. The type of data that an array can hold is part of its declaration. However, a `Vector` can hold instances of many different types of objects. Whereas arrays must have a specified size, `Vectors` do not; we can keep adding more and more objects to a `Vector`.

A typical data structure in computing is a *queue*. You might think of this as a list of data in which data is added to one side of the list and removed from the other side. No doubt you have experienced a queue when waiting in line to buy tickets to a movie. Print jobs are typically controlled by a queue: jobs are printed in the order in which they go into the queue.

If we think about creating a queue in Java, we soon recognize that `Vector` provides methods to add and remove `Objects` from a list whose size can grow arbitrarily large. It does not, however, provide explicit methods to insert an `Object` into one end of the queue and remove it from the other. We can create a subclass `Queue` of `Vector` to add in this functionality.

```
import java.util.Vector;
public class Queue extends Vector {
    public void insertElement(Object obj) {
        insertElementAt(obj, 0);
    }
    public boolean hasMoreElements() {
        return size() > 0?  true :  false;
    }
    public Object removeElement() {
        Object object = elementAt(size() - 1);
        removeElementAt(size() - 1);
        return object;
    }
}
```

Exercise 21: A *stack* is another kind of data structure, similar to a queue, in which objects are added to and removed from the same side of the list. Write a new class `MyStack` that extends `Vector` to implement this behavior. You might note that there is a class `Stack` already provided in `java.util`.

A few remarks need to be made now. First, a class cannot extend two different classes. The reason for this is clear: confusion would result if the two superclasses each defined methods with the same name. Second, if `B` extends `A`, there is no reason that another class `C` cannot extend `B`. In this way, the relation of subclasses gives the set of Java classes the structure of a tree. At the root of the tree is the class `Object` that every class implicitly extends.

This explains one mystery you may have noticed in our `GaussianInteger` class. That class defined a method `toString` that gave instructions for how to print an instance of `GaussianInteger`. You may have noticed that if `a` was an instance of `GaussianInteger`, we could print its `String` representation by simply using `System.out.println(a)`. This is because `Object` has an instance method `toString` that is used to represent any `Object` as a `String`. Generally speaking, this method reports the name of the class and the memory address of the instance. In our definition of `GaussianInteger`, the `toString` method overrode `Object`'s `toString`. This method was then invoked by the print command.

8. Visibility

We have seen the keyword `public` quite often in our examples. We are now able to give an explanation of it.

We have seen that an instance of a class has instance fields and methods and that these fields and methods can be accessed from outside the instance. However, it is sometimes desirable to have some data only accessible by the instance itself. For instance, suppose we modify the `GaussianInteger` class to add an instance field that records the modulus of the Gaussian integer. The new constructor might look like this:

```
int real, imag;
double modulus;
public GaussianInteger(int a, int b) {
    real = a;  imag = b;
    modulus = Math.sqrt(a*a + b*b);
}
```

In this example, `modulus` depends on `real` and `imag`. We would not want another class to modify `modulus` without modifying `real` and `imag` as in the following fragment

```
GaussianInteger x = new GaussianInteger(2, 1);
x.modulus = -10;
```

For this reason, we would like to hide the field `modulus` from other classes. We can do this with the `private` keyword:

```
int real, imag;
private double modulus;
public GaussianInteger(int a, int b) {
    real = a;  imag = b;
    modulus = Math.sqrt(a*a + b*b);
}
```

With this change, a compiler error will result if another object attempts to reference the `modulus` field of a `GaussianInteger` object.

By contrast, the `public` keyword grants access to a field or method to anyone who asks. There are also `protected` and `package` visibility modifiers that lie somewhere between these two extremes. If the visibility is not explicitly stated, it is by default set to `package`, loosely meaning the member is visible within the same package but not outside.

If the visibility of a variable is `private`, it can still be accessed through a `public method`. For instance, we could add a method to our `GaussianInteger` class

```
public double getModulus() {
    return modulus;
}
```

Generally speaking, `public` fields should be used with caution. Imagine we first write our `GaussianInteger` class with `real` and `imag` having `public` visibility. We then publish the class and others

begin writing code and accessing these fields directly. Later, when we decide to add the `modulus` field, it is not guaranteed that the values of `real`, `imag` and `modulus` are in sync since another piece of code could modify `real` and not `modulus`.

Instead, it is preferable to provide public methods for setting and retrieving the values of instance fields. In our example, we could add

```
public void setReal(int r) {
    real = r;
    modulus = Math.sqrt(real * real + imag * image);
}
public int getReal() {
    return real;
}
public void setImag(int i) {
    imag = i;
    modulus = Math.sqrt(real * real + imag * image);
}
public int getImag() {
    return imag;
}
```

You will see many examples of this paradigm in the Java documentation.

In the same way, a class may have methods that help an instance do its work but that are not meant to be accessed outside the instance. The visibility modifiers work for this purpose as well.

Exercise 22: Explain the meaning of each piece of the class `WelcomeToJava` that we met at the beginning of Chapter 1.