Classes and Objects

Motivation for Classes

We can write some fairly complicated programs in a single main method, and we can do just about anything using a bunch of static methods in a single file. However, there is a lot more to programming than just getting it to work – we also want to be able to easily reuse pieces of our code, and for our programs to be easy for other people to read. This requires us to think about good *design* for our programs instead of just functionality.

Program Design

Computer programmers almost always work in teams, which means it is vital that all team members can use and understand code written by others in the group. If you think about how a big program (like a popular computer game) would look if it was written in a single file...it would be a nightmare to read. Programs like that tend to have millions of lines of code – it would be very difficult to ever find what you were after.

It is much easier to read code that is divided into many files and many methods by functionality. That way, you could go directly to the section of code you were interested in without having to wade through everything else. When each method solves a small piece of the problem, and each file holds a group of methods that do related things, it's very easy to figure out what's going on.

New Data Types

Creating programs with multiple classes also gives us the option to create new specialized data types. Later in this chapter, we will see how to classes into data types, complete with their own fields (pieces of relevant information) and methods to operate on those fields. For example, we will be able to create a Rectangle type that store the width and height for the rectangle, and that has area and perimeter methods that operate on the width and height.

Class Syntax with Static Methods

We have already seen how to write several methods within one class. Now, we will learn how to create several classes with different methods. This section will not discuss how to create *objects* (instances of classes) – for now, we're just dividing methods into different files as an organization trick.

Example: Separate Class with Static Methods

Suppose we want to create a lot of methods that perform mathematical operations (average, round, max, min, etc.). It would be nice to be able to reuse these methods in other projects, so we will want to divide them into a separate class. (Actually, there is a `Math` class in the Java libraries that contains these methods, but we will create our own.)

Creating a separate class with static methods is exactly like the classes we've written in the past – the only difference is that the separate class will not have a `main` method. Here is our `MathOps` class:

```
//stored in the file MathOps.java
public class MathOps {

     //assumes arr has at least one element
     public static int max(int[] arr) {
          int m = arr[0];
          for (int i = 1; i < arr.length; i++) {
               if (arr[i] > m) m = arr[i];
          }

          return m;
     }

     //assumes arr has at least one element
     public static int min(int[] arr) {
          int m = arr[0];
          for (int i = 1; i < arr.length; i++) {
               if (arr[i] < m) m = arr[i];
          }

          return m;

     }

     public static int round(double num) {
          if ((int)(num+0.5) > (int) num) {
               //num must have decimal of .5 or higher
               //round up
               return (int)(num+0.5);
          }
          else {
               //round down
               return (int)(num-0.5);
          }
     }

     public static double avg(int[] arr) {
          int sum = 0;
          for (int i = 0; i < arr.length; i++) {
               sum += arr[i];
          }

          //need to cast to a double to avoid integer division
```

```
            return sum / (double) arr.length;
        }
    }
```

Calling Static Methods from a Different Class

Now, suppose we want to create a separate class with a `main` method that uses the `MathOps` class. When we call static methods from another class, we will use the format:

```
ClassName.methodName(params);
```

Here, `ClassName` is the name of the class that contains the method (in this case, `MathOps`), and `methodName` is the name of the method we want to call. Here is a program that asks the user for 10 numbers, and then prints the maximum, minimum, and average of those numbers:

```
//stored in Compute.java
import java.util.*;
public class Compute {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int[] nums = int[10];
        for (int i = 0; i < 10; i++) {
            System.out.print("Enter a number: ");
            nums[i] = Integer.parseInt(s.nextLine());
        }

        System.out.println("Maximum: " + MathOps.max(nums));
        System.out.println("Minimum: " + MathOps.min(nums));
        System.out.println("Average: " + MathOps.avg(nums));
    }
}
```

Compiling and Running

Compiling works differently when we have multiple files, because we need to be sure to compile all the source code files instead of just the one with the `main` method. Here's how:

```
javac *.java
```

This compiles ALL files with the `.java` extension – all your source code files. You could then run the program with:

```
java Compute
```

which runs your program as usual. Note that you will run your program with the name of the file that contains the `main` method.

Objects

In this section, we will learn to make classes into our own specialized data types – complete with data fields and methods that operate on the data.

General Class Syntax

In the previous section, we looked at a specific kind of class – classes that have only static methods. However, very few classes in Java are written with entirely static methods. In this section, we will explore a more generic definition of a class. These generic classes in Java are made up of three parts:

- **Instance variables** – variables that can be used throughout the class
- **Constructor** – special method that initializes the instance variables
- **Methods** – perform operations on this object's data (more on this later)

Here is the format of a general class (note that our classes with static methods also fit this syntax – they are just a specific kind of class):

```
public class Name {
        //declare instance variables

        //constructor

        //methods
}
```

For now, each class should be stored in a separate file. The name of the file should match the name of the class (including capitalization), plus the "`.java`" extension. For example, the class above should be stored in the file `Name.java`.

Visibility Modifiers

The three parts of a class (instance variables, constructor, methods) should be preceded by a **visibility modifier**. This specifies where that variable, constructor, or method can be seen. There are three visibility modifiers:

- **public** – visible anywhere the class is visible
- **private** – visible only within the class
- **protected** – discussed in the Inheritance section

Instance Variables

Again, instance variables are special variables that can be used throughout the class. They are defined at the beginning of the class, using the following format:

```
visibility type name;
```

For example:

```
private int size;
```

Like typical variables, instance variables can also be initialized on the same line as their declaration. For example:

```
private int size = 0;
```

However, initialization of instance variables in usually done in the constructor.

Constructor

The constructor is a special method that initializes the instance variables in a class. It must have the same name as the class. It can either take values for the instance variables as **parameters**, or it can assign default values. Here's the format of a constructor:

```
public Name(params) {
      //initialize instance variables
}
```

Here, `Name` is the name of the class, and `params` are possible parameters to the constructor. Here is a sample class with a constructor:

```
public class Dog {
      //instance variables
      private String name;
      public String breed;

      //constructor
      public Dog() {
            name = "Fido";
            breed = "Mutt";
      }
}
```

When the constructor is called, the name and breed are set to default values. We could also accept initial values for the name and breed as parameters to the constructor:

```
public Dog(String n, String b) {
      name = n;
      breed = b;
}
```

We will look more at methods parameters in the next section. For now, remember that parameters must have a specified type and a name – just like in static methods.

Creating Objects

Think of Java classes as a more complicated variable type. For example, we wrote a `Dog` class, so now we can create variables of type `Dog`. These variables are similar to ordinary variables, but they each have their own methods and instance variables. These types of variables are called **objects**, whereas `int`s and `double`s are **primitive types**.

Syntax and Examples

Declaring an object variable has the same format of declaring a primitive variable:

```
type name;
```

For example, to declare a `Dog` variable:

```
Dog d1;
```

However, initializing an object is a little different. We must call the object's constructor to set up values for its instance variables. Here's the format of initializing an object:

```
name = new ClassName(args);
```

Here, `name` is the name of the object variable, `ClassName` is the name of the class, and `args` are the arguments passed to the constructor. Here's how we would create a new `Dog` object:

```
d1 = new Dog();
```

This calls the no-argument `Dog` constructor, and initializes the `Dog`'s name to "`Fido`" and its breed to "`Mutt`". Supppose we had used the second `Dog` constructor (with arguments for the name and breed), instead. Here's how we could create a second `Dog` object:

```
Dog d2 = new Dog("Rover", "Labrador");
```

Keep in mind that although the `d1` and `d2` variables are both of type `Dog`, and both have values for the `name` and `breed`, they are completely separate in memory. `Dog` is just a template that allows us to create variables.

Accessing Instance Variables

Once we have created an object, we can access any of its public members. In the `Dog` class, the `breed` variable is public, so we can access it. Here's the format for accessing public instance variables:

```
objectName.variableName
```

For example, here's how we would print the breed of both dogs:

```
System.out.println(d1.breed);
System.out.println(d2.breed);
```

This would print:

Mutt
Labrador

Non-Static Methods

Non-static methods in Java are fairly similar to the static methods we already know about. The difference is that non-static methods are associated with an object of a given class type, not with the class itself. This means that we can call a method for each of our object variables, and that the method will use that object's instance variables in its calculations. (This is not always true, but we'll pretend for now.) Here's the format of a method:

```
visibility returnType name(args) {
      //code
      //possible return statement
}
```

Let's talk about each part separately. The `visibility` can be either `public` or `private`, depending on whether we want to call the method outside this class. (It can later be `protected` as well.) The `returnType` specifies what type of value will be returned. If the method doesn't return anything, its return type is **`void`**. `name` is just the name of the method, which you will use when calling this method. `args` are optional *arguments* (parameters) for the method.

If a method has a non-`void` return type, it must have a **return statement**. This looks like:

```
return value/variable;
```

where that `value` or `variable` has the same type as `returnType`.

Here's a very simple method that just prints "`Hello, World!`" to the screen:

```
public void greeting() {
      System.out.println("Hello, World!");
}
```

Method Arguments

Method arguments (parameters) are just values passed to the method. Each method can have zero or many arguments. Here's what the argument list should look like:

```
type1 name1, type2 name2, ...
```

If a method has no arguments, you still need to include the `()` after the method name (like the `greeting` method above). Here's an example of a method that takes two ints, and prints their product to the screen:

```
public void printProduct(int num1, int num2) {
     int product = num1*num2;
     System.out.println("The product is " + product);
}
```

Returning a Value

So far, we've only looked at methods that have a `void` return type – that don't return anything. Here's a method that returns something:

```
public int product (int num1, int num2) {
     int product = num1*num2;
     return product;
}
```

This method has an `int` return type, and so it includes a `return` statement. The variable returned (`product`) also has type `int`.

Calling Methods (from the Same Class)

Now that we can write methods, we need to be able to call them. This will be different depending on whether we're calling a method that's in the same class as us, or in a different class. Here's how we call a method in the same class as us:

```
methodName(args)
```

Here, `methodName` is the name of the method we're calling, and `args` are the values of the parameters we're passing. The parameters must have the same type as the corresponding arguments in the methods definition. Here's an example of calling the `printProduct` method (from the same class):

```
int val1 = 5;
int val2 = 7;
printProduct(val1, val2);
```

The VALUES of `val1` and `val2` (5 and 7) are passed to `printProduct`, and stored in the arguments `num1` and `num2`.

When the method we're calling returns a value, we need to store that return value when we call it. Here's an example of calling the `product` method (from the same class):

```
int val1 = 5;
int val2 = 7;
int result = product(val1, val2);
```

Now the product of `val1` and `val2` (35) is stored in the `result` variable.

Notice that this syntax for calling methods is similar to what we've done to call static methods, but we do not need to include the name of the class.

Calling Methods (from a Different Class)

Calling methods from a different class works similarly, expect we first need an object of that class type. Let's look at an example class:

```
public class Rectangle {
     private int length;
     private int width;

     public Rectangle(int l, int w) {
          length = l;
          width = w;
     }

     public int area() {
          return length*width;
     }

     public int perimeter() {
          return length*2 + width*2;
     }
}
```

Before we can call any of the methods in `Rectangle`, we need to create a new `Rectangle` object. Let's create two – a 4x6 rectangle and a 3x5 rectangle:

```
Rectangle r1 = new Rectangle(4, 6);
Rectangle r2 = new Rectangle(3, 5);
```

Now, here's the format for calling a method in a different class:

```
objectName.methodName(args);
```

So, to call the area method on the 4x6 rectangle, we'd do:

```
r1.area();
```

However, the area method returns a value, so we probably want to store or do something with a result. We'll store the area of the 4x6 rectangle, and print the perimeter of the 3x5 rectangle:

```
int result = r1.area();
System.out.println("The perimeter is " + r2.perimeter());
```

Method Overloading

Method overloading is when you have two or more versions of the same method, and each version has a different argument list. For example, suppose you wanted to define a method that computed the max of two numbers. You might want to do this for both ints and doubles. Here's how:

```
public int max(int num1, int num2) {
     if (num1 > num2) return num1;
     else return num2;
}

public double max(double num1, double num2) {
     if (num1 > num2) return num1;
     else return num2;

}
```

Notice that the two versions of `max` have the same name, but different argument lists. They also have different return types, but overloaded methods do not have to work this way. The compiler can figure out which one you want to call based on what types you pass. For example:

```
max(4, 7)
```

would call the first version of `max` since the arguments are both ints. On the other hand,

```
max(5.4, 8.76)
```

would call the second version since the arguments are both doubles.

Constructors can also be overloaded. This is done by creating two constructors (both with the name of the class) with different argument lists.

The static Keyword

Class variables and **class methods** are defined with the **`static`** keyword. We have written static methods in the past without talking about how they are different from non-static methods.

Variables and methods in a class that are not static are instance variables and instance methods, which means there is a different version of each method and variable for each object instance of the class. Static variables and methods are called class variables and class methods, which means there is only one version of them for all different object instances of the class.

Static methods and variables can also be accessed with the class name (as we've done before) without having to create an object instance. Regular variables and methods cannot be accessed in this way.

Here's how to declare a static variable:

```
visibility static type name;
```

And here's how to declare a static method:

```
visibility static returnType name(args)
```

Here's a sample class with static variables and methods:

```
public class Circle {
     public static double pi = 3.14159;

     public static double area(double radius) {
          return pi*radius*radius;
     }
}
```

Now, because `pi` and `area` are static, **we access them using the class name (`Circle`)**. All of the following are valid:

```
System.out.println(Circle.pi);              //prints 3.14159
double area = Circle.area(2);               //area = 12.566

Circle c1 = new Circle();
Circle c2 = new Circle();

System.out.println(c1.pi);              //prints 3.14159
double area = c1.area(2);               //area = 12.566

Circle.pi = 3.14;       //Changes pi for all Circle objects
```

Static methods cannot refer to any instance variables. They can only refer to class variables (static variables), method arguments, and local variables.

The `final` Keyword

The keyword "`final`" in front of a variable denotes that the variable is a **constant**. Any variable declared with "`final`" cannot be modified once it is initialized. For example, the `pi` variable in the `Circle` class could be constant – `pi` should really always be the same value. Here's how we would change the class:

```
public class Circle {
     public static final double PI = 3.14159;

     public static double area(double radius) {
          return PI*radius*radius;
     }
}
```

Notice that constants are traditionally given names in all capitol letters. Also, if I try to change the value of `PI`, such as:

```
Circle.PI = 3.14;
```

I will get a compiler error.

Null

When we declare any variable without assigning it a value, that variable has **no value** by default. (The exception to this rule is arrays. When you create a space for an array, all the elements in that array are automatically initialized to the default value for that type. This means 0 for ints, false for bools, etc.) This means if we tried to do something like this:

```
int val;
val++;
```

We would get a compiler error, because we can't add one to a variable with no value. Similarly, if we did this:

```
Rectangle r;
int a = r.area();
```

We would also get a compiler error, because we are trying to use `r` without initializing it to be a new object.

Sometimes you may want to initialize a class type variable, even if you are not ready to create a new object yet. To do this, you can set the variable to the special value **`null`**:

```
Rectangle r = null;
```

The `null` value is a valid value for all non-primitive variables (all variables that have a class type). Primitives like ints, chars, doubles, and booleans cannot be set to `null`.

Arrays of Objects

We have already seen how to create arrays of things like ints and doubles, but we can also create arrays of objects, where each element has a class type. The format for declaring an array of objects is just like declaring any other array:

```
type[] name;
```

But here, `type` should be the name of a class. We can also create space for the array just like we've done before:

```
name = new type[size];
```

Elements in arrays are initialized to the default value of that type, so elements in an array of objects are automatically initialized to `null`.

To see an example, recall the `Rectangle` class from the Methods section. Suppose that we want to create an array of 10 rectangles whose values are inputted by the user. Then, we want to print the area and perimeter of each rectangle. Here's how:

```
//Declare the array and allocate space
Rectangle[] rectArray = new Rectangle[10];

Scanner s = new Scanner(System.in);

for (int i = 0; i < rectArray.length; i++) {
     System.out.print("Enter the length: ");
     int length = Integer.parseInt(s.nextLine());
     System.out.print("Enter the width: ");
     int width = Integer.parseInt(s.nextLine());

     //Create a new Rectangle object with the
     //correct dimensions, and store it in the array
     rectArray[i] = new Rectangle(length, width);
}

for (int i = 0; i < rectArray.length; i++) {
     //Print the area and perimeter for the current rectangle
     System.out.print("Rectangle " + i + ": ");
     System.out.print("area " + rectArray[i].area())
     System.out.println(", perim " + rectArray[i].perim());
}
```

Notice that we're treating each object in the array exactly like we would any other object, but instead of having to create a separate variable to refer to each object, we can store them all in an array.

Passing Objects to Methods

We can pass objects to methods just like any other type of element. If we pass an object to a method, the type for that parameter is the type of the object (the name of the class). Suppose we want to write a `printRectangle` method that takes a `Rectangle` object as a parameter and then prints its area and perimeter. Here's how the method would look:

```
public void printRectangle(Rectangle r) {
      //Now we can treat r just like any other Rectangle object
      System.out.println("Area: " + r.area());
      System.out.println("Perimeter: " + r.perim());
}
```

Now, suppose we're in the same class as the `printRectangle` method. Then we could do:

```
Rectangle rect = new Rectangle(3, 4);
printRectangle(rect);
```

Notice that passing objects to methods works exactly like passing other types to methods – the only difference is that the parameter type is now the name of a class.

Objects as Instance Variables

We can also make objects be instance variables in another class – again, this works just like other types of instance variables, but the type for the instance variable will be the name of a class. For example, suppose we have the following `Person` class:

```
public class Person {
      public String name;
      public int age;

      public Person(String n, int a) {
            name = n;
            age = a;
      }
}
```

Now, suppose we want to write a `Child` class that holds a child's name, grade, school, and parent information. Each parent will be stored as a `Person` instance variable. Here is the `Child` class:

```
public class Child {
```

```
        private String name;
        private int grade;
        private String school;
        private Person mother;
        private Person father;

        //n: name, g: grade, s: school
        //mn: Mom's name, ma: Mom's age
        //fn: Dad's name, fa : Dad's age
        public Child(String n, int g, String s, String mn,
                        int ma, String fn, in fa) {
            name = n;
            grade = g;
            school = s;

            //Initialize instance variables to be new objects
            //with corresponding names and ages
            mother = new Person(mn, ma);
            father = new Person(fn, fa);
        }
    }
```

Now, suppose we are outside both the `Person` and `Child` classes and we want to create a `Child` object with the following information:

- Name: Fred
- Grade: 5th
- School: Bluemont Elementary
- Mom: Donna, 34
- Dad: Frank, 35

Here's what we would do:

```
Child c = new Child("Fred", 5, "Bluemont Elementary",
                    "Donna", 34, "Frank", 35);
```

Call-by-Reference vs. Call-by-Value

There are two ways to call a method – calling by reference and calling by value. While these two approaches can sometimes look the same, passing an array or an object to a method is very different from passing a primitive variable like an int. If you modify an object or array that is a method argument, it will modify the original variable. However, if you modify a primitive method argument, the original variable remains unchanged.

Call-by-Reference

When you pass objects and arrays to methods, you are calling the method by reference. This means that if you change a value in the array, or change a property of the object, then the original

variable that you passed to the method will also change. (NOTE: there is a way to pass primitive values by reference, but we will not discuss this in class.)

Consider the following method:

```
public void setZero(int[] nums) {
     for (int i = 0; i < nums.length; i++) {
          nums[i] = 0;
     }
}
```

Now, consider this call to `setZero`:

```
int[] vals = {1, 2, 3, 4};
setZero(vals);
```

When we return from `setZero`, every element in the `vals` array will be 0. This is because if an array is changed by a method, the original array (`vals`) is also changed.

Passing Objects

Passing objects is also a type of call-by-reference. Consider the following class:

```
public class Person {
     private String name;
     private int age;

     public Person(String name, int age) {
          this.name = name;
          this.age = age;
     }

     public void incAge() {
          age++;
     }
}
```

And consider this outside method:

```
public void changePerson(Person p) {
     p.incAge();
}
```

Now we create a `Person` object and call the `changePerson` method:

```
Person pers = new Person("Amy", 26);
changePerson(p);
```

When we return from the `changePerson` call, `pers` now has age 27 (since changing the object in the method changed the original object).

However, suppose `changePerson` instead looked like this:

```
public void changePerson(Person p) {
     p = null;
}
```

If I now created a `Person` object and called `changePerson`:

```
Person pers = new Person("Amy", 26);
changePerson(p);
```

Then `pers` would NOT have the value `null` after the method call. This is rather confusing (and makes more sense in C++, which uses pointers), but `p` and `pers` are two different variables that reference the same object in memory. If I change that object's age (like in the first version of `changePerson`), then the age changes for BOTH variables. However, if I set `p` to `null`, it just makes p reference null instead of the `Person` object. `pers` still references the `Person` object, so it remains unchanged.

Call-by-Value

When you pass primitive variables (like ints) to a method, whatever changes you make inside the method will not affect the original variable. For example:

```
public void inc(int x) {
     x++;
}

int num = 4;
inc(num);
```

Even though the `inc` method adds one to `x`, it does not affect the value of `num`. This is because instead of passing the `num` variable, only the value is passed. This value (`4`) is stored in the method argument (`x`). When I increment `x`, it does not change `num` because they are two separate variables.

Just remember that if you pass an `int`, `double`, or `char` and change it inside a method – the change won't stick. If you change an object or array, the change will stick.

"This" Keyword

The keyword `this` refers to "this object instance". We can use it inside a class to refer to instance variables and methods in this class. This keyword is primarily used to distinguish instance variables from local variables. For example:

```
public class Person {
     private String name;
     private int age;

     public Person(String name, int age) {
          this.name = name;
          this.age = age;
     }
}
```

Inside the `Person` constructor, "`this.name`" refers to the instance variable called `name`, while "`name`" refers to the constructor argument called `name`. In general, you can say:

```
this.name
```

to refer to a method or variable (called `name`) inside THIS class. However, you cannot use the "`this`" keyword with static methods or variables, since they don't depend on a particular object instance.

Command-Line Arguments

Recall that the declaration of the main method must look like:

```
public static void main(String[] args)
```

The "`String[] args`" portion of the declaration is a String array of command-line arguments – values that can be supplied to the program when it is executed. You can then use these arguments as you would any other String array.

Providing Command-Line Arguments

If you are using Eclipse, then you will provide command-line arguments through it (in the Run dialog). This section will describe how to use command-line arguments when you run your program from the command-line.

Suppose the class with the main method is called `Test`. Then you would run your program from the command-line by typing:

```
java Test
```

To supply command-line arguments, simply type input values after "`java Test`" when you execute your program. Separate each argument with a space. Each argument will be places in

the `String[] args` array. Suppose you want to get a person's name, age, and GPA as command-line arguments. Then you might type:

```
java Test Bob 20 3.45
```

The values "`Bob`", "`20`", and "`3.45`" will be placed in the `args` array. "`Bob`" will be at index 0, "`20`" will be at index 1, and "`3.45`" will be at index 2.

Using Command-Line Arguments

We can access command-line arguments within the main method by saying `args[i]`, where `i` is the index of the argument. For example, to store the name, age, and GPA from above:

```
public static void main(String[] args) {
      String name = args[0];                    //name = "Bob"
      int age = Integer.parseInt(args[1]);      //age = 20
      double gpa = Double.parseDouble(args[2]); //gpa = 3.45
}
```

Of course, this program will crash if the user mistakenly doesn't enter any command-line arguments. To tell if they did, we can make sure args.length (the length of the command-line argument array) is what we expect. If not, we can print an error and exit. Here's how:

```
public static void main(String[] args) {
      if (args.length != 3) {
            //print a descriptive error message
            System.out.println("Usage: java Test name age gpa");
            System.exit(0);
      }

      //We know we have 3 command-line arguments
      String name = args[0];
      int age = Integer.parseInt(args[1]);
      double gpa = Double.parseDouble(args[2]);
}
```

Full Example

In this section, we will work a full example of a C# program that involves classes and objects. Suppose we want to write a program that stores a collection of bank accounts. Once we have the account information stored, we want to be able to look up and modify account information by name and account number. Consider the following full program (each class is assumed to be stored in a separate file):

```
public class Account {
      private int accountNum;
      private String name;
```

```
	private double balance;

	public Account(int num, String str, double b) {
		accountNum = num;
		name = str;
		balance = b;
	}

	public int getNum() {
		return num;
	}

	public String getName() {
		return name;
	}

	public double getBalance() {
		return balance;
	}

	public void deposit(double amount) {
		balance += amount;
	}

	public boolean withdrawal(double amount) {
		balance -= amount;
		if (amount < 0) return false;
		else return true;
	}

	public String toString() {
		return accountNum + " " + name + " $" + balance;
	}
}

public class AccountManager {
	private Account[] accounts;
	private int pos;

	public AccountManager(int size) {
		accounts = new Account[size];
		pos = 0;
	}

	public boolean addAccount(int n, String s, double b) {
		if (pos < accounts.length) {
			accounts[pos] = new Account(n,s,b);
			pos++;
			return true;
		}
		else return false;
	}

	public boolean deposit(String name, double amount) {
		Account a = lookup(name);
		if (a == null) return false;
		a.deposit(amount);
```

```
	}

	public boolean deposit(int num, double amount) {
		Account a = lookup(num);
		if (a == null) return false;
		a.deposit(amount);
	}

	public boolean withdrawal(String name, double amount) {
		Account a = lookup(name);
		if (a == null) return false;
		return a.withdrawal(amount);
	}

	public boolean withdrawal(int num, double amount) {
		Account a = lookup(num);
		if (a == null) return false;
		return a.withdrawal(amount);
	}

	public Account lookup(String name) {
		for (int i = 0; i < pos; i++) {
			if (accounts[i].getName().equals(name)) {
				return accounts[i];
			}
		}

		return null;
	}

	public Account lookup(int num) {
		for (int i = 0; i < pos; i++) {
			if (accounts[i].getNum() == num) {
				return accounts[i];
			}
		}

		return null;

	}
}

import java.util.*;
public class AccountTester {
	public static AccountManager manage;
	public static Scanner s;

	public static void main(String[] args) {
		s = new Scanner(System.in);

		//Assume we want a max of 10 accounts
		manage = new AccountManager(10);

		//repeatedly ask user to add account, lookup account,
		//or make a deposit or withdrawal
```

```
        char option;
        String prompt = "\nEnter (a)dd, (l)ookup, (d)eposit, " +
                        "(w)ithrawal, or (q)uit: ");
        do {
              System.out.print(prompt);
              option = (s.nextLine()).charAt(0);

              switch (option) {
                    case 'a':
                          add();
                          break;
                    case 'l':
                          lookup();
                          break;
                    case 'd':
                          deposit();
                          break;
                    case 'w':
                          withdrawal();
                          break;
                    case 'q':
                          break;
                    case 'default':
                          System.out.println("\nInvalid option.");
                          break;
              }
        } while (option != 'q');
  }

  public static void add() {
        System.out.print("Enter name: ");
        String str = s.nextLine();
        System.out.print("Enter account number: ");
        int num = Integer.parseInt(s.nextLine());
        System.out.print("Enter balance: ");
        double b = Double.parseDouble(s.nextLine());

        boolean result = manage.addAccount(str, num, b);
        if (result == false) {
              System.out.println("\nAccount Manager is full.");
        }
  }

  public static void lookup() {
        System.out.print("Enter lookup by (n)ame or (a)ccount number: ");
        char choice = (s.nextLine()).charAt(0);

        switch (choice) {
              case 'n':
                    System.out.print("Enter name: ");
                    String str = s.nextLine();
                    Account a = manage.lookup(str);
                    if (a == null) {
                          System.out.println("\nNo such account.");
                    }
                    else System.out.println(a.toString());
                    break;
```

```
            case 'a':
                  System.out.print("Enter account number: ");
                  int num = Integer.parseInt(s.nextLine());
                  Account a = manage.lookup(num);
                  if (a == null) {
                        System.out.println("\nNo such account.");
                  }
                  else System.out.println(a.toString());
                  break;
            default:
                  System.out.println("\nInvalid option.");
      }
}

public static void deposit() {
      System.out.print("Deposit by (n)ame or (a)ccount number? ");
      char choice = (s.nextLine()).charAt(0);

      switch (choice) {
            case 'n':
                  System.out.print("Enter name: ");
                  String str = s.nextLine();
                  System.out.print("Enter amount: ");
                  double b = Double.parseDouble(s.nextLine());
                  boolean result = manage.deposit(str, b);
                  if (result == false) {
                        System.out.println("\nDeposit failed.");
                  }
                  break;
            case 'a':
                  System.out.print("Enter account number: ");
                  int num = Integer.parseInt(s.nextLine());
                  System.out.print("Enter amount: ");
                  b = Double.parseDouble(s.nextLine());
                  boolean result = manage.deposit(num, b);
                  if (result == false) {
                        System.out.println("\nDeposit failed.");
                  }
                  break;
            default:
                  System.out.println("\nInvalid option.");
      }
}

public static void withdrawal() {
      System.out.print("Withdrawal by (n)ame or (a)ccount number? ");
      char choice = (s.nextLine()).charAt(0);

      switch (choice) {
            case 'n':
                  System.out.print("Enter name: ");
                  String str = s.nextLine();
                  System.out.print("Enter amount: ");
                  double b = Double.parseDouble(s.nextLine());
                  boolean result = manage.withdrawal(str, b);
                  if (result == false) {
                        System.out.println("\nWithdrawal failed.");
```

```
                        }
                        break;
                  case 'a':
                        System.out.print("Enter account number: ");
                        int num = Integer.parseInt(s.nextLine());
                        System.out.print("Enter amount: ");
                        b = Double.parseDouble(s.nextLine());
                        boolean result = manage.withdrawal (num, b);
                        if (result == false) {
                              System.out.println("\nWithdrawal failed.");
                        }
                        break;
                  default:
                        System.out.println("\nInvalid option.");
            }
      }
}
```