Code	Comments
Student st;	
Undergrad ug;	
st = new Undergrad("joe", "23");	
ug = st;	ERROR (at compile time)
ug = (Undergrad) st;	cast down to Undergrad
st = new GradStudent("jim","32");	
ug = (Undergrad) st;	ERROR (at run time)

Figure 6.8: Assignment to super-class variable, with a cast

Here we should be able to store the reference, `stud1` , into the variable `younger` because the dynamic type, `Undergrad` matches the static type of `younger`, but the compiler will not accept it:

```
younger = stud1; // ERROR
```

The problem is the compiler is not convinced (and not smart enough to know) that the dynamic type of stud1 is `Undergrad` (recall that dynamic type is determined at execution time). We need to convince the compiler that everything will be okay at execution time; this is done with a *cast*. A cast forces the type of a reference to a particular type. We have already seen casts with respect to primitive types; they can also be used with reference types. To apply a cast to an expression, simply pubt the casting type in parentheses and preceding the expression. The format is:

(type) expression

For example, `(Undergrad) stud1;` produces a reference to an Undergrad, which can now be assigned to a variable whose static type is Undergrad:

```
younger = (Undergrad) stud1; // OK
```

However, if the the dynamic type of stud1 is not truly `Undergrad` as we are claiming, we will get a runtime error when the cast is applied. Figure 6.8 shows another code segment, similar to Figure 6.7, in which we show some valid and non-valid operations.

When considering the use of a cast, remember that you are always casting *down* the class diagram, from a superclass to a subclass. `(Undergrad) student` is fine, but `(Student) undergrad` is not permitted.

6.3.3.3 Importance of inheritance

Inheritance is *extremely* important in Java programming. Looking at the Java class library, you will see inheritance *everywhere*. Moreover, we can extend classes from the Java class library to form extensions. For example, we might want an improved version of the ArrayList class which can tell us whether the items are in ascending order. This can be done easily by extending ArrayList with a subclass that has the desired method:

```
public class BetterArrayList extends ArrayList
{
```

```
public boolean isAscending()
{  ///  method body here  }
}
```

6.3.4 Exercises

1. Fill in the missing entries in the table below with one of the following:

   - Is-a
   - Has-a
   - Neither
   - Both

Car	Has-a	Engine
SUV		Car
Student		Person
Person		SSN
Petunia		Plant
Petunia		Flower
Petunia		Stem
Student		University
University		Collection of Students
Windows Folder		Collection of Documents and Folders
Java method		Signature

2. Assume `Person` and `Animal` have been defined as classes. Which of the following contain syntax errors?

   (a)
   ```
   public class Student subclass Person
   {  }
   ```

   (b)
   ```
   public class Student extends Person
   {   }
   ```

   (c)
   ```
   public class Student extends Person, Animal
   {   }
   ```

3. Refer to the classes `Student` and `GradStudent` described in this section. Define classes named `PhdStudent` and `MastersStudent`. Both should be sublcasses of `GradStudent`. A PhdStudent should have two fields:

   - A boolean field, true only if the PhdStudent has passed qualifying exams.
   - A String field, storing the PhdStudent's dissertation topic, or null if the PhdStudent has no dissertaion topic.

Newly created PhdStudents have not passed qualifying exams and have no dissertation topic.

A MastersStudent has one field, a boolean which is true if the MastersStudent is on a thesis track. When a MastersStudent object is created, it should be possible to specify whether the MastersStudent is on a thesis track.

Include appropriate public accessor and mutator methods in these classes.

4. Assume we have defined a class named `Vehicle` which has two subclasses named `Bicycle` and `Car`. These classes all have default constructors. Which of the following contain syntax errors, and which contain run-time errors?

   (a)
   ```
   Vehicle v;
   Bicycle b;
   v = new Bicycle();
   ```

   (b)
   ```
   Vehicle v;
   Bicycle b;
   b = new Vehicle();
   ```

   (c)
   ```
   Vehicle v;
   Bicycle b;
   v = new Bicycle();
   b = v;
   ```

   (d)
   ```
   Vehicle v;
   Bicycle b;
   v = new Bicycle();
   b = (Bicycle) v;
   ```

   (e)
   ```
   Vehicle v;
   Bicycle b;
   Car c;
   v = new Bicycle();
   c = (Car) v;
   ```

5. Look at the API for the `ArrayList` class in the `java.util` package.

   (a) What is the superclass of `ArrayList`?

   (b) What are the (direct) subclasses of `ArrayList`?

6.4 Polymorphism and dynamic method look-up

In this section we will examine how dynamic type is used in method calls. We will also see that objects can exhibit different behavior, depending on their dynamic types. The word *polymorphism*, in the natural sciences, means 'taking on different forms or appearances.' In object-oriented programming polymorphism is exhibited when two identical method calls can result in the invocation of different methods. This can be particularly useful when we have a collection of objects, with different dynamic types, and the action we wish to perform on each of those objects will depend on its dynamic type.

6.4.1 Dynamic method look-up

Before going into the details of polymorphism, we need to take a closer look at the mechanism which is used when methods are called. Continuing with our Student classes, as shown in Figure 6.3, assume we wish to print the name of a particular student:

```
Student st;
st = new UnderGrad("jim", "22");
System.out.println ("The student's name is " + st.getName());
```

The method call `st.getName()` means to apply the `getName()` method to the `st` object. However, `st` refers to an object of type `Undergrad` and that class has no `getName()` method. We need to understand that methods are invoked by a process known as *dynamic method look-up*. When a method is called via `object.method()`:

- Look in the class of the given object's dynamic type. If the given method is there, that is the method which is invoked.
- If the method is not there, look in the superclass of the given object. If the method is found there, that is the method which is invoked.
- If the method is not there, continue to look in the super-super-superclass.
- Ultimately the method will be found, or the Object class will be reached (Object is always at the root of the class hierarchy). If the method is not found anywhere on the path to Object, an error is produced (as we will see later, this could be a compile-time error or a run-time error).

In the code segment shown above the call `st.getName()` will find no such method in the Undergrad class, so it will look in the superclass, `Student` and find the method to be invoked in that class.

6.4.2 Polymorphism

Having defined dynamic method look-up, we are now in a position to understand polymorphism in object-oriented programming. Consider the case where we wish to invoke a method such as calcGPA on a variable whose static type is Student:

```
Student st = new Undergrad("jim", "22");
int cr = readCredits(st);
int gp = readGradePoints(st);
st.calcGPA(gp,cr);
```

In this case the compiler will issue an error on the call to calcGPA because there is no such method in the Student class. Unfortunately we know how to calculate a GPA only for GradStudents and Undergrads, but not for ordinary Students. To remedy this we can include a method in the Student class as a place-holder, simply to satisfy the compiler, as long as we are sure that it never actually gets called (in the next section we'll see a better way to handle this). In the `Student` class:

```
/** This method should never be invoked.  It is here only as
 *  a place-holder, so that calls to calcGPA() can be compiled
 */
public void calcGPA(int gradePoints, int credits)
{ // do nothing
}
```

Now the method call `st.calcGPA(int,int)` will work fine; it will call the `calcGPA(int,int)` method in the `Undergrad` class because the dynamic type of `st` is `Undergrad`. To further explain polymorphism, we could extend that code segment with two more statements:

```
Student st = new Undergrad("jim", "22");
int cr = readCredits(st);
int gp = readGradePoints(st);
st.calcGPA(gp,cr);
st = new GradStudent ("mary", "32);
cr = readCredits(st);
gp = readGradePoints(st);
st.calcGPA(gp,cr);
```

The first call `st.calcGPA(gp,cr)` will invoke the `calcGPA(int,int)` method in the `Undergrad` class, and the second call `st.calcGPA(cr,gp)` will invoke the `calcGPA(int,int)` method in the `GradStudent` class. Two identical statements result in different methods being invoked. This is polymorphism and is depicted in Fig 6.9

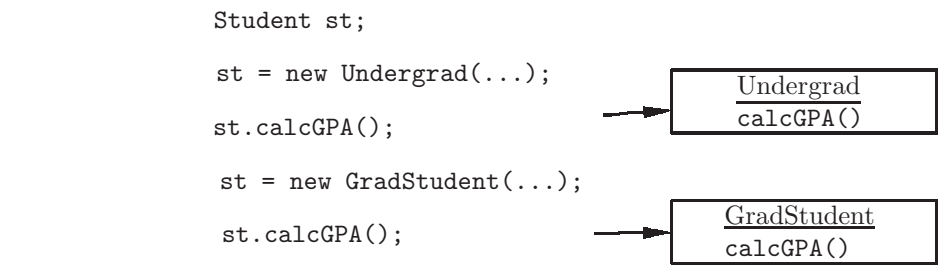


Figure 6.9: Polymorphism: Identical method calls invoke methods in different classes

6.4.2.1 Polymorphism with collections

Polymorphism is most often used in connection with collections. For example, assume that `roster` stores a reference to a list of students:

```
List <Student> roster;
roster = new ArrayList <Student>();
roster.add (new UnderGrad ("jim", "22");
roster.add (new GradStudent ("mary", "32");
roster.add (new UnderGrad ("joe", "56");
```

We now wish to print the name and GPA of each Student in the list. Before printing a student's gpa, we will make sure it has been calculated. Polymorphism handles this perfectly:

```
for (Student st : roster)
  {  int cr = readCredits(st);
     int gp = readGradePoints(st);
     st.calcGPA(gp,cr);
     System.out.println ("The GPA for " + st.getName() + " is " +
                         st.getGPA();
  }
```

The method call to `calcGPA(gp,cr)` will invoke the appropriate method using dynamic method look-up.

6.4.3 Exercises

1. Assume that we have a class named `Vehicle` with two subclasses: `Car` and `Bicycle`. These classes all have default constructors. The subclasses each has a method named `getMPG()`.

   In the Car class:

   ```
   public double getMPG()
   {  return 35.0;  }
   ```

In the Bicycle class:

```
public double getMPG()
{  return 0.0;  }
```

(a) Which line(s) shown below will cause syntax error(s)?

```
Bicycle b = new Bicycle();
Car c = new Car();
Vehicle v = new Bicycle();
System.out.println (b.getMPG());
System.out.println (c.getMPG());
System.out.println (v.getMPG());
```

(b) Show change(s) to any of these classes which will prevent the syntax error(s) from the previous problem.

(c) In the previous problem, the method call `v.getMPG()` will result in a call to the `getMPG()` method in which class (after the correction has been made)?

(d) Define a method named `getMPG()` with one parameter, a List of Vehicles, which will return the average MPG those Vehicles:

```
/** @return the average MPG of the given vehicles, or 0 if the List
 *  is empty
 */
public double averageMPG (List <Vehicle> vehicles)
```

2. This exercise refers to dynamic method lookup. Figure 6.10 depicts a class diagram showing public void methods that are defined in each class. Notice that the same method signature can occur in several different classes. Assume the following declarations:

```
Class2 c2;
Class3 c3;
Class4 c4;
Class5 c5;
```

Assume the variables declared above have been assigned non-null values. In each of the following show which method is invoked, by giving the name of its class, or indicate that an error will occur.

(a) `c4.method1();`

(b) `c4.method2();`

(c) `c5.method1();`

(d) `c2.method2();`

(e) `c2.method1();`

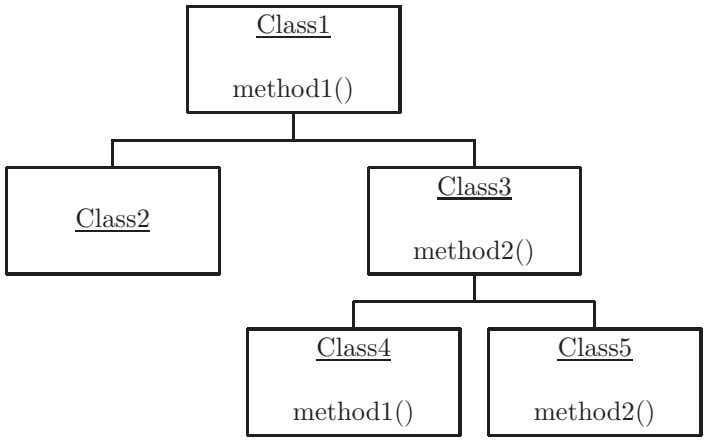


Figure 6.10: Class diagram for exercise on dynamic method lookup. Each method is public void.

6.5 Overriding methods from the Object class

Recall that the `Object` class is (directly or indirectly) a superclass of every Java class. The `Object` class is in the package `java.lang` and does not need to be imported. Looking at the API for the `Object` class we see at least three interesting methods:

- `public String toString()` - An object of this class can be represented by a String.
- `public boolean equals (Object)` - An object of this class can be compared for equality with any other object (more on this in chapter 7).
- `public int hashCode()` - An object of this class can produce an int likely to be unique for unequal objects of the same class (more on this in chapter 8).

6.5.1 Overriding the `toString()` method

The purpose of the `toString()` method is to produce as a result a String representation of an object. This will be useful when one needs to display data for a user; the `toString()` method should format the data to be readable and clear to the user. It should return that formatted result as a single String (it may contain newline characters). As an example, we could override the `toString()` method in our Student class as shown below:

```
/** @return this Student as a String */
public String toString()
{   String result = "Name: " + name + "\n";
```

```
    result += "SSN:  " + ssn + "\n"; // concatenate ssn
    result += "GPA:  " + gpa + "\n"; // concatenate gpa
    return result;
}
```

The following code

```
Student s1 = new Student ("Joe", "123-45-6789");
System.out.println (s1.toString());
```

would produce the following output:

```
Name: joe
SSN:  123-45-6789
GPA:  0.0
```

The `toString()` method becomes even more useful when we learn that it is called automatically by the Java runtime environment when:

- An object which is not a String is concatenated with a String. The toString() method produces a String for the concatenation.
- An object is passed as a parameter to the System.out.println method since println is expecting its parameter to be a String.

This means that we can concatenate a String with a Student object:

```
"Best student is " + s1
```

and we can simplify the call to println:

```
System.out.println (s1);
```

without explicitly calling toString().

Most of the classes in the Java class library override the `toString()` method. This means that you can print objects of those classes easily, and expect to get a pretty good looking result. Even Collection classes such as ArrayList have a toString() method. In the case of an ArrayList the toString() method will produce a result consisting of:

1. An open bracket - [
2. String representations of all the elements in the ArrayList (these are produced by calling toString() on each element), separated by commas
3. A close bracket -]

An ArrayList of 3 Strings might appear like this:

```
["joe","jim","mary"]
```

Incidentally, primitives can also be converted to Strings with concatenation. If the variable `sum` is an int, with value 23, the value of `"Sum is " + sum` is the String `"Sum is 23"`. However, Figure 6.11 shows instances where concatenation might produce unexpected results. To understand Figure 6.11 recall that when there are several + operators in an expression, they are executed left to right. To create a String representation of a primitive, simply concatenate it with a String of length 0:

`17 + ""` produces the String `"17"`.

Plus operation(s)	Result	Explanation
`2 + 3`	`5`	Addition of ints
`"2" + "3"`	`"23"`	Concatenation of Strings
`2 + "3"`	`"23"`	Concatenation of Strings
`"2" + 3`	`"23"`	Concatenation of Strings
`2 + 3 + " is the result"`	`"5 is the result"`	Addition done first
`"Result is " + 2 + 3`	`"Result is 23"`	Concatenation done first
`"Result is " + (2 + 3)`	`"Result is 5"`	Parentheses take precedence

Figure 6.11: The overloaded + operator can mean addition of numbers or concatenation of Strings, depending on the context.

6.5.1.1 Failing to override toString()

What would have happened in the prior examples if we had not included a `toString()` method in our Student class? The compiler will allow a call to `s1.toString()` because the `toString()` method is defined in a superclass (Object). Then at runtime when `toString()` is called, dynamic method lookup tells us that it will search for this method in superclass(es), until it is found. In this case it will be found in the `Object` class. A quick look at the API for `Object` shows that `toString()` will call `hashCode()` (see below) which returns an int. This int is formatted in hexadecimal (base 16), concatenated with the name of the class, and returned as a String. This is most likely not what you wish to happen. In summary, if you are printing an object, and you see some strange looking output, such as `Student@49a3c30`, you need to define a `toString()` method in the Student class.

6.5.2 Exercises

1. Consider the following class

```
public class Vehicle
{   int wheels;

    public Vehicle (int wheels)
    {  this.wheels = wheels;  }
}
```

   In some other class define a method in which you have the following:

```
Vehicle v1 = new Vehicle(18);     // semi
System.out.println (v1);
```

   If the output makes no sense, fix the Vehicle class so that the output will be more readable, such as:

```
Vehicle with 18 wheels
```

2. Test your solution to the previous problem by creating a List of at least 3 Vehicles, all with different number of wheels. Print the list without using a loop.

3. Show the output in each case:

   (a) `System.out.println (" 5 + 4 + is " + 5 + 4) ;`

   (b) `System.out.println ( 5 + 4 + " is " + 5 + 4) ;`

   (c) `System.out.println (" 5 * 4 + is " + 5 * 4) ;`

   (d) `System.out.println (" 5 + 4 + is " + (5 + 4)) ;`

6.6 Abstract methods and classes

6.6.1 Abstract methods

We now return to the definition of the `calcGPA(int,int)` method in the Student class. Recall that it was included simply as a 'place-holder', to satisfy the compiler; we expect that it will never be invoked:

```
public void calcGPA(int gradePoints, int credits)
{          // do nothing
}
```

If it seems strange to you that there should be a method which does nothing at all, you are not alone. This occurs so often that Java has a designation for this kind of method called *abstract*. A method which exists in a superclass merely to support the existence of methods having the same name in subclasses should be declared as `abstract`:

```
public abstract void calcGPA(int gradePoints, int credits);
```

Note that instead of a method body, there is a single semicolon; abstract methods have no body, and need no body, because they are never invoked. With this slight change our Student classes should work just as well.

Remember the following concerning abstract methods:

- Used as a place-holder for methods of the same name in subclasses
- Declared with the `abstract` keyword in the signature
- Semicolon after the parameter list
- No method body, not even the curly braces
- Must be implemented in subclasses
- May be used only in abstract classes (see next section)

6.6.2 Abstract classes

Any class which has at least one abstract method must be declared as an *abstract class*. This is done with the keyword `abstract` in the declaration of the class:

```
public abstract class Student
```

Earlier we said that we were not sure that it would make sense to instantiate a Student, not knowing what kind of Student he/she is. If this truly is not desirable, an abstract class is exactly what we need. When a class is abstract it cannot be instantiated:

```
new Student("jim", "33"); // ERROR
```

Thus by making the Student class abstract, we ensure that no client will ever be able to instantiate a Student, but will be able to instantiate GradStudent and Undergrad because they are *concrete* classes (i.e. not abstract).

When using abstract classes which may have one or more abstract methods, we must be sure that the methods are implementd in subclasses. If an abstract method were not implemented in one of the subclasses, a method call to an object of that subclass would have no method to invoke. For example, if the `Undergrad` class had no `calcGPA(int,int)` method, then dynamic method look-up would fail for a call to `st.calcGPA(gp,cr)` in the case that the dynamic type of `st` is `Undergrad`.

Remember the following about abstract classes:

- Declared with `abstract` keyword at the top
- May have one or more abstract methods
- Cannot be instantiated
- Abstract methods must be concrete (not abstract) in subclasses

A subclass can also be abstract, in which case it would not be required to implement abstract methods inherited from a superclass, but *its* (concrete) subclasses would be required to implement the abstract methods. In other words, viewing the class diagram from top to bottom, all abstract methods must be implemented somewhere on a path from the top to a concrete class, as depicted in Figure 6.12.

Notice in Figure 6.12 that the method `meth1` is implemented (i.e. concrete) in all concrete subclasses, but method `meth2` need not be implemented in class `Sub3` nor in class `Sub4` because it is implemented in class `Sub2` and is therefore available in classes `Sub3` and `Sub4`.

6.6.3 Exercises

1. Point out the syntax error, if any, in each of the following:

   (a)
   ```
   public class Class1
   {  public abstract void method1();  }
   ```

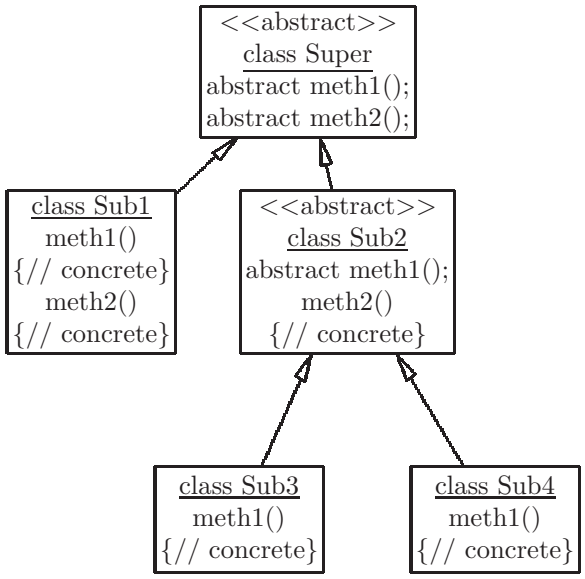


Figure 6.12: Class diagram showing an abstract method which must be implemented (concrete) for use in concrete sub-classes

(b)
```
public abstract class Class1
{  public void method1()
   {     }  // do nothing
}
```

(c)
```
public abstract class Class1
{  public abstract void method1();  }

public class Class2 extends Class1
{  int field1;
   public void method2()
   {   }    // do nothing
}
```

(d)
```
public abstract class Class1
{  public void method1()
   { Class1 c1 = new Class1();    }
}
```

2. Use the Student, GradStudent, and UnderGrad classes from the project university-ch6 in the code repository. We wish to calculate the GPA for a list of Students:

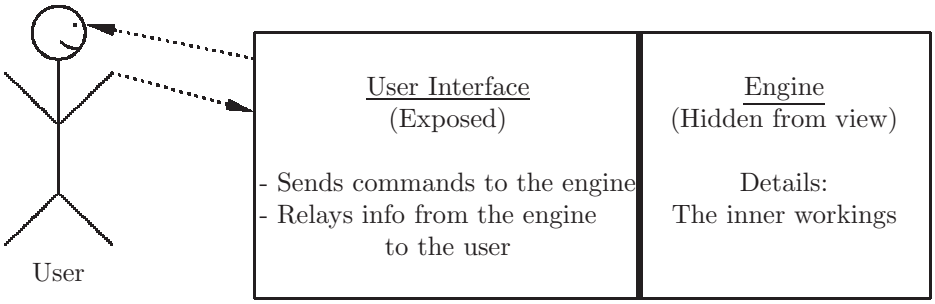


Figure 6.13: User Interface: All communication with the engine is through the user interface

```
List <Student> roster = new ArrayList <Student>();
//  several students added to roster
for (Student st : roster)
    st.calcGPA(17,3);
```

Make the necessary changes to the Student class to get this code to compile and execute. Assume that Student will never be instantiated.

6.7 Java Interfaces

There are at least three different, but related, meanings of the word *interface* in computer science:

- A user interface is that which stands between an entity and the user of that entity. All communication with the entity generally goes through the interface (in both directions) as shown in Figure 6.13. We'll call the entity being used the *engine*. User interfaces generally simplify usage of the engine for the user and can provide the user with useful information about the state of the engine while hiding unnecessary details of the internal workings of the engine.

  Examples of user interfaces include:

  - The API for a class constitutes an interface showing a potential user how the class can be used.
  - The API and signature for a method constitutes an interface showing a potential user how the method can be used.
  - The command language for an operating system such as DOS or Unix constitutes an interface between the user and the services available in the operating system.
  - An automobile's dashboard, gear stick, foot pedals, etc. constitute an interface with the engine of an automobile.

Many feel that user interfaces should be standardized, rather than proprietary (owned by a single company). What would happen if an automobile manufacturer designed a car in which the brake pedal was on the right, and the accelerator pedal was on the left?

- A *graphical user interface* (GUI) generally refers to software which allows the user to communicate with a program while it is executing using a graphic images (icons, trash basket, folders, etc) and some sort of pointing device such as a mouse. The first such GUI was the desktop GUI provided on the early Apple Macintosh computers (actually derived from a system developed at Xerox PARC). The desktop metaphor provided users with a means of coommunication with the operating system which was fairly intuitive and easy for the user. Today most software applications provide a GUI for the user.

  Related to the question of standardization of user interfaces, when Microsoft followed Apple's strategy by introducing a GUI for the PC – Windows, Apple filed a copyright infringement lawsuit, claiming that Microsoft had stolen the *look and feel* of their GUI (ironically, Apple had taken the idea from Xerox years earlier).

- A *Java interface* is similar to an abstract class which has no fields and no concrete methods. In this section we will provide some motivation for, and examples of, Java interfaces.

6.7.1 The need for Java interfaces – multiple inheritance

In this section we provide some motivation for the need for Java interfaces, but first we should discuss *multiple inheritance*.

We have said that a Java class may not have more than one superclass, but some progamming languages will actually permit this so-called multiple inheritance. To motivate this discussion, we return to our example involving the classes Student, Undergrad, and GradStudent. We now add two more classes to this project: `Prof` and `Instructor`. A Prof is a member of the faculty whose job it is to teach classes and conduct research. An Instructor is anyone who teaches classes. Often at a research university, grad students are asked to teach classes. This would mean that a GradStudent *is-a* Student, and a GradStudent *is-an* Instructor. As we saw previously, the is-a relationship implies the need for inheritance, as shown in Figure 6.14. We now have a problem because Java will not allow multiple inheritance:

```
public class GradStudent extends Student, Instructor // ERROR
```

A discussion of the pros and cons of allowing multiple inheritance in a programming language is beyond the scope of this book; suffice it to say that multiple inheritance can complicate things for both the programmer and the compiler writer.

Java provides a good solution to this problem: the *Java interface*. A Java interface is similar to an abstract class which has no fields and in which all the

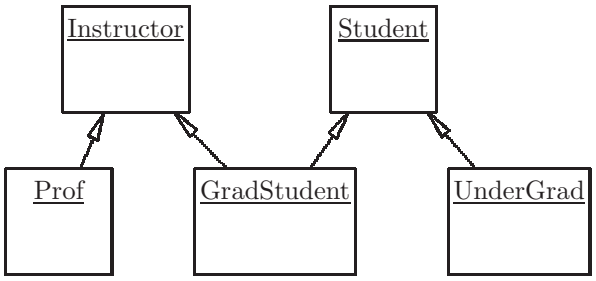


Figure 6.14: Class diagram showing multiple inheritance for the GradStudent class; this is not permitted in Java

methods are abstract. An interface is declared with the keyword `interface` instead of `class`. It is basically a template for subclasses, showing all the methods which must be implemented:

```
public interface Instructor
{   public abstract List <Course> getCourses();
    public abstract String getName();
}
```

This is an interface with 2 methods, both abstract. Note the following concerning Java interfaces:

- An interface must not contain any instance variables (i.e. non-static fields).
- An interface must not contain a constructor.
- All methods in an interface must be public abstract. If not declared as such, the compiler will assume they are public abstract.
- An interface, like an abstract class, must not be instantiated.

We can redefine our interface more briefly as:

```
public interface Instructor
{   List <Course> getCourses();
    String getName();
}
```

Now we can deal with the problem of multiple inheritance. To specify a subclass relationship with an interface, we say that a class *implements* the interface:

```
public class GradStudent extends Student implements Instructor
```

Our class diagram can now be drawn as shown in Figure 6.15 It shows that every GradStudent *is-a* Student and every GradStudent *is-an* Instructor.

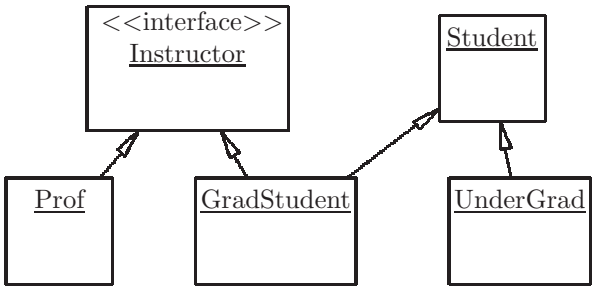


Figure 6.15: Class diagram showing multiple inheritance with an interface: Instructor

The compiler is satisfied because GradStudent extends only one class: Student. Instructor is an interface, not a class.

Since the class GradStudent implements the Instructor interface, GradStudent will have to implement all the methods in that interface: `getCourses()` and `getName()`.

A class may implement more than one interface; they are listed in the declaration separated by commas. For example if there was another interface called `Researcher`, we could define the GradStudent class as shown below:

```
public class GradStudent extends Student implements Instructor,
Researcher
```

6.7.2 Interfaces which we've already been using

Interfaces are very common in the Java class library; we've already started using them: List and Set.

If you look at the API in the Java.util package, you'll notice that:

- Interfaces and classes are separated.
- List is an interface; ArrayList implements List. Every ArrayList *is-a* List. There are other classes which implement List, which we have not yet used.
- Set is an interface; HashSet implements Set. Every HashSet *is-a* Set. There are other classes which implement Set, which we have not yet used.

When we declare a list:
`List <Student> roster;`
we are saying that the variable `roster` may store a reference to any kind of list. When we instantiate the list:
`roster = new ArrayList<Student> ();`
we determine the specific kind of list – ArrayList.

This points out another advantage of interfaces which you will learn if and when you study Object-oriented Design: It is better to program to an interface rather than an implementation whenever possible.

While we are on the subject of interfaces in the Java class Library, take a look at Iterator; you'll see that it is an interface, not a class. That's ok because nowhere do we instantiate Iterator - we always obtain an instance from a collection, using the method `iterator()`. The particular kind of Iterator obtained is determined by the collection, but that is of little concern to us; we just use it as some kind of Iterator.

6.7.3 Exercises

1. Point out the syntax error, if any, in each of the following (refer to Figure 6.15):

(a)
```
public class SeniorStudent
             extends GradStudent, Undergrad
{
  // 5 year BSMS program
  private boolean fiveYear;

    public SeniorStudent (String name, String ssn,
                          boolean fiveYear)
    {  super (name, ssn);
       this.fiveYear = fiveYear;
    }
}
```

(b)
```
public interface Administrator
{  private int salaryLevel;
   public abstract int getSalaryLevel();
}
```

(c)
```
public interface Administrator
{
   public Administrator (String name, String ssn);
   int getSalaryLevel();
}
```

(d)
```
public interface Administrator
{
   int getSalaryLevel()
   {   return 17;  }
}
```

(e)
```
public interface Administrator
{
```

```
   int getSalaryLevel();
}
```

(f) The following code is included in a method in some other class of the same project:

```
Student junior = new Student ("jim", "222");
```

(g)
```
public class MyList extends List
{  private boolean isSorted;    // true only if the elements are
                                // in increasing order.

   /** @return true only if the elements of this MyList
    *  are in increasing order
    */
   public boolean getSorted()
   {  return isSorted;  }
}
```

2. Show the class diagram, similar to Figure 6.15, which would result from the following class and interface declarations (the fields and methods of each are not shown). Be sure to designate interfaces as such to distinguish them from classes.

```
public class Mammal
{  ... }

public class Fish
{  ... }

public interface Swimmer
{  ... }

public class Whale extends Mammal implements Swimmer
{  ... }

public class Guppy extends Fish implements Swimmer
{  ... }
```

3. Show the class and interface declarations (no need to show any fields nor methods) corresponding to the class diagram shown in Figure 6.16.

4. Arrange the following concepts into an appropriate class (or interface) hierarchy, and show the class or interface declarations and the class diagram,

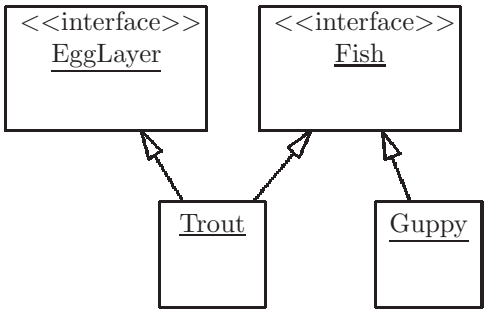


Figure 6.16: Class diagram showing multiple inheritance with an interface for exercise set 6.7

as shown in the previous two exercises. Each interface must have at least one implementing class.

`Bear, Fish, Mammal, DangerousAnimal, Shark`

6.8 Inheritance and Polymorphism in the GridWorld case study

6.9 Projects

1. Use the project `dome` from the code repository. The word 'dome' as used here is an acronym for 'database of multimedia entertainment'. In this project we allow the user to maintain information on a collection of CD's, DVD's, etc. In the project that we start with there are three classes:

   - `Database` – This class stores a List of CDs (Compact Disks) and a List of DVDs (Digital Video Disks). There are methods which allow the user to:
     - Add a CD to the List of CDs
     - Add a DVD to the List of DVDs
     - Print all the CDs and DVDs currently stored
   - `CD` – This class stores information for one CD:
     - The title of the CD (String)
     - The performing artist on the CD (String)
     - The number of tracks on the CD (int)
     - The playing time of the CD (int)
     - Whether we currently own this CD (boolean)
     - A comment providing other information on this CD (String)

     There are accessor and mutator methods for some of the above fields, and a method which will print the CD.

- `DVD` – This class stores information for one DVD:
  - The title of the DVD (String)
  - The director of the movie on the DVD (String)
  - The playing time of the DVD (int)
  - Whether we currently own this DVD (boolean)
  - A comment providing other information on this DVD (String)

  There are accessor and mutator methods for some of the above fields, and a method which will print the DVD.

We wish to make some improvements, in stages, to this project.

(a) Make a list of fields and methods which the CD and DVD classes have in common. Factor out these fields and methods to a superclass called `Item`. You will need some accessor methods in the Item class. Be sure to include a constructor in the Item class, and make the appropriate changes to the constructors in the CD and DVD classes. Caution: The print methods in CD and DVD are different, so don't factor them yet.

(b) Change the Database class so that instead of storing two Lists, it stores one List of Items. The Database class will have a method to add any Item to the database. The Item class will need a print() method; for now, give it a print() method with an empty body. Test your work by creating a Database object and adding several CDs and DVDs to it. Then invoke the list() method to print everything in the database.

(c) The print() method in CD and DVD have some duplicated code which needs to be factored to the Item class. The print() method in each class should print only the fields in that class. The print() method in the Item class can be invoked

(d)

2. This project involves the simulation of traffic in a city. We wish to simulate the activities of vehicles, such as busses, taxis, and cars. We also wish to include people who will walk to and from bus stops, or be picked up by taxis.

   We will start with a few simple classes:

   - Location - This class encapsulates locations within the city. At this point it will consist of a square grid, so a Location has x and y coordinates, but future implementations could involve actual city streets. As entities move in the city they can move to any adjacent Location. This class should have the following capabilities:
     - Construct a new Location with random x and y values.
     - Construct a new Location with given x and y values.

  - Find the distance from this Location to another Location (i.e. the number of distinct moves to get from one to the other).
  - Determine which way to go in order to move closer to a given target Location.

- Passenger - This class deals with Passengers who will be riding on various kinds of Vehicles (such as Busses). A Passenger has a current Location and a destination Location. This class should have the following capabilities:
  - Produce a new Passenger with random origin and destination.
  - Move toward the nearest bus stop, if not on a vehicle.
  - Move toward destination after exiting a vehicle.
- Vehicle - An abstract class which is a superclass of Bus, Taxi, Car, etc. A Vehicle has a capacity (i.e. number of passengers it can accommodate), a current Location, a destination Location, and possibly a speed. This class should have the following capabilities:
  - Move closer to its destination Location.
- Bus - A Bus is a Vehicle. It should have:
  - a List of Locations which are the Locations where it stops to pick up or discharge Passengers. The Bus should move from one stop to the next, in a continuous circle (after the last stop in the List has been reached, it should then proceed to the first stop).
  - It should have a List of Passengers who are riding on the Bus.

  This class should have the following capabilities (in addition to the capabilties inherited from Vehicle):
  - Pick up passengers waiting at a bus stop.
  - Discharge passengers who have arrived at their stop.
  - Provide the nearest stop to a given Location.
- Actor - An Actor is anything which acts, or takes part in the simulation. This includes Passengers and all Vehicles. Since a Bus is a Vehicle, and a Bus is an Actor, Actor will have to be an interface. It will have only one method - `void act()`. Each class which implements Actor will have its own implementation of the `act()` method.
- Simulation - This class will initialize and drive the simulation. It will have a List of Actors and possibly a List of Passengers who have been generated during the Simulation. This class will have methods to:
  - Initialize the Simulation with one or more Vehicles. Each of these should be added to the List of Actors.
  - Run the simulation for one step. On each step, the Passenger class should be given the opportunity to create a new Passenger. If one is created, it should be added to the List of Actors (and Passengers, if necessary). Then each Actor in the simulation

should be told to act. For Passengers this could mean moving toward a Bus stop or toward a destination. For Vehicles this could mean moving toward a destination. For Buses this could mean picking up or discharging passengers if it is at a stop.

- Run the simulation for a given number of steps.
- Provide the nearest bus stop to a given Location.
- Provide a List of Passengers at a given Location.

To test your simulation, print all the Actors on each step to see if they are behaving in a sensible way (this means you will need `toString()` methods).

These specifications allow for some leeway in implementation, and no two solutions will be the same. If time permits, you can extend the simulation to include Cars, Taxis, Bikes, Pedestrians, etc.

3.

Chapter 7

Maps, Collections Revisited

7.1 Fast look-up

One of the most important functions of any computer system is to store large quantities of data, and provide quick access to any portion of that data storage. The ability to store a lot of data is of little value if we cannot access what we need quickly.

Consider the young researcher who is visiting the New York Public Library (one of the world's largest) and needs the answer to a question: What is the relationship, if any, between per-capita income and suicide rate across countries in the world? Our young friend has been assured that the information needed to answer the question is stored somewhere in that library, but how can he/she find it? One solution would be to walk to the nearest stack and read through every book on the shelf, then proceed to the next stack, and so on until he/she eventually finds the necessary information or determines that it is not to be found in all the volumes of that huge library.

This strategy is probably doomed to failure from the start; the researcher will not find what he/she is looking for in his/her lifetime, and will probably tire of the task long before that. The point is that smart search methods are critical to the information retrieval problem. Consequently the stored data must be organized in such a way that it can be searched quickly.

7.1.1 Exercises

1. Use a hard-copy dictionary to find a solution to one of the following problems:

   (a) Find a word which means "(adj) Covered or marked with numerous shallow depressions, grooves, or pits."

   (b) Find the definition of the word *wollasonite*.

2. In 1945 the author Max Shulman published a collection of humerous short stories titled *The Many Loves of Dobie Gillis*. The title character was a university student who spent more time chasing after women than he did studying. In one of the stories, "The face is familiar but...", Dobie is introduced to a beautiful girl at a dance but doesn't hear her name. He spends the rest of the evening trying to get her name, but fails at every attempt. He takes her home that evening, and she gives him her phone number; he now knows her home address and phone number but not her name. Among the several tactics that Dobie uses to discover this girl's name is the following:

   He approaches a pledge to his fraternity named Ed and says:

   > Varlet, I have a task for you. Take yon telephone book and look through it until you find the name of the people who have telephone number Kenwood 6817.

   [This was in the days before cellular phones.] The story continues, narrated by Dobie Gillis:

   > In ten minutes Ed was in my room with Roger Goodhue, the president of the fraternity. "Dobie", said Roger, "you are acquainted with the university policy regarding the hazing of pledges... You know very well that hazing was outlawed this year by the Dean of Student Affairs. And yet you go right ahead and haze poor Ed."

   (a) Explain why Dobie Gillis was accused of hazing a pledge.

   (b) Would Dobie have been accused of hazing if he had told the pledge to search for the girl's home address instead of her phone number?

7.2 Sequential search

The strategy of searching by starting at the beginning (of a list, for example) and examining every item until you find the one you are looking for is called a *sequential search*. For relatively small lists it is not unreasonable, and it is easy to implement. The following method will return the position of the target in a list of numbers, or -1 if the target is not found:

```
/** @return Position of target in the given list, or -1 if not found
 */
public int sequentialSearch(List <Integer> numbers, int target)
{  int pos = 0;
   for (int n : numbers)
     {  if (n == target)
           return pos;          // found the target
        pos++;
```

```
    }
return -1;                    // target not found
```

If the size of the list is less than a few million, this will not take long. However, for much larger lists a sequential search will not be acceptable. The algorithms which may be involved for more effective searching are discussed in chapter 12 In addition, Java provides some fairly clear classes which can be used to access data quickly.

7.2.1 Exercises

1. If it takes 5 nanoseconds for one iteration of the loop in the `sequentialSearch` method shown above, how long would it take to search a list of 50,000 numbers:

   (a) In the best case (the target is the first number in the list)?

   (b) In the worst case (the target is not in the list)?

   (c) In the average case (the target is in the list, but could be anywhere in the list)?

   Hint: A nanosecond is 10^{-9} sec.

2. Assuming that `nums` is a List of 1250 numbers, given the following code, how many times will the comparison in the `sequentialSearch` method (`if (n == target)`) be executed?

```
int countMissing = 0;
for (int i=0; i<10000; i++)
    if (sequentialSearch(nums, i) == -1)
       countMissing++;
```

7.3 Java maps

In an English dictionary we have a list of words, with a definition for each word. We will call each word, together with its definition, an *entry* in the dictionary. It is easy to look up a word in the dictionary to find its definition, but it is very difficult or time consuming to look up a definition. For example, what is the word which means 'A fruiting body or the stalk of a fruting body in a fungus?' You can find it by starting on page 1 and looking at the defintion of each word until you arrive at this definition. This is a sequential search and would probably take a long time. The words in a dictionary are called *keys*. The definition of a word is called a *value*. When using the dictionary, we always search for an entry using its key, and we never search for an entry using its value. There are a few classes in the Java Class Library which give us a key-value look-up capability, and they are all implementations of the *Map interface*. Maps can be found in the java.util package, along with collections, though strictly speaking a Map is

not a collection. Looking at the API for the Map interface we see that there are methods which allow us to add an entry to a map, search a map using a key, remove an entry from a map, etc. As with lists and sets, we can specify the type of the items stored; however, with maps we will specify two types: the type of the keys (K) and the type of the values (V). Each entry in a map consists of a key-value pair. Here are a few of the most useful methods which are available for all maps:

- Put a new entry into a map. If the key of the new entry is already in the map, the new value replaces the existing value, and the old value is returned.

```
/** Put the given key-value pair into this Map.
 *  If the key is already in this Map, replace the existing
 *  value with the*  given value.
 *  @return The existing value for the given key,
 *  or null if the given key is not found in this Map.
 */
V   put (K key, V value);
```

  Get a value from a map. Provide a key to obtain its corresponding value.

- ```
  /**
   *  @return The value corresponding to the given key, or null if the
   *  given key is not found in this Map.
   */
  V   get (Object key);
  ```

  Note that the parameter need not be of any particular type.

- Determine whether a given key is in this Map.

```
/**
 *  @return true only if the given key is in this Map.
 */
boolean   containsKey (Object key);
```

  Note that the parameter need not be of any particular type.

- Remove the entry with the given key from a map.

```
/** Remove the entry with the given key from this Map.
 *  @return The existing value corresponding to the given key,
 *  or null if the given key is not found in this Map.
 */
V   remove (Object key);
```

  Note that the parameter need not be of any particular type.
```

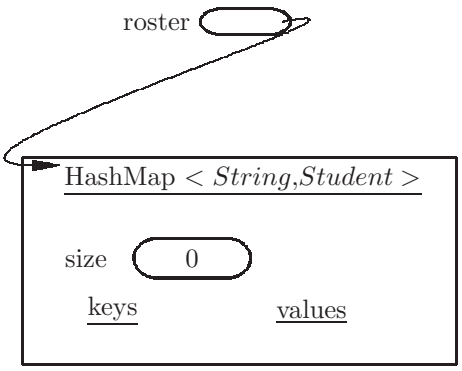


Figure 7.1: An empty map in which the keys are Strings and the values are Students

- Determine the number of entries in a map.

```
 /** @return The number of entries in this Map.  */
int size();
```

- Obtain a set of all the keys in a map.

```
/**
 *  @return All the keys from this Map, as a Set.
 */
Set<K> keySet();
```

In a map, the keys must be *unique*; i.e. there cannot be two entries with the same key. This should be evident from the description of the `put` method above. On the other hand there may be several entries with the same value.

We can further explain the structure of a map by looking at object diagrams. In an object diagram we will show the size of a map, as we did with lists and sets, at the top of the object. The size of a map is simply the number of entries contained. This is followed by two columns: the left column is for the keys, and the right column is for the corresponding values. As with lists and sets, we will treat wrapper classes and Strings as primitives, showing their values directly rather than as references to other objects.

Figure 7.1 shows a newly created map which is empty. No entries have been put into this map. The keys in this map are student numbers (as Strings), and the corresponding values are Students. The variable `myMap` does not store a null reference; it stores a reference to the map object, which has a size of 0.

Figure 7.2 shows the object diagram for a map into which three entries have been put. The size is now 3, and each of the three entries consists of a key (a String) and a value (a reference to a Student).

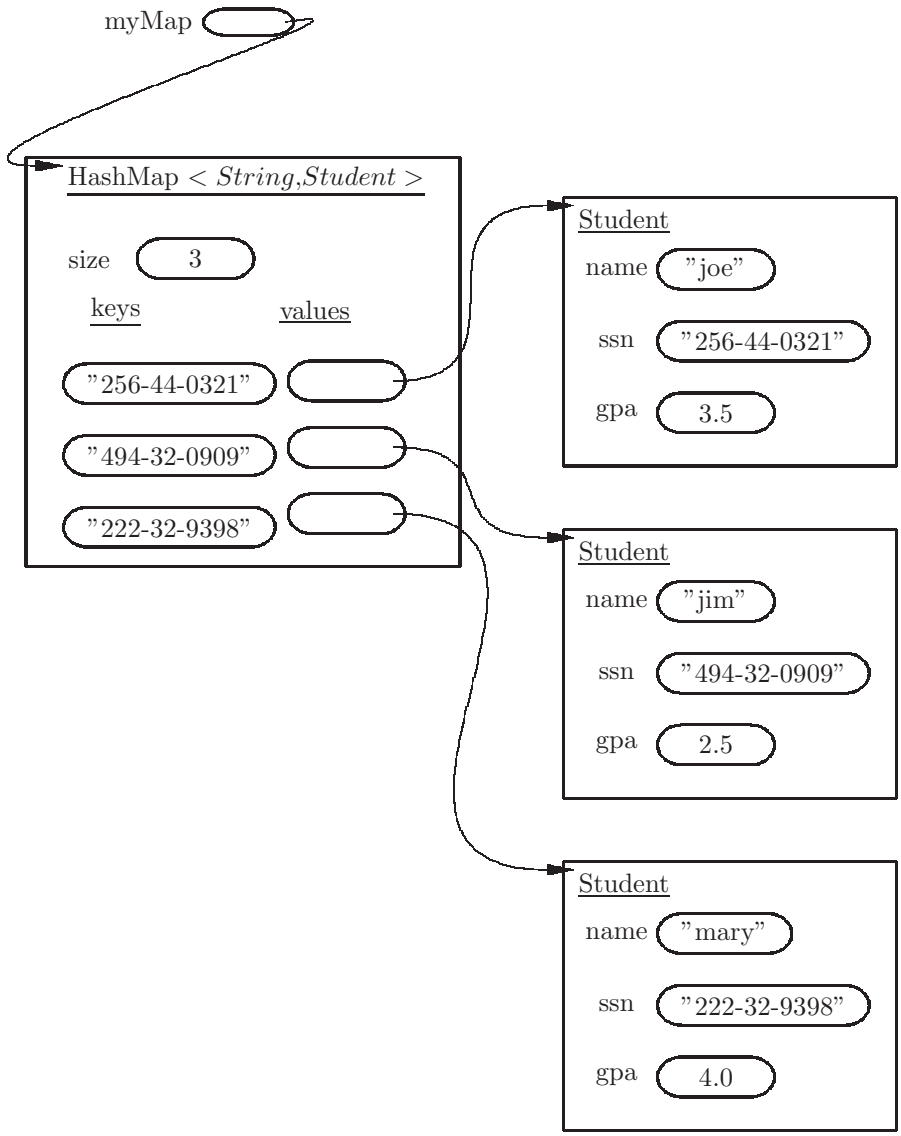


Figure 7.2: An object diagram showing the value of the variable `myMap` storing a reference to a map, after three entries have been added

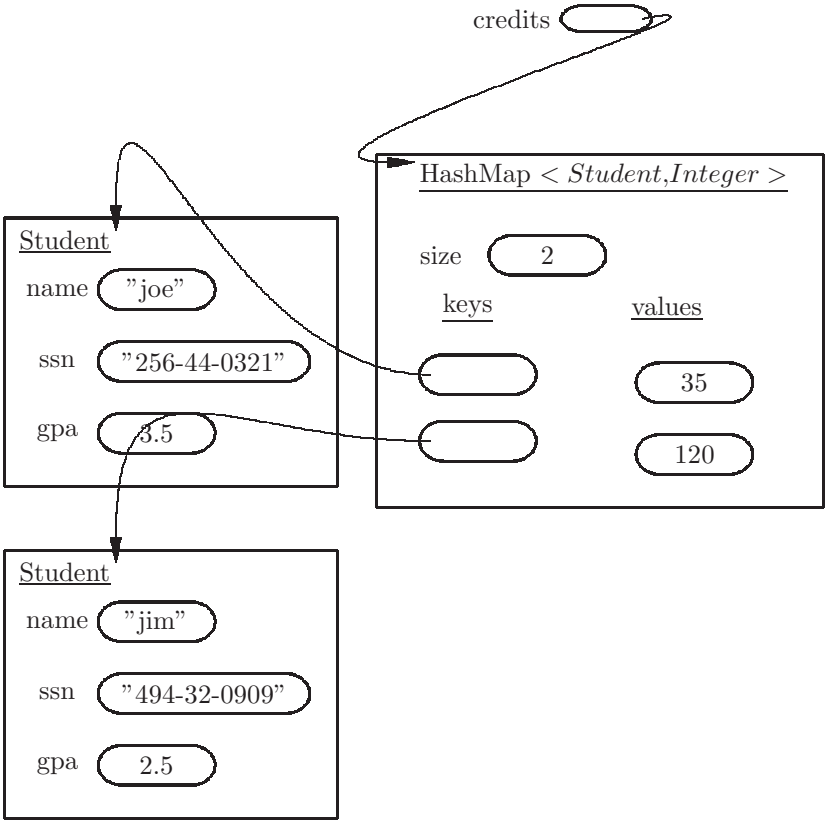


Figure 7.3: An object diagram showing the value of the variable `credits` storing a reference to a map, in which the keys are students and the values are the number of credits accrued, after two entries have been added

Note that the keys need not be primitive types and the values need not be reference types. Figure 7.3 shows the object diagram for a map in which the keys are Students and the values are the number of credits accrued by the corresponding Student. Two entries have been put into this map.

7.3.1 Exercises

1. What value is returned by the `put` method when the key of the entry bein put into a map is not already in the map?

2. What is the effect on a map when the key of the entry being put into the map is already in the map? In this case what value is returned by the `put` method?

3. What value is returned by the `get` method when the key is not in the map?

4. What method (other than `get`) can be used to determine whether a given key is already in a map?

5. What is returned by the `remove` method when removing an entry from a map?

6. True or false:

   (a) The values in a map must be unique.

   (b) The keys in a map must be unique.

   (c) The size of a map is twice the number of entries because each entry consists of a key and a value.

7.4 Examples of methods which use maps

Before we see how to instantiate maps, we will show a few examples of methods which make use of maps. These methods assume the map has already been created and may contain several entries.

```
/** @return The name of the student with the given ssn,
    or "NOT FOUND"  if the student is not in the given map.
    @param roster is a map in which the key is an ssn,
    and the value is the corresponding Student.
 */
public String getName (String ssn, Map<String,Student> roster)
{
    String result = roster.get(ssn);
    if (result == null)
       return "NOT FOUND";
    return result.getName();
}
```

In this example each entry in the map named `roster` stores an ssn as key, and a reference to the corresponding Student as the value. This method uses the `get` method to extract the Student with the given ssn. If the ssn is not in the map, a null reference is returned, and the method shown here checks for null and returns the String "NOT FOUND" in that case. Otherwise it will invoke the getName() method on the Student, to obtain the Student's name.

In the next example we show a method which will count the number of students in the given map who have a perfect GPA.

```
/** @return the number of students in the map roster who
 *  have a perfect 4.0 GPA.
```

```
 */
public int perfectCount (Map <String,Student> roster)
{  Set <String> ssns = roster.keySet();    // set of all keys in the map
   int count = 0;                          // result
   Student student;

   for (String ssn : ssn)
      {  student = roster.get(ssn);        // get the student
         if (student.getGPA() == 4.0)
            count++;
      }
   return count;
}

In this method we use the \texttt{keySet()} method to obtain a Set
of all the keys in the given map.
Then we use a for-each loop to cycle through the set of ssns,
incementing the \texttt{count} variable each time we encounter a Student
with a GPA of 4.0.
If the map is empty, the loop repeats 0 times, and the returned
value is 0.
Note that the call to \texttt{roster.get(ssn)} cannot return a null
reference, because the ssn was obtained from the map in the first place.
```

7.4.1 Exercises

1. Define a method which will count the number of occurrences of a student with a given name in a given map in which the keys are students' ssns, and the values are the corresponding students.

   ```
   /** @param roster Is a Map in which the keys are ssns, and the
                     values are the corresponding students.
    *  @param name  Is the name of a student which may be in the given map.
    *  @return The number of students in roster with the given name.
    */
   public int countNames (Map <String,Student> roster, String name)
   ```

2. Define a method which will return the number of ssns in the given list occur in the given map. Do not use the `get` method.

   ```
   /** @param roster Is a Map in which the keys are ssns, and the
                     values are the corresponding students.
    *  @param ssns  Is a List of ssns.
    */
   public int countSSNs (Map <String, Student> roster, List <String> ssns)
   ```

3. We wish to count the number of occurrences of various words in some text. We will use a map to accomplish this; the keys will be Strings (i.e. the words) and the corresponding values will be Integers (the number of occurrences of the corresponding word). For example, if the text is `"I yam what I yam"` then the map will contain the following information:

   ```
   key           value
   ---           -----
   "I"           2
   "yam"         2
   "what"        1
   ```

   Define a method named `update` which will put a word into a given map, so as to tabulate the distribution of words as shown above. (To create the map shown above, your method would have been called 5 times, once for each word to be entered)

   ```
   /** Include the given word into the distribution map.
    *  @param word An (additional) word to be included in the map
    *  @param distribution A Map storing the number of occurrences of each of
    *         several words.
    */
   public void update (String word, Map <String, Integer> distribution)
   ```

   Hint: Before putting an entry in the map, check to see whether the word to be entered is already in the map.

4.

7.5 Instantiating maps

We are now ready to create maps. There are two kinds of maps available in the Java class library: *HashMaps* and *TreeMaps*. Both of these implement the Map interface. As with lists and sets, when *declaring* a variable, we will simply call it a Map, and this means that it is capable of storing a reference to some kind of Map. When *instantiating* an object, we will have to decide what kind of Map it should be.

7.5.1 HashMap

HashMap is a class in the package java.util which implements the Map interface. It is designed to provide quick access to any of its entries if you provide the key to the entry you are seeking. The order in which the entries are stored is determined by the HashMap class, and is not likely to be the same order in which the entries were put into the map. In this respect a HashMap is similar to a HashSet.

To declare a variable which can store a reference to any kind of Map, use the following format:

```
Map <keyType, valueType> variableName;
```

To instantiate the Map as a HashMap, and store the reference in the variable:

```
variableName = new HashMap <keyType, valueType> ();
```

The following code shows how to declare a variable, create an instance of a HashMap, and put three entries into it. The keys of this Map are Strings and the values are Students:

```
Map <String,Student> roster;        // roster is null
roster = new HashMap <String,Student> ();   // roster is not null
                                    // size of roster is 0 entries
Student st = new Student ("jim","254-33-3221");
roster.put ("254-33-3221",st);      // size of roster is 1 entry
st = new Student ("sue","873-34-856");
roster.put ("873-34-8563", st);     // size of roster is 2 entries

System.out.println (roster.get("873-34-8563"));     // prints sue
System.out.println (roster.get("999-32-2222"));     // prints null
                                                    // key not found

st = new Student ("sueAnn","873-34-8563", st);      // same ssn
st = roster.put (st.getSSN(), st);     // size of roster is still 2 entries
System.out.println (st);               // prints sue
System.out.println (roster.get ("873-34-8563");      // prints sueAnn
```

Note that using the `put` method does not necessarily increase the size of the map. When sueAnn was put into the map, her ssn was already in the map (as a key). So the reference for the corresponding value was simply replaced. In this case the `put` method returned a reference to the old value, sueAnn.

7.5.1.1 HashSets revisited

In chapter 5 we pointed out that when creating a set of objects, those objects must have two methods defined:

- `boolean equals (Object obj);`

  This method is needed because the items in a Set must be unique. When you attempt to add an item to a Set, the implementation (e.g. HashSet) needs to compare the item you are adding with the items already present in the Set to make sure there are no duplicates. Suppose you are working with a Set of Students. How can the add method in HashSet determine whether the given Student is already in the HashSet? It will need to compare the given Student with each Student in the Set. But how can equality of Students be determined? This must be decided by the Student class; this is where it is decided whether this Student is equal to some

other Student. Since we designed the Student class, we could decide, for example, that two Students are equal if and only if they have the same ssn.

In that case we could define a method in the Student class to determine whether two Students are equal, as shown below:

```
/** @return true only if the ssn of this student equals the
 *  ssn of the other Student.
 */
public boolean equals (Object other)
{  if (! (other instanceof Student)    // is other a Student?
      return false;
   Student otherAsStudent = (Student) other;  // cast to Student
   return this.ssn.equals (otherAsStudent.getSSN());
}
```

Here we are making use of the fact that the String class itself has an `equals(Object)` method. The parameter, `other`, in the above method is declared as Object, so as to be the same as the `equals` method in the Object class. If the parameter `other` is anything but a Student, our method needs to return `false`. That is the purpose of the *instanceof* operator. `someObject instanceof someClass` will return true only if `someObject` is an instance of `someClass`.

- `int hashCode();`

  Unfortunately a complete explanation of the need for this method is beyond the scope of this book. Suffice it to say that HashSets (and HashMaps) use hash tables for implementation, which require the use of a hash code. A good `hashCode()` strategy recommended by Bloch is shown below:

  1. Start with an inital value of 17.
  2. For each field which is used in the equals method:
     (a) If the field is a primitive type, use the field's value as an int.
     (b) If the field is a reference type, use the hashCode of that field.
  3. Multiply the result by 31 and add the field's value to the result

  A hashCode() method for the Student class is shown below:

```
/** @return a value such that two objects have the same
 *  hash code if they are equal, and two objects are
 *  likely to have different hash codes if they are not
 *  equal.
 */
```

```
    public int hashCode()
    {  int result = 17;
       result = 31 * result + ssn.hashCode();
    }
```

With these two methods in the Student class, we can now work with a HashSet of Students:

```
Set <Student> roster = new HashSet <Student> ();
roster.add (new Student ("jim", "343-55-8494"));
roster.add (new Student ("mary", "242-87-5943"));
```

7.5.1.2 HashMaps: Using our own class as a key: `equals(Object)` `hashCode()`

Figure 7.3 is a bit premature; in a HashMap the keys must have the methods `equals(Object)` and `hashCode()`. Now that we have these methods in the Student class, we can work with a HashMap in which the keys are Students.

As an example, suppose we wish to store the number of courses taken by each student. We could use a map in which the keys are Students and the values are Integers. Each value represents the number of courses taken by the corresponding Student. Each time a student registers for a course, we increment the value of that student's entry; each time a student drops a course, we decrement the value of that student's entry. Now that we have `equals(Object)` and `hashCode()` methods we can build the map:

```
public class CourseCounter
{
private  Map <Student, Integer> courseCounts =
         new HashMap <Student, Integer> ();

/** This method is called when a Student registers
 *  for one course.
 */
public void reg (Student st)
{  int count = 1;
   if (courseCounts.containsKey(st))
      { count = courseCounts.get(st);   // number of courses for st
        count++;
      }
   courseCounts.put(st,count);    // increment count for st
}

/** Drop one course for the given Student
 *  @param st A student who is registered for at
 *  least one course.
```

```
 */
public void drop (Student st)
{  int count;
   count = courseCount.get(st);
   count--;
   courseCount.put(st,count);
}

/** @return the number of courses for the given Student
 */
public int getCount (Student st)
{  int result = 0;
   if (courseCounts.containsKey(st))
      result = courseCounts.get(st);
   return result;
}

public String toString()
{   return courseCounts.toString();  }
}
```

In the example above when the `reg` method is called the student's course count is extracted from the map, incremented, and put back into the map. When the `drop` method is called, the student's course count is extracted from the map, decremented, and put back into the map.

We could use this class as shown below:

```
CourseCounter cc = new CourseCounter();
Student joe = new Student ("joe", "256-44-0321");
Student jim = new Student ("jim", "494-32-0909");
cc.reg(joe);
cc.reg(jim);         // registered for 1 course
cc.reg(joe);
cc.reg(joe);         // registered for 3 courses
```

The resulting object diagram for `cc` is shown in Figure 7.4

7.5.2 Exercises

1. Find the syntax error, if any, in each of the following:

   (a) `Map <String> myMap;`

   (b)
   ```
   Map <String, Student> myMap = null;
   myMap = new HashMap <Student,String> ();
   ```

   (c)
   ```
   Map <String, Student> myMap;
   myMap = new HashSet <String, Student ();
   ```

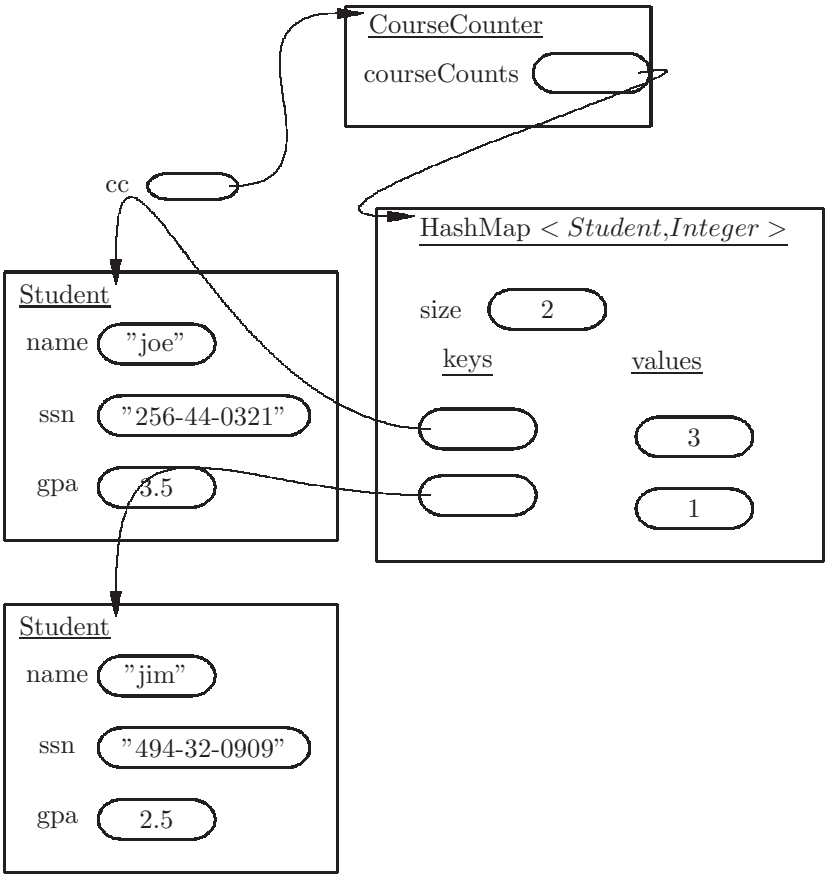


Figure 7.4: An object diagram showing the value of the variable `cc` storing a reference to a CourseCounter object, which stores a reference to a map, in which the keys are students and the values are the number of courses for which the corresponding Student has registered

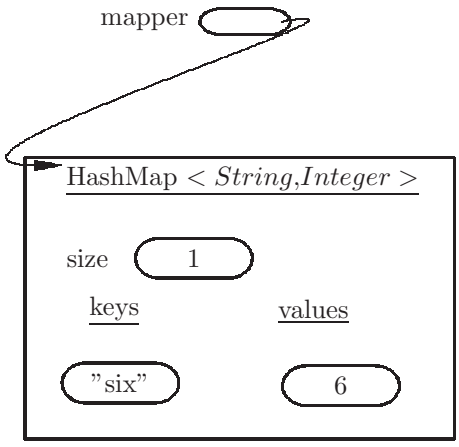


Figure 7.5: Object diagram for a variable storing a reference to a Map (see Exercises)

2. Show the code which will declare a variable which stores a reference to a Map in which the values are Strings and the keys are Integers. It should initialize that variable with a reference to an empty HashMap.

3. Draw object diagrams for the variables *map1*, *map2*, and *map3*, after the code shown below has executed:

```
Map <Integer, String> map1;
Map <Integer, String> map2 = new HashMap <Integer,String>();
Map <Integer, String> map3 = new HashMap <Integer,String>();
map3.put (5, "five");
```

4. Show the code which would have produced the object diagram shown in Figure 7.5.

7.6 TreeMap and Collections revisited: TreeSet and LinkedList

Now that we have a better understanding of inheritance, interfaces, and maps, we are ready to take a look at another class that implements the Set interface, and another class that implements the Map interface.

7.6.1 TreeSets

In chapter 5 we mentioned that there is more than one class in the Java class library which implements the Set interface. In that chapter we discussed Hash-

sets; we now present another class which implements the Set interface – *TreeSet*.

We mentioned that sets, in general, are not concerned with the order in which its items are stored. The TreeSet, in a sense, contradicts this property; a TreeSet will maintain its elements in increasing order. If we have a TreeSet of numbers, and we iterate through that TreeSet, we will obtain the numbers in increasing order, regardless of the order in which they were added to the TreeSet. Strings would be obtained in alphabetic order. Other than that, TreeSets and HashSets have very similar behavior. They both provide fast access to any item in a set.

To instantiate a TreeSet of Strings, and add three Strings to it:

```
Set <String> names;
names = new TreeSet <String> ();
names.add ("jim");
names.add ("al");
names.add ("mary");
```

If we were to cycle through this set with an iterator, we would obtain the items in the following order:

`"al", "jim", "mary"`

For a TreeSet storing objects of some other class, that class must have a *compareTo* method which can be used to determine whether a given object is less, equal, or greater than another object of the same class.

For example, if we were to create a TreeSet of Students, our Student class would need a `compareTo` method. Here is an example of a `compareTo` method for the Student class, assuming that we wish Students to be ordered by SSN:

```
/** @return  a negative number if this Student precedes the other Student,
             0 if this Student equals the other Student,
             a positive number if this Student follows the other Student.
 *  Students are to be ordered by SSN
 */
public int compareTo (Student other)
{   return ssn.compareTo(other.ssn);    }
```

The `compareTo` method is a standard method found throughout the Java class library. In the comparison `foo.compareTo(other)`, the returned value is negative if foo is less than bar (foo precedes bar) and positive if foo is greater than bar (foo follows bar). The easiest way to remember this convention is to imagine that the compareTo method simply does the subtraction: `foo - bar` (which is what it actually does).

One additional modification is needed in the Student class. The class should implement the `Comparable` interface:

`public class Student implements Comparable<Student>`

Thus, the compiler will require you to include the compareTo method in the Student class, which will permit the client to compare this Student with any other Student. In the implementation of this method you will need to decide

how to order Students. You may decide, for example, to use the Student's ssn as the sole criterion for ordering Students, as shown above. The `compareTo` method is needed by `TreeSet` so that the `TreeSet` class can keep the elements in order.

Having included this method in our Student class, we can now create a TreeSet of Students:

```
Set <Student> roster;
roster = new TreeSet <Student> ();
roster.add (new Student ("al", "832-43-4342"));
roster.add (new Student ("mary", "135-34-7839"));
roster.add (new Student ("joe", "135-27-3482"));
```

An iterator would produce these Students in the order `joe, mary, al`.

7.6.2 TreeMaps

One of the methods in the Map API is *keySet*. It returns a Set of all the keys in the map. If we wish to search the entries in a Map, we must first obtain a Set of all keys in the Map. We can then iterate through the Set of keys, using each key to obtain an entry in the Map. The example shown below is a method which will print all Students who have a sufficiently high GPA.

```
/**
  * @param roster is a Map in which the keys are Students' SSNs and
  * the values are the corresponding Students.
  * @param min The minimum GPA required to be considered a good Student.
  * This method will print all Students with a gpa of min or greater.
  */
public void showGoodStudents (Map <String,Student> roster, double min)
{  Set <String> ssns = roster.keySet();   // obtain the set of ssns
   Iterator <String> itty = ssns.iterator();
   String ssn;
   Student st;
   System.out.println ("Students with a minimum GPA of " + min);
   while (itty.hasNext())
     {  ssn = itty.next();
        st = roster.get(ssn);
        if (st.getGPA() >= min)
           System.out.println (st + "\n");
     }
}
```

Notice that this method knows that `roster` stores a reference to a Map, but doesn't know what kind of Map it may be. If it happens to be a HashMap, the sequence in which the Students are obtained with the Iterator is unspecified.

They could be obtained in any order; moreover, if entries are added or removed, the ordering could be totally different.

We now introduce another kind of Map called *TreeMap*. A TreeMap is similar to a TreeSet in that the entries will be ordered according to the `compareTo` method of the Map's keys. If the keys are Strings, the keys will be in alphabetic order. More precisely, they will be obtained in alphabetic order by an Iterator.

If the parameter `roster` in the method shown above happens to be a reference to a TreeMap, the Students will be printed in order of increasing SSN, since the SSN is a String.

7.6.3 LinkedList

In chapter 5 we introduced the List interface, and an implementing class called ArrayList. We also mentioned that there could be other implementing classes for the List interface; i.e. there could be other kinds of Lists. In this section we introduce one such List, called *LinkedList*. Because a `LinkedList` implements List, it has all the methods shown in the List interface: `add, get, set, size, remove, ....`

As an example, we show how to instantiate a LinkedList and add some items to it in the code segment below:

```
List <Integer> grades = new LinkedList <Integer> ();
grades.add (92);
grades.add (88);
grades.add (100);
System.out.println (grades.get(2));
```

In this code segment the only unusual aspect is that we have instantiated a LinkedList instead of an ArrayList.

For the most part, everything you can do with an ArrayList you can also do with a LinkedList, and vice versa. So why is there a need for LinkedList at all? The answer concerns run-time efficiency. These two classes can differ in the time required to work with long lists. When processing a long list, some operations can take a long time for ArrayLists, but a short time for LinkedLists, and vice versa.

To decide which kind of List should be used in a particular application, we offer the following general guidelines:

- If the size of the list will be changing (increasing and decreasing) as the program executes, LinkedList will be faster than ArrayList.
- If the size of the list will remain stable, ArrayList might be faster than LinkedList.
- The `get` and `set` methods are slow for LinkedList.
- The `add` and `remove` methods are slow for ArrayList (they both change the size of the list).

Operation	ArrayList	LinkedList
get(int inx)	fast	slow
set(int ndx, E item)	fast	slow
add(E item)	ok	fast
add(int ndx, E item)	slow	fast
remove (int ndx)	slow	fast

Figure 7.6: Relative efficiency of operations on ArrayLists vs. LinkedLists

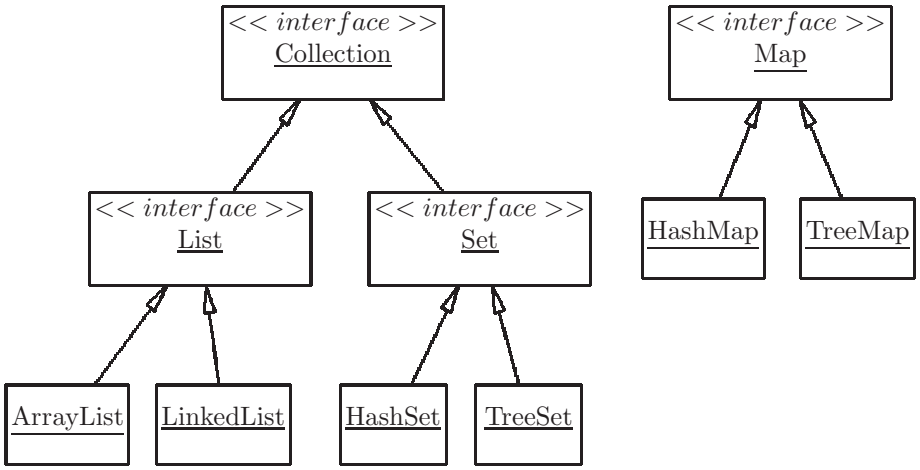


Figure 7.7: Class diagram showing some of the interfaces and classes in the package java.util

- When working with a variable declared as `List` and the actual kind of List is unknown, use an Iterator to cycle through the items of the List.

These performance characteristics are summarized in Figure 7.6. The `add (E item)` method adds an item at the end of a List. The figure indicates this operation is 'ok'. By this we mean that adding at the end of an ArrayList will generally be fast; there might be occassions when it would slow down a little, but this is not a major concern. However, when inserting an item at an arbitrary position in a List – `add(int ndx, item)` – LinkedList is clearly faster than ArrayList. You will understand why these performance characteristics are as shown here, and you will be able to describe them mathematically, when you study Data Structures.

Now that we have taken a look at several of the classes in the java.util package of the Java class library, we can examine the way in which they relate to each other. The best way to do that is with a class diagram. Figure 7.7 shows a class diagram for some of the classes in the Java class library (Not shown is the `Object` class which is a super-class, directly or indirectly, of all classes).

7.6.4 Exercises

1. Point out the syntax error, if any, in each of the following (assume the Student class implements Comparable¡Student¿):

   (a) `Set <String> names = new Set <String> ();`

   (b) `Set <String> names = new TreeSet <Student> ();`

   (c) `Map <Integer,Student> grades= new TreeMap <Integer, Student> ();`

   (d) `List <Integer,Student> grades = new LinkedList <Integer, Student> ();`

2. Define a method which, given a Set of Students, will print all Students whose names are longer than 10 characters. In what order will those students be printed?

```
/** Print all Students in roster whose name is longer than 10
 *  characters.
 */
public void showLongNames (Set <Student> roster)
```

3. Define a method similar to the previous problem in which the parameter is a Map rather than a Set. In what order will those students be printed?

```
/** Print all Students in roster whose name is longer than 10
 *  characters.
 */
public void showLongNames (Map <String, Student> roster)
```

4. Define a method which, given a Set of Students, will print all the Students in order, by ssn, one student per line. Assume the `compareTo` method in the Student class orders Students by ssn.

```
/** Print all Students in roster whose name is longer than 10
 *  characters.
 */
public void showBySSN (Set <Student> roster)
```

   Hint: Copy the Students to a TreeSet and print the TreeSet.

5. Define a method which, given a list of integers, will return the largest integer in the list. Be concerned with the efficiency (i.e. execution time) of your method.

```
/** @return The largest value in numbers.
 */
public int largest (List <Integer> numbers)
```

6. Define a method which, given a List of Students, will return a List of those with a perfect GPA (4.0). Be concerned with the efficiency (i.e. execution time) of your method.

```
/** @return A List of all Students in roster who
 *  have a perfect 4.0 GPA.
 */
public List <Student> perfect (List <Student> roster)
```

7. For each of the following determine whether the List involved should be an ArrayList or a LinkedList (or whether either is ok), for purposes of efficiency:

   (a) A method which, given a List, determines whether the elements are in ascending order.

   (b) A method which, given a List of items, obtains other items from the user's keyboard, and, one at a time, deletes each of those items if found in the given list.

   (c) A method which, given a List of items in ascending order, obtains other items from the user's keyboard and inserts them at the appropriate places in the given List.

   (d) A method which, given a List of items, will arrange those items in ascending order. The given List may contain duplicate values.

7.7 Projects

1. Use the project `university` for this project. It should contain the `Student` class which we have been using. We wish to implement a simple information system for a university in this project. Define a new class named `UniversityInfoSys`. This class should maintain information on all students admitted to a university. It should have a field which stores all the students in a Map in which the keys are Strings (ssns) and the values are Students. In addition to a constructor, it should have the following methods:

   - A method to add a new student to this university.

```
/** Add the Student st to this UniversityInfoSys
 *  @return false If not added (e.g. already in the system)
 */
public boolean addStudent(Student st)
```

   - A method to obtain a reference to a list of all students who are active at this university.

```
/** @return a List of all Students in the system
 */
public List <Student> getStudents()
```

- A method to search for a particular student, given the student's ssn. This method should not involve a loop.

```
/**
 * @return the student with the given ssn, or
 * null if not found.
 */
public Student searchBySSN(String ssn)
```

- A method to search for all students, who have a given name.

```
/**
 *  @return the set of students with the
 *  given name. Must match exactly, case
 *  sensitive.
 */
public Set <Student> searchByName(String name)
```

- A method to find the average GPA of all students at this university.

```
/** @return Average GPA of all students
 */
public double averageGPA()
```

- A method to remove those students whose GPA has fallen below a given minimum value.

```
/** Remove all students who have a GPA less than minimum.
 *  @return Number of students dismissed.
 */
public int dismiss (double minimum)
```

2. In a new project, we wish to develop a class which will enable us to encrypt and decrypt secret messages. We will use a few maps for this purpose. Cryptologists call this kind of encryption scheme a *codebook*. Open the project `crypto` from the repository for this chapter. Define methods to:

   - Build a map which stores English words as keys, and the corresponding encrypted words as the values. Make up your own entries, such as:

plain text	cipher text
the	spritz
fox	glmph
jump	foo

   - Build the inverse map, i.e. the map in which the keys are encrypted (i.e. cipher) words, and the corresponding values are real English

words (i.e. plain). Build this map automatically from the other map, using a loop.

- Given a List of English words, return a List of the corresponding encrypted words.
- Given a List of encrypted words, return a List of the corresponding English words.
- A method to test your work is provided. It encrypts the words "I have things to do", producing cipher text. It then decrypts the cipher text to produce the original plain text.

3. You have intercepted a message sent by the enemy, and you need to decode it to save us from attack. All you know is that the enemy is using a permutation cipher; each word in the intercepted message is simply a permutation (i.e. an anagram) of an English word. The message is:

   ```
   niaiuanmrisotzrtiiocm fo uealcisonlesm npesoaw tpso nialtrenosotiaertuc
   ```

   Use the project `unscramble` from the repository. In this project a class named `WordProduce` is provided. It has methods which will provide you with over 10,000 English words, one at a time, each time you call the `getWord()` method. This class also has a boolean method which will tell you whether there are more words yet to be obtained. Use this method to control a loop in which you call `getWord()`.

   The strategy here is to use a map in which the keys are Strings in which the characters are in alphabetic order. The corresponding values are sets of Strings which are the anagrams of the keys. For example, one entry in the map could be:

   ```
   arst = {"arts", "rats", "star", "tars", "tsar"}
   ```

   Here is what you'll need to do:

   (a) In the MapBuilder class implement a method to build the map, using wordProducer. It should return a reference to the map.

   Hint: In the String class there is a method which will produce an array of chars from a String. In the Arrays class there is a static method which will sort an array of chars.

   (b) In the Unscrambler class:

   i. In the constructor instantiate MapBuilder, and use it to create the map.

   ii. Define a method which will get the anagrams of a sorted String from the map.

iii. Define a method which will show all the anagrams of each word in a given list of words.

iv. Define a method which will allow you to test what you have done. See if you can decode the secret message shown above.

Chapter 8

Exceptions - Handling Errors

Unfortunately, most software projects of appreciable size are not flawless. Because of the complexity of software, it is not unusual to encounter incorrect behavior (bugs) when using software. If the project is viable, the bugs are corrected as they are encountered and reported; over time the quality of the software improves. Some software applications which have been widely used for a long period of time are relatively robust. However, getting to that point is a long and difficul road.

Some bugs result in a minor inconvenience to the user; the program comes to a crashing halt and needs to be restarted. In other cases incorrect output can go undetected, until it is too late. There are also safety-critical applications in which bugs are completely unacceptable:

- Software controlling the flight patterns of aircraft at a busy airport.
- Software controlling medical machinery such as X-ray machines, heart-lung machines, intensive care monitor machinery, etc.
- Software used for national defense – early attack warning systems, radar systems, secure international communications software.
- Military applications – drone control systems, missile guidance systems, field communications cryptologic software, etc..
- NASA flight control software.

There are many more applications, too numerous to mention, where we rely for our own safety on computerized systems. Such systems are tested rigorously, and often have fail-safe redundancies built in. Nevertheless, in developing software we will find that debugging is an inevitable part of the process.

In this chapter we examine some of the run-time errors which can occur when a Java program executes. We will also attempt to 'trap' or *catch* those

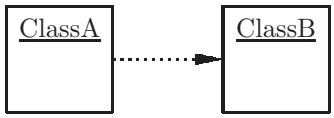


Figure 8.1: Class diagram showing a client (ClassA) and server (ClassB)

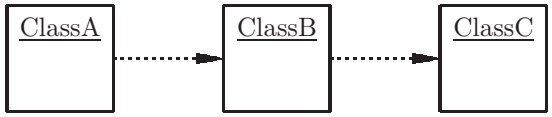


Figure 8.2: Class diagram showing a class (ClassB) which acts as both client and server

errors, and handle them in such a way that the program continues to execute without crashing. In the best of all worlds, if an error does occur, the system will tolerate the error and continue execution, and the the user will be unaware that all this has transpired. We call this *error-handling* or *fault-tolerant computing*.

8.1 Client/Server terminology

Before discussing the handling of errors, we should introduce some terminology having to do with the providing of services. This terminology can refer to classes which provide services to other classes, or to methods which provide services to other methods.

If classA uses the services of classB, we say that classA is a *client* of classB and that classB is a *server* for classB. A server class is simply the one providing one or more services to one or more client classes. This is often indicated in class diagrams as shown in Figure 8.1. A dashed arrow pointing from classA to classB means that classA *uses* classB. In this case classA is the client, and classB is the server. Note that the arrow does *not* represent inheritance; a solid arrow with hollow arrowhead is used to represent inheritance.

Suppose we introduce a third item, classC, and that classB uses the services of classC. The class diagram is shown in Figure 8.2. We now see that the terms 'client' and 'server' are relative to a context. As in Figure 8.1, classA is still a client of classB, and classB is still a server for classA, but now classB is a client of classC, and classC is a server for classB. In other words, classB acts as both client and server.

This terminology can also be applied to methods. If methodA calls methodB, methodA is a client of methodB, and methodB is a server for methodA. As with classes, a method can be both a client and a server (and this is very often the case).

8.1.1 Exercises

1. Refer to the project `university` in the repository for chapter 7.
   (a) Which class is a client and which is a server?
   (b) In the `Student` class which methods are client methods, and which methods are server methods?
2. True or False:
   (a) A *client*, as defined in this section, is a person.
   (b) It is possible for one class to act as both client and server.
   (c) The method `getName()` in the `Student` class is a client of the method `searchByName()` in the `UniversityInfoSys` class.

8.2 Assertions

Before getting to Exceptions, we would like to address the problem of determining the actual location of a program error (bug). Java has the capability of reporting to the programmer the method and line number within the source file where the error took place, along with a brief description of the nature of the error. Older programming languages and run-time systems provided little more than a hexadecimal memory dump, leaving the programmer the rather difficult task of tracking down and fixing the error.

Despite all the information received from the Java run-time environment, finding the actual bug is not always a simple task. Consider a division-by-zero error. This might occur because a parameter passed to a method stored, erroneously, a value of zero. The real programming error took place in the calling method, i.e. the client method (or perhaps the *client's* client). The incorrect line(s) of code could be far removed from the actual line where the error occurred, as shown in Figure 8.3.

Because the actual cause of a run-time error can be buried deep in many levels of method calls, we could simplify the debugging process if we could somehow detect that our program is in a non-valid state before the error actually occurs. In the example above, in the method meth1, if the value of `a` is 4, there will eventually be a division-by-zero error in meth3. We can save ourselves a lot of time if we could detect the problem in meth1, before any other methods are called. This can be done with an *assertion*.

An assertion is a true/false statement about the state of the variables at some point in a method. An assertion *should be true*; if it is false, we'd like to know about it before going any further. Assertions come in two formats; the first format is:

```
assert booleanExpression;
```

The booleanExpression is an expression which evaluates to true or false. If the booleanExpression is true, execution continues as if the assertion did not exist. However, if the booleanExpression is false, the program terminates,

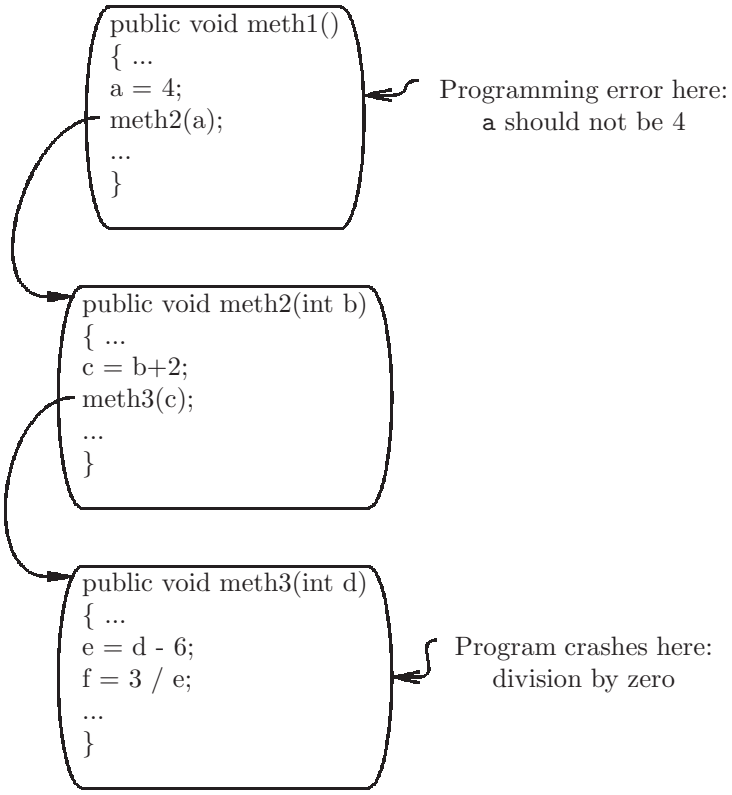


Figure 8.3: A run-time error buried deep in a series of method calls

providing information as to the location of the assertion which failed (name of method, and line number of the source file). In our example, we know that if the parameter `b` in `meth2` is 4, the program will ultimately fail, so we use that in our assertion:

```
public void meth2(int b)
{   assert b != 4;       // b should not be equal to 4
    ...                  // other statements not shown
    c = b+2;
    meth3(c);
    ...                  // other statements not shown
}
```

The second format for assertions allows us to put out more information about the cause of the problem:

```
assert booleanExpression :  String ;
```

The String should provide some helpful clues as to exactly what has gone wrong; the String need not be a String constant, but may be any expression which evaluates to a String:

```
public void meth2(int b)
{   assert b != 4 : "The value of b should not be 4";
    ...                  // other statements not shown
    c = b+2;
    meth3(c);
    ...                  // other statements not shown
}
```

In this case when the assertion fails the programmer will see the message, "The value of b should not be 4" in addition to the method name and line of the source file for the assertion.

Once we have finished testing our program and are ready to release it to users, we may wish to remove all the assertions. If we do so, and then a bug is encountered, we would have to put all the assertions back in place (a time consuming task).

Instead there is a better solution. Assertions can be disabled or enabled. When assertions are disabled, they are still in the source file, but the compiler does not include them in the output (.class file); in other words it is as if they have been removed. If further testing reveals a bug, we simply enable assertions to help discover the source of the error.

We caution the programmer to avoid doing something like the following:

```
assert roster.remove(6) != null :  "Position 6 of roster is null";
```

The remove method will remove the item at position 6 of roster, and return the reference at that position, which we compare against null. The problem with this is that when assertions are enabled, the size of `roster` is changed, but when assertions are disabled, the size of `roster` is not changed. The program

will exhibit different behavior depending on whether assertions are enabled or disabled – not a a good thing.

In summary, an assertion is a statement about the state of things as the program executes which should be true, and is generally false only if something unexpected or incorrect has occurred. As a college professor, I make the assertion: "All my students work hard and will pass this course". It is my hope that this assertion will always be true, but if it happens to be false, I'd like to understand what has caused it to be false.

8.2.1 Exercises

1. Refer to the `university` project in the repository for this chapter. What will be printed when each of the following code segments is executed?

   (a)
   ```
   String name = "joe";
   Student frosh;
   assert frosh != null : "reference to Student is null";
   frosh = new Student (name, "232-34-9756");
   System.out.println (frosh);
   ```

   (b)
   ```
   String name = "joe";
   Student frosh;
   assert name.length() > 0 : "Name " + name + " is not valid";
   frosh = new Student (name, "232-34-9756");
   System.out.println (frosh);
   ```

2.

8.3 Exceptions

8.3.1 Run-time errors resulting in an Exception

When a Java program is executing, the Java run-time system can detect an incorrect operation and determine that the program needs to be terminated. The particular kind of error is called an *Exception*, and the process in which it occurs is called a *throw*. This can result in program termination, also known as a *crash*. Java has the capability of providing the method and line number within the source file where the error took place, along with a brief description of the nature of the error (Exceptions are similar to assertions; Java assertions are actually implemented using Exceptions). Older programming languages and run-time systems provided little more than a hexadecimal core dump, leaving the programmer the rather difficult task of tracking down and fixing the error.

You may have encountered several examples of Exceptions already:

- `NullPointerException` – This occurs when you attempt to use a null reference inappropriately. In the expression `foo.bar` or `foo.bar()` the variable `foo` must not be `null`; if `foo` is null, a `NullPointerException` is

thrown, and the program terminates. Note that comparing a reference will not cause a problem: `if (foo == null) ...` is not a problem. When you get a NullPointerException, always look at the variable to the left of the dot as the likely culprit.

- `IndexOutOfBoundsException` – An array index or List index is not in the correct range. If `grades` is a List of size 5, the range of indices is [0..4] inclusive. Thus, either of the following will cause an IndexOutOfBoundsException:
  - `grades.get(5)`
  - `grades.get(-1)`
- `ArithmeticException` – This is typically a division by zero, though there could be other causes resulting from a calculation that cannot produce a valid result.
- `ConcurrentModificationException` – This results when changing the size of a Collection in a for-each loop. If you need to selectively remove or add items to a Collection in a loop, use an Iterator (a ListIterator will allow you to add items to a List).
- `ClassCastException` – This Exception is thrown when the Java run-time environment is unable to perform the requested cast. For example:

```
Student st = new GradStudent ("jim", "353-33-9303");
Undergrad under = (Undergrad) st;
```

  It is not possible to cast the variable `st` as an Undergrad because its dynamic type is GradStudent.
- `IllegalArgumentException` – This Exception is thrown when a method's parameter has a non-valid value. [1]

These are just a few of the more common Exceptions; they are all described in the API for the Java class library (see, for example, the package `java.lang`). As we will see, Exception is a class which has many sub-classes. Each package can have its own Exception classes.

In addition to the information provided by the Java run-time system, most IDEs provide a *debugger*, which is a utility that allows one to step through the statements of a program one at a time while watching the values of variables change. Though a debugger will not eliminate bugs for you, it is a useful tool that allows you to diagnose a problem and decide on an appropriate fix. Debuggers will be discussed in more detail later in this chapter.

[1] An *argument* is terminology carried over from other programming languages and is essentially the same as a parameter.

8.3.2 Throwing exceptions in a server method

Every public method should have an API which describes pre and post conditions for that method. These pre and post conditions constitute a contract for all client methods: If the pre conditions are satisfied, the post conditions will also be satisfied; if any pre condition is not satisfied, the server method is not under any obligation to do anything. This 'contract' is vital to the proper interaction of methods.

In the case where a precondition is not satisfied (the server method is not able to complete its obligations) the server method may wish to *throw* an Exception. This is a way of signalling that something is drastically wrong, and normal execution cannot continue. The throwing of an Exception differs from a failed assertion in that the client method can handle the Exception and attempt to continue executing the program; the program will not necessarily come to a crashing halt.

To throw an Exception, use a `throw` statement in which you instantiate the Exception being thrown:

```
throw new Exception-Name();
```

For example: `throw new IllegalArgumentException();`

When this is executed,

- The server method is terminated immediately; no return statement is executed, and even a non-void method is not required to return a value.
- Control returns to the statement in the client method which invoked the server method. The client method can then handle the Exception as described in the next section. If the client method chooses not to handle the Exception, the client method will throw the same Exception, to be handled by *its* client method. If no method handles the Exception, the Java runtime environment will bring the program to a crashing halt.

When instantiating the Exception, the particular Exception being instantiated may have several different constructors. They usually have at least a constructor with no parameters, and a constructor with a String parameter (as in the example above). The purpose of the String is to provide information for the programmer as to why the Exception was thrown. See the API for the Exception class to understand all the options available when instantiating an Exception.

If a server method can potentially throw an Exception, it is essential that this be made clear in the method's API. There is a javadoc keyword, *throws* used for this purpose. Consequently, the programmer writing a client method will understand that an Exception might be thrown, and be prepared for it. As an example, we return to our Student class in which we wish to calculate a Student's GPA:

```
/**
 *  @param gradePoints The number of gradePoints earned by this Student.
 *  @param credits The number of credits earned by this Student,
```

```
 *  should be positive.
 *  @throws IllegalArgumentException if credits is less than or equal
 *  to zero.
 */
public void calcGPA (int gradePoints, int credits)
{  if (credits <= 0)
      throw new IllegalArgumentException ("credits is " + credits);
   gpa = gradePoints / (double) credits;
}
```

If and when the IllegalArgumentException is thrown, the gpa is not calculated; instead we are back in the client method, where the thrown Exception can be handled or ignored.

One final caveat on throwing Exceptions is in order. Exceptions should be used only for exceptional, unexpected, or erroneous events. Do not use Exceptions in place of ordinary logic. Do NOT do the following (we have actually seen students do this):

```
// The WRONG way to control execution of a loop
try {
      int i=0;
      while (true)
         {  System.out.println (roster.get(i));
            i++;
         }
    }
catch (IndexOutOfBoundsException ioobe)
   {  }
```

8.3.3 What to do when an Exception is thrown

In this section we address the issue of handling a thrown Exception; we are concerned with the client method here, not the server method. Our client method has called a server method which has thrown an Exception and we need to decide how it should be handled or ignored in the client method:

- Should we ignore the exception? If so, our method automatically throws the same Exception, so that it can be handled in the client method which called our method.

- Should we throw the same Exception explicitly? This has the same effect as ignoring the Exception (but may be required depending on the kind of Exception – see checked vs. unchecked Exceptions, below)

- Should we try to handle the Exception right here in our method? This would involve a *try/catch* block.

8.3.4 Handling exceptions with try/catch in a client method

If we choose to handle a thrown Exception in our client method, we must use a statement called a *try/catch block*. This is a statement with at least two parts: a try block and one or more catch blocks. The format is:

```
try {  // statement(s) containing a call to a method which might
       // throw an Exception.
       // Include all statements which depend on the result of
       // the method call.
    }
catch (ExceptionClass name1)
    {  // Statements to handle the Exception
    }
catch (ExceptionClass name2)
    {  // Statements to handle the Exception
    }
.
.
.
catch (ExceptionClass name)
    {  // Statements to handle the Exception
    }
}
```

The try block includes a statement which calls a server method – the one which potentially will throw an Exception. The try block should also contain all statements which *depend on* the result of the server method.

If an Exception is thrown, control is immediately transferred to the catch blocks. The first catch block to correctly name the Exception is selected, and the statements in its block are executed. Control then falls through to whatever follows the try/catch statement. If none of the named Exceptions match the thrown Exception, this method throws the same Exception (the Exception is not handled here).

As an example, consider the following method which attempts to calculate the GPA of each student in a given List.

```
/**  @param min The minimum GPA needed to be on the dean's list
 *   @return A List of all students on the roster
 *   who qualify for the dean's list.
 *   Uses getGradePoints() and getCredits() to obtain data for
 *   a student.  Ignores students with non-valid data.
 */
public List<Student> deansList (List <Student> roster, double min)
{  List <Student> result = new LinkedList <Student> ();
   int credits, gradePoints;
   for (Student st : roster)
```

```
       {  credits = getCredits(st);
          gradePoints = getGradePoints(st);
          try {
                 st.calcGPA (gradePoints, credits);
                 if (st.getGPA() >= min)
                    result.add (st);
              }
          catch (IllegalArgumentException iae)
              {  System.err.println ("Illegal data for " + st);
                 System.err.println ("This student not processed");
              }
       }
}
```

In the `deansList` method shown above note that the `try` block contains all statements which depend on the result of the call to `calcGPA()`. In this example there is only one `catch` block, but if we knew that `calcGPA` threw other Exceptions we could include additional `catch` blocks. Also note that the try and catch blocks require the curly braces, even if there is only one statement in the block.

8.3.4.1 Checked and unchecked Exceptions

As with most classes in the Java class library, Exception classes form a hierarchy of sub-classes. This hierarchy is critical to understanding the requirements placed on a program by the compiler. The compiler will *require* handling certain Exceptions and will allow other Exceptions to be ignored.

Some selected Exception classes (and anything which can be thrown) from the Java class library are shown in Figure 8.4. In this diagram a `Throwable` is a super-class representing anything that can be thrown (at this point the only things we can throw are Exceptions). An `Error` represents an error in the Java class library or run-time system; it is safe to say that you will not encounter this error. This software has been thoroughly tested by millions of users for many years. If you do encounter this error, rest assured it is not your mistake, and you will have to find a work-around (report this to Oracle).

We should concern ourselves with the Exception class in Figure 8.4 and its sub-classes. These classes fall into two categories:

- Unchecked Exceptions – `RunTimeException` and all of its sub-classes are said to be *unchecked*.
- Checked Exceptions – All other sub-classes of `Exception` are said to be *checked*.

The difference between checked and unchecked Exceptions has to do with whether the compiler allows you to ignore them:

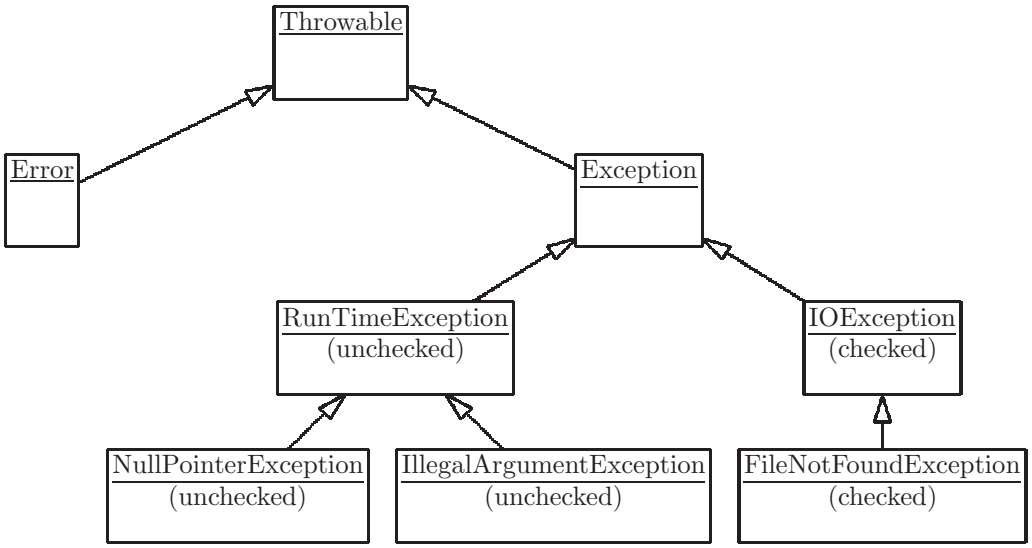


Figure 8.4: Class diagram showing some of the Exception classes in the Java class library

- Unchecked Exceptions – The client method is not required to check for any unchecked Exceptions that might be thrown by server methods. It is permissable to use a try/catch block, but not required.
- Checked Exceptions – The client method is required to check for any checked Exceptions that might be thrown by server methods. In this case, the client method has two choices:
  - Use a try/catch block to handle the Exception as discussed previously.
  - Declare, in the method signature, that the client method *throws* the Exception. In this case the method which called the client method is faced with the same requirement (we call this 'passing the buck'). The API of the client method should also have an @throws line describing the circumstances under which this Exception would be thrown.

Why would the client method wish to pass the buck rather than handling the Exception? It could be that the client method does not have enough information to handle the Exception appropriately, but the method which called it would be better able to handle the Exception. Thus Exceptions can be propagated from any depth all the way to your top-level method (the one that started everything at the beginning). If your top-level method throws an Exception, the Java runtime environment will catch it and your program will come to a crashing halt.

Methods which involve input and output are often subject to external constraints which are difficult, if possible, to deal with in a program. For example, when attempting to open a data file, the file may not exist, the disk might be full (when opening for output), or network problems may exist (for drives mapped to a network). This is why all sub-classes of IOException are checked Exceptions; the likelihood of an Exception occurring is so great that the developers of Java decided to force you to check for them. We will discuss I/O exceptions further in chapter 9.

As an example, reconsider our `deansList` method to return a List of all students from the given List who have a given minimum GPA. It called a few methods: `getGradePoints(Student)` and `getCredits(Student)`. Suppose these methods obtain information by reading data from a disk file; it is likely that they would throw some sort of (checked) IOException. If this is the case, we should see it described in the API, and the compiler would see the `throws` keyword in the signature:

```
/**  @param st A Student in our disk file of Students
 *   @return the number of credits earned by st.
 *   @throws IOException
 */
public int getGradePoints (Student st)
       throws IOException
{ //  Code to open a data file and find the number
  //  of grade points for the given Student...
}
```

Because this method specifies `throws IOException` in the signature, the compiler will force us to make a choice: handle the Exception right here in the deansList method or pass the buck to the calling method.

If we choose to handle the Exception here in the deansList method, it could be done as shown here with a try/catch:

```
/**  @param min The minimum GPA needed to be on the dean's list
 *   @return A List of all students on the roster
 *   who qualify for the dean's list.
 *   Uses getGradePoints() and getCredits() to obtain data for
 *   a student.
 */
public List<Student> deansList (List <Student> roster, double min)
{  List <Student> result = new LinkedList <Student> ();
   int credits, gradePoints;
   for (Student st : roster)
       {  try                // handle a possible IOException
            {  credits = getCredits(st);
               gradePoints = getGradePoints(st);
               st.calcGPA (gradePoints, credits);
```

```
               if (st.getGPA() >= min)
                    result.add (st);
             }
          catch (IOException ioe)
             {  System.err.println ("Unable to obtain data for " + st); }
       }
}
```

Note that the calls to `getCredits` and `getGradePoints` are in the try block, as well as all statements which need the results of those methods. It would make no sense to call `calcGPA` if we have failed to obtain the information which it needs.

Our other option is to pass the buck; i.e. notify the calling method that we are unable to handle this Exception so the calling method must decide what to do:

```
/**  @param min The minimum GPA needed to be on the dean's list
 *   @return A List of all students on the roster
 *   who qualify for the dean's list.
 *   Uses getGradePoints() and getCredits() to obtain data for
 *   a student.  Ignores students with non-valid data.
 *   @throws IOException if credits and/or grade points cannot
 *   be obtained from the data file.
 */
public List<Student> deansList (List <Student> roster, double min)
       throws IOException       // pass the buck
{  List <Student> result = new LinkedList <Student> ();
   int credits, gradePoints;
   for (Student st : roster)
       {  credits = getCredits(st);
          gradePoints = getGradePoints(st);
          st.calcGPA (gradePoints, credits);
          if (st.getGPA() >= min)
               result.add (st);
       }
}
```

Note that the `@throws` keyword in the API is for the benefit of people (the programmers who will be writing methods that call this method), and the `throws` keyword in the method signature is primarily for the compiler.

8.3.5 Defining your own Exception classes

Classes in the Java class library can be extended (sub-classed) by classes that we create. This is certainly true for Exception classes. When we define our own Exception class, we must decide which of the Exception classes in the Java class library is to be the super-class. If we wish our class to be an unchecked

Exception, it should extend RunTimeException, but if we wish our class to be a checked Exception, it should extend Exception (or some sub-class of Exception other than RunTimeException).

When doing so, we will use the existing Exception classes as models of good behavior, and design our classes in a similar way. In particular, we will have a few constructors: a default constructor with no parameters, and a constructor with a String parameter, to be displayed for the programmer of the client method.

Continuing with our Student example, suppose we wish to throw an Exception when a GPA cannot be calculated. We may wish to define our own Exception specifically for this error. If we wish the compiler to force checking for this Exception, we would want it to be a checked Exception, and therefore it could be a sub-class of Exception:

```
/** This Exception should be thrown when a Student's GPA cannot
 *  be calculated.
 *  Detailed message and the Student can be included, but are
 *  optional.
 */
public class BadGpaException extends Exception
{  private String message;     // error detail
   private Student student;    // Student with the problem

/** Construct a BadGpaException with null as its
 *  error detail message and null as its
 *  offending Student;
 */
   public BadGpaException()
{  }

/** Construct a BadGpaException with the given String as its
 *  error detail message, and the given Student as the
 *  offending Student.
 *  @param s Detaileds info on the error, or null if no info
 *           is available.
 *  @param st Student who caused the problem, or null if
 *           not available.
 */
   public BadGpaException(String msg, Student st)
{  message = msg;
   student = st;
}

/**
 *  @return the details of the error,
 *  which could be a null reference,
```

```
 *  if not available.
 */
public String getMessage()
{  return message;  }

/**
 *  @return the offending Student,
 *  which could be a null reference,
 *  if not available.
 */
public Student getStudent()
{  return student;  }

/**
 *  @return this BadGpaException as a String
 */
public String toString()
{  String result = "GPA could not be calculated";
   if (message != null)
      result = result + " because " + message;
   if (student != null)
      result = result + "\nCaused by " + student;
   return result;
}
}
```

We can now add the BadGpaException class to our class diagram; it is shown in Figure 8.5.

In the `BadGpaException` class note that:

- There are two fields in this class:
  - A String containing details on the cause of the Exception
  - A reference to the Student who caused the Exception to be thrown (the offending Student).
- There are two constructors in this class.
  - A constructor with no parameters (default constructor) which is used when the error details and the offending Student are unavailable. This constructor leaves both fields initialized to null references.
  - A constructor with two parameters which is used when either, or both, the error details and the offending Student are available. It initializes at least one of the two fields.
- If a method wishes to throw this Exception, there are a few ways this can be done:

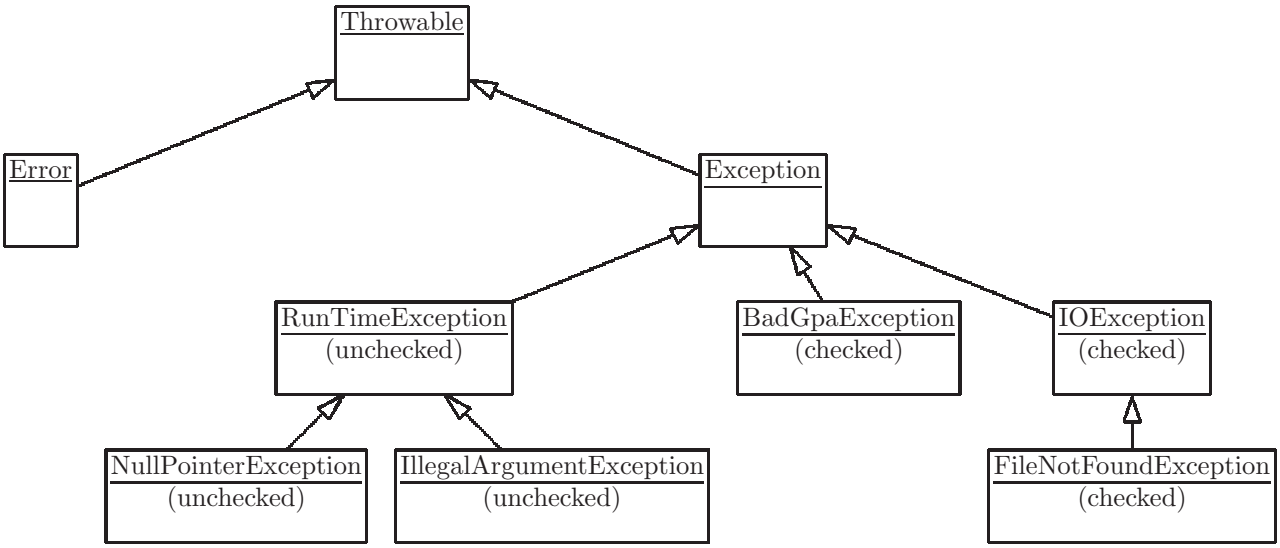


Figure 8.5: Class diagram showing some of the Exception classes in the Java class library with our own class, BadGpaException

- If it has no information on both the error details and the offending Student, it could throw an Exception using the default constructor:
  `throw new BadGpaException ( );`
- If it has information on both the error details and the offending Student, it would call the constructor with two parameters:
  `throw new BadGpaException ("No credits earned", someStudent);`
- If it has information on either the details of the error or the offending Student, but not both, it would call the constructor with two parameters, with `null` as a parameter:

```
// Offending Student not available
throw new BadGpaException ("Student not in database", null );

// Error details not available
throw new BadGpaException (null,someStudent);
```

8.3.6 Exercises

1. List a few java Exceptions not mentioned in this section.
2. Can a NullPointerException be created with a String as a parameter?
   `NullPointerException npe = new NullPointerException ("foo is null");`
   If so, what is the purpose of the parameter?

Hint: See the API for the package `java.lang`.

3. Refer to the `university` project in the repository for this chapter. Which of the following code segments will cause an Exception to be thrown, and if so, what is the class of the Exception?

   (a)
   ```
   Student frosh;
   frosh.calcGPA(12,3); // 12 grade points
   System.out.println (frosh);
   ```

   (b)
   ```
   Student frosh = new Student ("jim", "232-34-3333");
   frosh.calcGPA(12,0); // 12 grade points
   System.out.println (frosh);
   ```

4. In the `Student` class of the `university` project in the repository for this chapter, there are accessor methods for a Student's name and ssn. Make the following modifications to these accessor methods, and include an explanation for the reason the that it is being thrown (don't forget to update the API).

   (a) Modify the `getName()` method so that it will throw an `IllegalStateException` if the name is a null reference.

   (b) Modify the `getSSN()` method so that it will throw an `IllegalStateException` if the ssn is not valid. The format of an ssn should be "999-99-9999", where the "9" represents any numeric digit.

5. Use the `university` project in the repository for this chapter. In the `UniversityInfoSys` class define a method named `buildPassword` with one parameter, a `Student`. The method should return an initial password for the given Student consisting of the first initial of the Student's name and the last two digits of the Student's ssn. If one of the server methods throws an Exception, it should be caught and handled in the `buildPassword` such that:

   - An error message is sent to stderr (System.err.print...)
   - Execution continues; the program does not crash.

   ```
   /** @return An inital password for the given Student,
    *  consisting of first initial of name and last two digits
    *  of ssn.
    *  Execution continues if an IllegalArgumentException is thrown,
    *  with a message sent to stderr.
    */
   public String buildPassword (Student st)
   ```

8.4 Debuggers

A software tool which is helpful in finding and fixing programming errors is called a *debugger*. The debugger will not help you with compile-time errors; for these you must rely on the error message provided by the compiler. But for complex run-time errors, which may be buried deep in several nested method calls, or nested loops, a good debugger is essential. There may be some disagreement over the value of a debugger versus *code review*. Code review, recommended by many software engineers, is the process of examining the program carefully, either alone or with other programmers, to make sure that every possible problem or error is avoided. We agree that code review is valuable, but there always seem to be bugs which slip through after the most careful code review. For these, one needs to have the necessary skills to use the debugger effectively.

Each Integrated Development Environment (IDE) has its own debugging tool with a user interface. These user interfaces may appear different, and somewhat daunting, at first. Here we attempt to describe some of the features available in most debuggers, without going into the specific details of any one particular debugger. We stress that the debugger is used to find the line(s) of source code which are causing erroneous output, a run-time exception, or an invalid program state. Note that a debugger will NOT help with:

- A debugger will not help you with compile-time errors. The compiler will provide an error message and line number in the source file, either of which might be accurate or helpful.
- A debugger will not help you discover that a flaw in the program exists. For this we use code review and thorough testing.
- A debugger will not automatically show you where the bug is; it is simply a tool which will help you find the bug.
- A debugger will not fix the bug for you. A good understanding of the program logic is necessary to fix the bug yourself.

The features of a typical debugger include:

- Single Step (or Step) – Step through the statements of the program one line at a time. Click a button to allow the next line to be executed. There are usually two ways to do this:
  - Step Over – Execute the current line. If the line contains a method call, run the called method at full speed before returning control to the debugger, and move on to the next line of code.
  - Step Into – If the current line contains a method call, step into that method and return control to the debugger to execute the called method one line at a time.

- Watch variables – Show the value of selected variables as the program executes. Many debuggers show all active variables (local variables, instance variables, class variables) by default. Many debuggers will also show you the call stack – the sequence of method calls which led to the current line.

- Set Breakpoint – Any executable line may be set with a breakpoint. When running a method at full speed, execution pauses when encountering a breakpoint. Control is returned to the debugger so that one can single-step from that point.

- Continue – Continue executing at full speed, pausing at the next breakpoint, if any.

- Conditional commands – Choose one of the options shown above conditionally, depending on the value(s) of variable(s). This feature is available only with the more advanced debuggers.

- Step backwards – If you have gone past the bug and do not wish to start over from the beginning, you can step to the previous line. This feature is available only with the more advanced debuggers (notably in the Eclipse IDE).

We cannot overemphasize the importance of learning to use a debugger skillfully. Once you have experience with one particular debugger, it is relatively easy to learn a different debugger.

8.4.1 Exercises

1. Briefly describe the difference between the meaning of *step into* and *step over* for a typical debugger.

2. True or False: A Debugger is a tool which can be used to patch a defective program, after the programmer has determined which statement is at fault.

3. Consider the following code segment:

```
for (int i=0; i<100; i++)
  for (int j=0; j<10000; j++)
     System.out.println (i*j - i/j);
```

   Enter this code into a method in a new class, and execute it with your debugger. Step through the statements and pause when the value of i is 25 and the value of j is 10.

8.5 Debugging with print statements

One of the oldest and most obvious debugging techniques involves the use of print statements. Insert print statements at various points in the program to make sure that variables contain correct values. This is sometimes easier than using a debugger, but if your print statements are in nested loops, you could get large quantities of output, making it difficult to search for useful information. The following print statement assures the programmer that the value of the variable `credits` contains the correct value, 45:

```
System.out.println ("credits should be 45, credits = " + credits);
```

8.6 Projects

1. Use the project `university-debug` in the repository. In this project the class `UniversityInfoSys` is responsible for providing information on courses and students enrolled. The `Student` class has been enhanced to store a collection of courses in which the Student has enrolled. You should start out by viewing the API (Documentation View) for each of the classes.

   This project compiles without errors (ignoring the class `Driver2`), but it contains at least 6 run-time errors. Some of these will cause the program to crash, and some will cause incorrect output. Correct all run-time errors (they will be found in the Student and UniversityInfoSys classes only); it may be helpful to use a debugger. When you test your solution using the `main` method in the `Driver1` class, the output should look like the output in the plain text file `university-debug-output1.txt`. (Do not try to compile the class `Driver2`; you could temporarily remove it from the project. It is for the next problem.)

   Hint: Your debugger should be able to distinguish between local variables and fields.

2. Use the same project that you used in the above problem; here you will test with the `Driver2` class.

   In this problem a Student's ssn must be in the format '999-99-9999', i.e. all characters must be numeric (0-9) except for the two dashes, and the dashes must be in positions 3 and 6.

   Define an Exception named `InvalidSSNException`. This Exception will be thrown when a Student is given an invalid ssn. We will handle the exception by attempting to put the ssn in the correct format, if possible. It that is not possible, we will use the given ssn, print an error message, and continue processing. Be sure that your `InvalidSSNException` class has the appropriate javadoc comments for a useful API. It will be necessary to make some changes to the `Student` class.

   The output should match what is shown in the plain text file `university-debug-output2.txt`.

Chapter 9

Console Applications – Input and Output

Most programs need to communicate, in some way, with the world outside. They may need information from some external source, or they may need to communicate results to an external device. In this chapter we examine some common operations which are used to bring data into an executing program (input) and operations used to send data out from an executing program (output).

The terms *input* and *output* are used from the point of view of the primary memory (RAM). The input operation brings data into RAM from some external source such as a disk, a USB port, a keyboard, or a network port. The output operation sends data out from RAM to some external device such as a disk, a USB port, a monitor, a printer, or a network port. Figure 9.1 shows a simple diagram of these hardware components, with arrows showing the direction of data transfer for input and output.

This chapter will discuss these input and output operations for *console* applications only. A console application is one in which the data being input, or put out, for the user is plain text (i.e. ASCII or Unicode characters). In communicating with the user, console applications cannot use graphics of any kind, including graphical user interfaces. Nor do console applications normally utilize images or sound.

9.1 Standard io files

Most systems which accommodate console applications make use of *standard IO* files. These are somewhat abstract concepts that represent the source or destination of data being input or put out, respectively. Standard IO is usually abbreviated `stdio`, and there are three such files which we discuss here:

- `stdin` – The standard input file. Normally this refers to the user's keyboard. When we read from from `stdin` we are obtaining plain text that

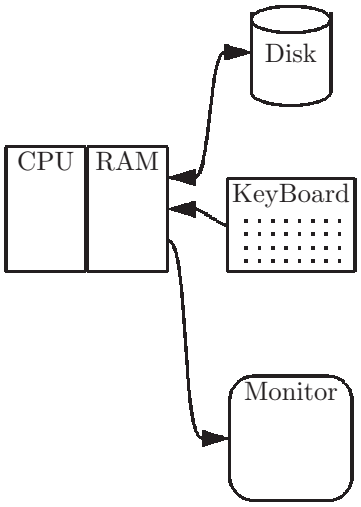


Figure 9.1: Simplified diagram of a computer with peripheral devices

the user types on the keyboard.

- `stdout` – The standard output file. Normally this refers to the user's monitor (or terminal window in an IDE such as BlueJ). When we print to `stdout`, we are sending plain text to the user's monitor.
- `stderr` – The standard error file. Similar to `stdout`, `stderr` is normally used for error messages directed to the user. When running an application from the system command line, text sent to `stdout` will not be distinguished from text sent to `stderr`. Some IDEs, such as BlueJ, will show `stdout` and `stderr` in separate windows, with the `stderr` output in red.

The user can redirect any, or all three, of these standard files to some other source or destination.

9.1.1 Exercises

1. True or false:
   (a) Both `stderr` and `stdout` are initially directed to the user's monitor but can be redirected to some other destination.
   (b) `stdin` is initially directed to the user's keyboard but can be redirected to some other source.
   (c) Console applications may utilize images or sound to improve communication with the user.
2. Is a printer an input device, or does it perform output as well?
3. Is a scanner primarily an input or output device?

9.2 Output to stdout or stderr

In this section we explain how to send plain text to the user, on `stdout` or `stderr`. Normally the output will be sent to the user's monitor, but it can be redirected to some other destination such as a disk or USB port.

9.2.1 Output to stdout

One way to send text to `stdout` we've already seen:

```
System.out.println("some message");.
```

`System` is a class in the package `java.lang` which does not need to be imported. `System` has other interesting features, but we will be using it for output of plain text to `stdout`. When the `println` method is executed, its parameter, which is a String, is sent to `stdout`, and the user is able to see the output. If the parameter of println is not a String, it is automatically converted to String via a call to the toString() method. All primitive types can be converted to String automatically by the run-time system, and all reference types inherit a toString() method from Object if they don't define one. The toString() method inherited from Object is probably not going to do what you want. It will call the hashCode method on the object, and convert the result to hexadecimal (base 16 characters), concatenated with the name of its class. If you see something like `Student@148ae300` on your output screen, you are probably missing a toString() method in your Student class.

9.2.2 Output to stderr

In order to send plain text to `stderr`, we simply use `err` instead of `out` in the call to the println method. This form is intended to be used for error messages, such as:

```
System.err.println ("Number of credits should not be negative");
```

9.2.3 Exercises

1. Which IDE do you use? When executing a program does it show `stdout` and stderr in separate windows?

2. Write a simple java method to write some text to `stdout` and some to `stderr`. Compile and test this method from the system command line. Are the output lines of `stdout` separate from the output lines of `stderr`?

   Hint:

   (a) Open a terminal window (or cmd window).

   (b) Move to the directory in which you wish to work.

   (c) Define a class named `Test` in a text file named `Test.java`. It should have a method:

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

(d) Compile your class by issuing the command:
`javac Test.java`

(e) If there are no compilation errors, run the main method in the class by issuing the command:
`java Test`

3. Only Strings can be printed to `stdout` or `stderr`. Explain how it is possible to print something that is not a String:
`System.out.println (new Student ("jim", "123"));`

9.3 Input from stdin

Input is the process of transferring information from an external source such as disk, keyboard, or USB port into the computer's RAM as a program executes. We can use the standard input file, `stdin`, which is normally directed to the user's keyboard. There are several ways of accomplishing this; one of the easiest and most powerful tools for input is the `Scanner` class which is in the `java.util` package. Do not let the huge API for this class scare you away; it has many powerful features, but we will be using a few of the easier features.

In order to use Scanner it must be imported at the beginning of the class:
`import java.util.Scanner` or `import java.util.*`
When instantiating a Scanner object, you can provide it with the source of the input. In this section we wish to read from `stdin`, i.e. the user's keyboard. Consequently we identify this as `System.in`:

```
Scanner scan = new Scanner (System.in);
```

We can now apply any of the methods in the `Scanner` class to this object, which we have named `scan`. The two methods from `Scanner` which we introduce now are:

- `public boolean hasNextInt();` – This method will return true only if an int has been entered on the input stream (System.in in our case).
- `public int nextInt();` – This method will return an int, if possible, from the input stream.

As an example, we could implement a method in our `Student` class to get the number of credits from the user's keyboard:

```
/**  Get the number of credits for the given Student
 *   from stdin.
 *   @return The number of credits read from stdin.
 */
public int getCredits (Student st)
{  int result = 0;

   // Prompt user for input
```

```
    System.out.println ("Enter the number of credits for " + st);

    if (scan.hasNextInt())
        result = nextInt();
    else
        System.err.println ("Invalid number entered, 0 is assumed");
    return result;
}
```

Note that this method first prompts the user to enter something on the keyboard. If you forget this prompt, the user will not know that the computer is waiting for input; the user and computer will sit there waiting for each other to do something. Users need to be prompted for action, and they need to be told what the program is expecting to be entered.

The call to `hasNextInt()` is recommended, so that if the user enters something which is not a valid int, the program will not crash with an IOException.

9.3.1 Exercises

1. Which method in the Scanner class can be used to read the next valid double precsion number from the input stream?

2. Which method in the Scanner class can be used to determine whether there is a valid double precsion number at the front of the input stream?

3. Experiment to see what happens when a program tries to get a double precision number from the input stream, and the front of the input stream is not a valid double precision number.

4. Write a method to prompt the user to enter three whole numbers at the keyboard. Your method should return the average of the three numbers.

   ```
   /** @return the average of three whole numbers entered
    *  on the user's keyboard */
   public double average3 ()
   ```

9.4 Data Files

Large quantities of data can be stored permanently on external or internal storage devices such as disks and USB flash memory. In this section we will see how to input data from storage, and how to put out data to storage, from an executing program. Before a data file can be accessed, it must be *opened*. Once it has been opened, it is possible to read data from the file, append data to the file, or overwrite existing data with new data to the file.

9.4.1 Opening a data file

Before any input or output can occur, a data file must be opened. This is the point where the program requests a service from the operating system:

- I will tell you whether I will be using it for input or for output.
- If using it for input, can this file be found in the folder where I think it exists?
- If using it for output, can this file be created in a particular folder?
- Do I have permission to use this file?

A file can opened by instantiating an appropriate class from the Java class library; this will typically be an instance of the File class for input, or the FileWriter class for output; the constructor is provided with the name of the file to be opened:

```
new File("myInputFile.txt");
new FileWriter("myOutputFile.txt");
```

More complete examples are given in the following sections.

9.4.2 Input from Data Files

Input is the process of transferring information from an external source, such as a disk or USB flash memory, into RAM for an executing program. We will use the same class that we used for keyboard input: `Scanner`. To open a file for input, we will instantiate a `Scanner` object using an instance of the `File` class. `File` is in the `java.io` package so it will need to be imported:

`import java.io.File;` or `import java.io.*;`

We will also need to import the Scanner class:

`import java.util.Scanner` or `import java.util.*;`

We can now open a file for input by instantiating Scanner:

```
Scanner scan = new Scanner (new File("myFile.txt"));
```

Notice that we are careful to provide `Scanner` with a `File`. If we had omitted the `new File` and simply provided `Scanner` with the file name as a String, we would have been calling a different constructor in the `Scanner` class – Scanners can also read from Strings, and that is not what we are interested in doing here.

As shown above, however, the compiler will not accept this. If you look at the API for the constructor in the `File` class you will see that it *throws* IOException. The API goes on to explain that if the file cannot be found or cannot be opened for some other reason, an `IOException` will be thrown. Recall from chapter **??** (Figure 8.4) that `IOException` is a *checked* exception, which means that the client method (i.e. the calling method) must do one of the following:

- Handle the Exception with a try/catch statement.

- Declare that this method *also* throws IOException (i.e. pass the buck).

If you omit both of the above, the compiler will remind you of your choices:
`Unreported Exception java.io.IOException; must be caught or declared to be thrown`
Once the file has been opened for input, we can use the `Scanner` object to read data just as if we were reading from the user's keyboard. In addition to `hasNextInt()` and `nextInt()`, the `Scanner` class has methods which will read an entire line as a String: `hasNextLine()` and `nextLine()`.

When finished reading data from a file, we should close it. If we forget to close it, the system will close it when the program terminates; however we should close it anyway. This is a good habit to get into; good citizens always close their files when finished:
`scan.close();`
We can now write code which will read all the lines in a text file and print them on `stdout`:

```
import java.util.*;             // Scanner
import java.io.*;               // File
...

String line;
// Open the file roster.txt for input
try {
       Scanner scan = new Scanner (new File ("roster.txt"));

       while (scan.hasNextLine())
       {  line = scan.nextLine();   // Read a line from the file
          System.out.println (line);
       }

       scan.close();                  // Good citizens close files
     }
catch (IOException ioe)
    {  System.err.println ("File roster.text not found"); }
```

Notice that we include in the try block not only the call to the File constructor, but everything which depends on its successful termination as well.

9.4.3 Output to data files

There are many ways of writing output to a data file. We will discuss one of the simpler techniques. Though it is possible to write binary data to a file, we will be writing plain text. We will need the `FileWriter` class which is in the `java.io` package:
`import java.io.FileWriter;` or `import java.io.*;`

Though it is possible to append data to an existing file, we will simply write to a file, assuming that if it does not exist, it will be created, and if it does exist it will be overwritten. To open the file for output, instantiate the `FileWriter` class, by giving it the name of the file as a String:

```
FileWriter writer = new FileWriter("myFile.txt");
```

Like the `File` class, the `FileWriter` constructor also throws IOException, so we enclose its instantiation in a try block.

The example above presumes that the file to be opened is in the same directory (i.e. the same folder) as the Java source file. If it is located somewhere else, a full path description of its location is needed:

```
FileWriter writer = new FileWriter("c:myStuff/myFile.txt");
```

After opening the file, we can write Strings to it using the `write (String)` method:

```
writer.write("any string...");
```

Each time this method is executed a String will be written to the file. Note that if you want the Strings stored on separate lines, you must include a newline character, `'\n'` at the end:

```
writer.write("any string...\n");
```

Finally, when finished writing data to the file, it should be closed. Most systems will automatically close a file when the program terminates, but you should close it anyway. This is a good habit to get into; all good citizens close their files when finished using them:

```
writer.close (); // good citizens close files
```

The following code segment will open a file named `roster.txt` for output, and write data for two students to that file:

```
import java.io.*;
...
try {
       FileWriter writer = new FileWriter ("roster.txt");
       writer.write ("joe\n");            // name
       writer.write ("234-54-9498\n");    // ssn
       writer.write ("3.5\n");            // GPA

       writer.write ("jim\n");            // name
       writer.write ("432-45-8949\n");    // ssn
       writer.write ("0.0\n");            // GPA

       writer.close();                    // good citizens close files
    }
catch (IOException ioe)
    {  System.err.println ("File not found:  roster.txt"); }
```

9.4.4 Exercises

1. Under what circumstances might an attempt to open a file result in the throwing of an Exception?

2. Define a method named `createFile` with one parameter which will accept input from stdin and store the lines in a file with the given name.

```
/** Open the given file for output.  Obtain lines of text from stdin,
 *  and write them to the given file.
 *  @param filename Name (and path) of the text file to be created.
 */
   public void createFile (String filename)
```

   When testing your solution, use ctrl-d (Unix) or ctrl-z (Windows) to terminate the input.

3. Define a method named `copyFile` with two parameters which will copy all the lines from a text file into a new text file.

```
/** Open source file for input and open target file for output.
 *  Copy all lines of source file to target file and close.
 *  Pre: Source file is a text file.
 *  @param sourceFilename Name (and path) of the text file to be copied.
 *  @param targetFilename Name (and path) of the text file to be created.
 */
   public void copyFile (String sourceFilename, String targetFilename)
```

   Test your program by using it to copy the source file to a new file.

4.

9.5 Running an Application from the Command Line

As we develop software in Java, we generally use an Interctive Development Environment (IDE), such as BlueJ, Eclipse, or NetBeans. The IDE provides a convenient way to edit, compile, test, debug, etc. Many IDEs will provide information on a project's classes as well as a nice-looking API. Some IDEs, such as BlueJ, allow the developer to execute an individual class or instance method when testing a particular class (BlueJ also allows direct execution of a single Java statement with a CodePad feature).

9.5.1 Compile and Test from the Command Line

However, a Java application can also be invoked from the Unix or DOS command line. Follow these steps to create your source file(s), compile, and test your application:

1. There should be a source file (.java) for each class in your project. The name of the file should be the same as the name of its class. These files will normally all be in the same directory (folder), and they should be plain text files, created and edited with a line editor such as vi, emacs, edlin, or notepad.

2. One of the classes in your project, the *main* class, should have a `main` method which starts the application:

   ```
   public static void main (String [] args) { ...  }
   ```

   This is the method which will be called when execution begins, and is explained in the next section.

3. When ready to compile these source files, they can be compiled individually, or all at the same time:

   - `javac ClassName.java`
   - `javac *.java`

4. If there are error messages from the compiler, use your editor to correct the errors, save, and recompile.

5. If there are no error messages from the compiler, you will notice that there is a .class file in the project's directory for each class in the project which has been compiled. These files contain *byte code* which can be directly executed by the Java RunTime Environment. To execute, you must invoke the Java runtime environment from the command line:

   ```
   java MainClassName parm1 parm2 parm3 ...
   ```

   The parameters, parm1, parm2, parm3, ... are not required, but they should correspond to the array of Strings (args) in the declaration of the main method. Note that there is no .java suffix (or any suffix) on the class name.

9.5.2 public static void main (String [ ] args)

Here we explain why the main method must be declared as shown in the previous section. We dissect this declaration and explain each part below:

- `public` – Since this method will be invoked from outside the class, access must be public.

- `static` – The main method must be a class method; it cannot be an instance method because there are as yet no instances of that class. The runtime environment is assuming it will be a class method and will invoke it as:

  ```
  MainClassName.main (...)
  ```

- `void` – The runtime environment does not make use of a returned value, and requires that the main method be a `void` method.

- `main` – Since the runtime environment is calling a method named `main`, that is what the name of your method must be.

- `String` – The runtime environment calls your main method with one actual parameter; thus your main method must have one formal parameter.

- `[ ]` – The parameter is an array of Strings.

- `args` – The parameter name can be any valid Java name, but most people use `args` which is short for *arguments*. The term 'argument' is synonymous with 'actual parameter' and is carried over from older languages such as C++ which have functions.

The command line arguments correspond to the items in the array of Strings which is the main method's formal parameter. If the runtime environment is invoked as shown below:

```
java MyClass sam joe bill
```

then when the main method starts up, the value of its parameter will be:

```
args.length = 3
args[0] = "sam"
args[1] = "joe"
args[2] = "bill"
```

If you supply too many arguments on the command line, your program simply ignores the extra arguments. However, if you enter too few arguments on the command line, you could get an `IndexOutOfBoundsException`.

9.5.3 Exercises

1. Choose one of the methods you defined in the exercises of the prevous section, and modify its class so that it can be executed from the command line of a terminal window.

2. Define a command named Alphabetic which will determine whether its three arguments are in alphbetic order. It should print either "in order" or "not in order". For example:

```
> java Alphabetic alpha gamma beta
not in order
> java Alphabetic alpha beta gamma
in order
```

3.

4.

9.6 Projects

1. A classic data processing task involves the *merging* of two or more ordered data files. The output file contains all the data of the input files and is also ordered.

   In this project we wish to merge two text files, each of which contains a single word on each line. The words are in alphabetic ordeer. We wish to create a file containing all the words in both files (duplicates are ok) in alphabetic order.

   Use the project `fileMerger` from the code repository. Define a class named `Merger` in which you will define a method with three parameters named `merge`. The parameters are the names of the two input files and the name of the output file to be created.

   ```
   /**
   * Pre: file1 and file2 are text files in alphabetic order.
   * @param result is the name of the file produced by merging
   *  file1 with file2.
   * The result is also in alphabetic order.
   */
   public void merge (String file1, String file2, String result)
   ```

   Test your solution by using the data files provided in the code repository, `file1.txt` and `file2.txt`.

2. Java (and many other programming languages) have two kinds of comments:

   - Single-line comments: Begin with // and extend to the end of the line
   - Multi-line comments: Begin with /* and end with */

   In a new project create a class named `Comments`. In this class define two methods:

   - A method named `countSingleLineComments` which will count all the comments beginning with // in a java source file.

```
/** @return number of single-line comments in the
*  given java source file
*/
public int countSingleLineComments (String filename)
```

- A method named `countMultiLineComments` which will count all the comments beginning with /* and ending with */ in a java source file.

```
/** @return number of multi-line comments in the
 *  given java source file
 */
public int countMultiLineComments (String filename)
```

Be careful:

- The following counts as two multi-line comments:
  `/* hi */ /* there */`
- The following counts as one single-line comment:
  `// hi there /* today */`
- The following counts as one multi-line comment:

```
/*  This is a // multiline
    comment
*/
```

- The following counts as one multi-line comment:

```
/*  This is a /* multiline
    comment
****/
```

Hint: Define one method which returns both values as a `List<Integer>` of size 2. Your method will behave like a *state machine* as it scans the characters of the source file:

- It will initially be in a *default* state (not inside a comment).
- When it sees a single slash, it will be in the *slash* state.
- When it sees // it will be in *single-line* state, and will remain in that state until it reaches the end of the line, at which point it will return to the default state.
- When it sees /* it will be in *multi-line* state, and will remain in that state until it reaches */ at which point it will return to the default state.

This state machine is shown in Figure 9.2. The circles represent states, and the labels on the arcs represent input characters (n represents a newline character, and A represents any character other than /, *, or newline). As each input symbol is scanned, the state of the machine changes according to the arrow. The machine starts out in the default state (DEF).

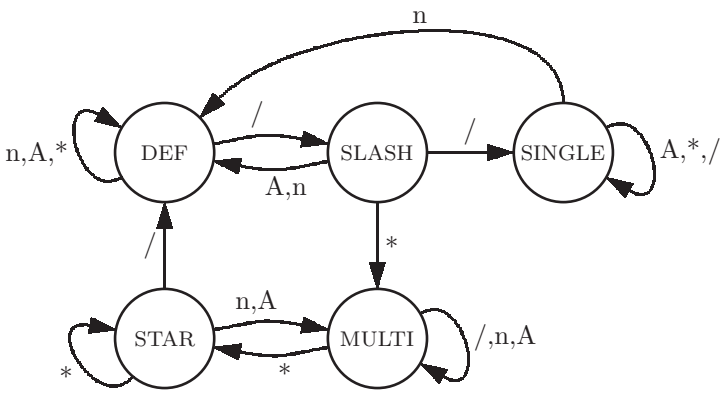


Figure 9.2: State machine to process java comments. `n` represents a newline character. `A` represents any input character except newline, *, and /. Start at the default state (DEF).

Chapter 10

Graphical User Interfaces

In the early days of computing (1945-1965) large mainframe computers executed programs in *batch* mode. Programs (and data) were submitted via punched cards or paper tape. At some later time, paper output would be available to the user. There was no interaction with the user as the program executed.

This was followed (1965-1980) by time-sharing systems which allowed users to provide input, and read output, via electric typing machines (often teletype machines, or IBM Selectric typewriters). Users were now able to interact with programs at execution time. Users could manage files and launch programs by typing commands to the operating system at the *console*. Such *command line* user interfaces are still in use today (Unix, DOS, e.g.).

In the mid 1970's researchers at the XEROX Palo Alto Research Center (PARC) experimented with a new user interface in which a pointing device (a mouse) was used to select icons on a monitor. XEROX did not pursue this effort, but the Apple corporation had recently developed micro-computers to compete with the IBM PC. Apple took PARC's idea, and produced the Macintosh computer with a *graphical user interface* (GUI) using a desk-top metaphor:

- Images of folders represented directories
- Copying of files was done by dragging items with the mouse
- Programs were launched by clicking on filenames with the mouse
- Files were deleted by dragging them to a trash can

Thus the GUI revolution was born; this was clearly a better way for the average user to communicate with an operating system. This user interface is sufficiently intuitive that novice users spent little, if any, time reading manuals. Apple's leading competitor in software, Microsoft, was compelled to follow suit or be left in the dust; a GUI for DOS, called Windows, was the result.

Today, though we still have command-line interfaces for operating systems, console applications are rare. Any software which expects user interaction will

have a graphical user interface. In this chapter we expose some of the Java classes which can be used to provide an application with a GUI.

10.1 Packages java.awt and javax.swing

Related classes can be grouped together in a *package*. There are primarily two such packages which can be used to build a graphical user interface:

- `java.awt` – This is the original package used for building a GUI. Some of its classes have been replaced by a better version in javax.swing.
- `javax.swing` – This package includes newer versions of some of the classes in java.awt, and other classes not found in java.awt.

Since many of the classes in java.awt are still considered usable, and some have been replaced by newer versions in javax.swing, we will need to import from both of these packages. The easiest way to do this is:

```
import java.awt.*;      // all the old classes
import javax.swing.*;   // all the newer classes
```

We do this at the risk of introducing name conflicts with classes imported from java.util or other sources (more on that later). A quick perusal of the java.awt package API shows that we have classes for some common GUI components, such as `Button`, `TextField`, `CheckBox`, `Frame`, and `Color`. These components will allow the user to communicate with an application; think of a Frame as being like a window frame, in which our GUI components will reside.

Looking at the javax.swing package, we see similar classes: `JButton`, `JTextField`, `JCheckBox`, and `JFrame`. However, there is no `JColor` class. The designers of javax.swing felt that the Button, TextField, and Frame classes needed to be rebuilt from scratch, but there was no need to make changes to the Color class, as depicted in Figure 10.1. In general, classes in javax.swing which replace older classes from java.awt begin with a J. Conceivably, we could build a GUI using java.awt only; however, we wish to be more up-to-date, so we will be using swing classes whenever possible.

10.1.1 Exercises

1. Which of the following classes from java.awt have been updated in javax.swing?

   BorderLayout, Menu, Label, Insets, Applet

2. (a) Why did Apple file a lawsuit against MicroSoft and Hewlett-Packard in 1988?
   (b) Which company filed a lawsuit against Apple at the same time and for the same reason?
   (c) What were the outcomes of those lawsuits?

Class in awt	Updated version in swing
Button	JButton
TextField	JTextField
Frame	JFrame
CheckBox	JCheckBox
Color	——
——	ImageIcon

Figure 10.1: Classes in package java.awt and updated versions, if present, in package javax.swing

Figure 10.2: A simple frame with a title and contentPane

10.2 Starting out: Frame and ContentPane

A GUI will generally consist of at least one *Frame*. A Frame is the basic structure from which a window may be constructed. Since there is an updated version, `JFrame`, in javax.swing, we will use the updated version. Looking at the constructor for JFrame, we see that it can have a String as its parameter; this is the title of the JFrame. A frame also has a *contentPane*. This is the part of the frame which can store the components (buttons, textfields, etc) of the frame. The contentPane is a *Container*, which is simply a general class which contains zero or more components. A simple frame, with title and contentPane, are shown in Figure 10.2. The appearance of the frame can vary, depending on the host platform (MacOS vs Windows vs Android, etc.).

To expose the various elements of a GUI we will use our Student class and develop a simplified information system for a typical university. All interaction with the user of this informaion system will take place through the GUI. Below we show the inital structure of the GUI:

```
public class UniversityGUI
{
   private JFrame frame;

   public UniversityGUI()
   {
     frame = new JFrame ("State U");   // title
     makeFrame();
   }
```

```
   // initialize the frame
   private void makeFrame()
   {
      frame.setVisible(true);         // default is false
    }
}
```

At this point our class has one field, `frame`. We instantiate it in the constructor, then call `makeFrame()` to initialize the frame (this is done in a private helper method rather than in the constructor itself because we will be adding more to it later). Note that frames are initially invisible, and we must set the visibility to `true` if we wish the user to see the frame.

If you try this yourself, you will see a small frame, perhaps in the upper left corner of your screen; it may not display the full title. To remedy this we can set the size of the frame with the `setSize(int width, int height)` method, for example:

```
frame.setSize(200,100);
```

This will set the width of the frame to 200 pixels and the height of the frame to 100 pixels. A *pixel* is a picture element; it is the fundamental (atomic, or indivisibile) unit of a graphics display. To understand the meaning of 'pixel' examine a photograph in your newspaper very closely. You will see that it is composed of many small dots; each dot can be considered a pixel.

We now wish to access the frame's contentPane, and add components to the contentPane. An example of a component would be a *label* which merely displays some text (or picture) on the frame. In order to work with the contentPane we will use an instance variable, `contentPane`, a Container which can be obtained from the frame:

```
contentPane = frame.getContentPane();
```

In our `makeFrame()` method we can now add one or more components to the contentPane:

```
contentPane.add(new JLabel ("I am a label"));
```

Our GUI class now looks like this:

```
public class UniversityGUI
{
   private JFrame frame;
   private Container contentPane;

   public UniversityGUI()
   {
     frame = new JFrame ("State U");            // title
     contentPane = frame.getContentPane();
     makeFrame();
   }

   // initialize the frame
```

```
   private void makeFrame()
   {
      contentPane.add (new JLabel ("I am a label"));
      frame.setSize(200,100);
      frame.setVisible(true);                // default is false
    }
}
```

10.2.1 Exercises

1. Point out the syntax errors, or possible run-time errors, if any, in the following statements:

   (a) `Frame frame = new JFrame("State U");`

   (b) `JFrame frame = new JFrame();`

   (c)
   ```
   JFrame frame = new JFrame ("Sate U");
   Container contentPane;
   contentPane.add (new JLabel("label"));
   contentPane = frame.getContentPane();
   ```

2. (a) In a new project define a class named `GUI` which creates a GUI for which the title of the frame is "Exercise". Add a textfield to this frame. The initial width of the frame should be 100 pixels and the initial height of the frame should be 200 pixels. Instantiate the GUI to see that it displays properly.

   (b) Experiment to see what happens if you add a button, in addition to the textfield.

10.3 Adding components to a container

JFrame!adding components Component, in a JFrame

10.3.1 Designing the GUI

We now wish to give our information system some functionality; we wish to admit students to our university. The user should have the capability of:

- Admitting students to the university
- Displaying all students currently admitted
- Searching (by name or by ssn) for a particular student

We propose using a GUI which looks like the one in Figure 10.3. The frame for this GUI has the following features:

- The title 'State U'

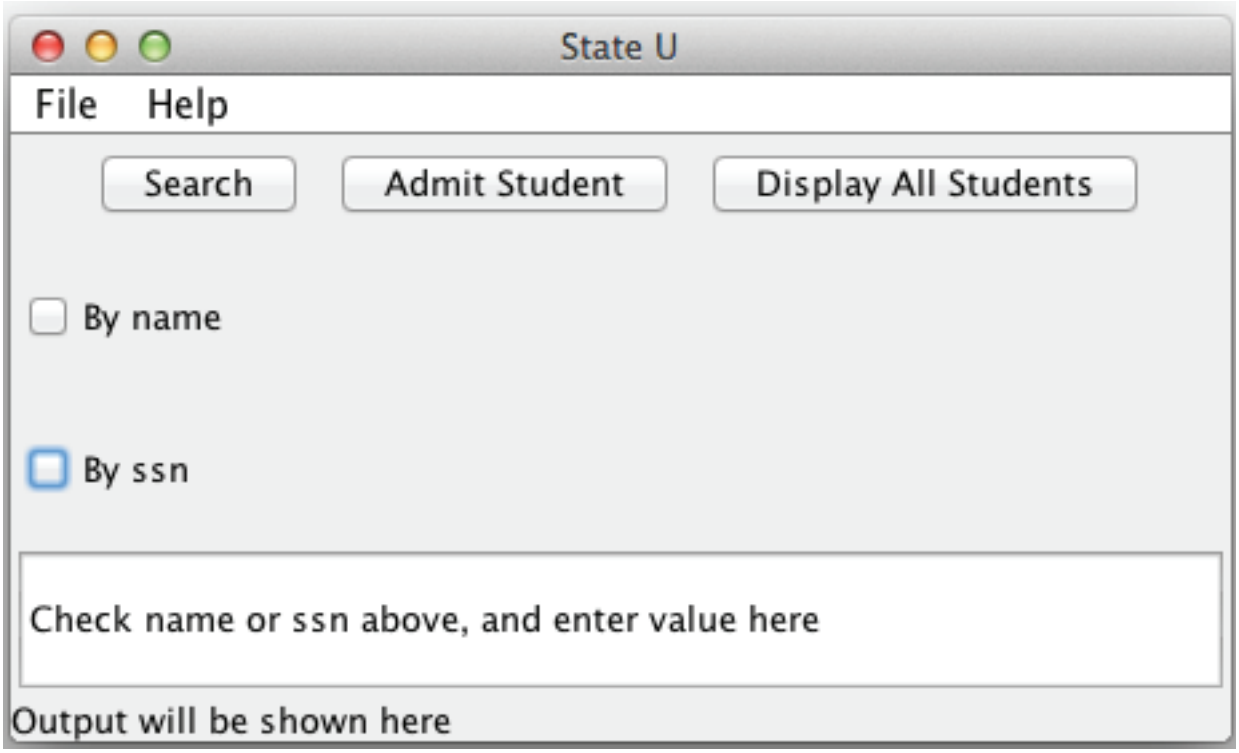


Figure 10.3: Proposed GUI for the university information system

- Three buttons:
  - Search – search for a particular student
  - Admit Student – Allow the user to enter the new student's name and ssn
  - Display All Students
- Check boxes to search by name, or by ssn
- A text field to allow the user to enter a search string (name or ssn)
- A label showing where the output of a search or a display will be shown.

10.3.2 Adding components

Since the contentPane is a Container, we can add components to it. We will now remove the label from the contentPane, and add a few buttons, as shown at the top of Figure 10.3. We can create a button, with text often called a 'caption', showing its purpose:

```
JButton someButton = new JButton("Caption");
```

We then add the button to the contentPane:

```
contentPane.add(someButton);
```

For our university GUI, the `makeFrame()` method would now be:

```
// initialize the frame
private void makeFrame()
{
   JButton searchButton = new JButton("Search");
   contentPane.add (searchButton);
   JButton admitButton = new JButton("Admit Student");
   contentPane.add (admitButton);
   JButton displayButton = new JButton("Display All Students");
   contentPane.add (displayButton);

   frame.setSize(200,100);
   frame.setVisible(true);                 // default is false
 }
}
```

When you instantiate UniversityGUI, you will see the frame with the correct title, but unfortunately it will show only one button – `Display`. To understand what is happening, we will need to learn about layout managers.

10.3.3 Exercises

1. Give some examples of other kinds of components, in addition to JButton and JLabel.

2. Is a Container an example of a component? (i.e. is Container a subclass of Component?)

10.4 Layout managers

We have seen that components can be added to containers to produce a GUI. A contentPane is an example of a container, and a button is an example of a component. When several components are added to a container, those components must be visually positioned within the container in some way. Hopefully, they will be positioned in such a way that the user will find it easy to understand the meaning and purpose of each component; it would be easy to confuse the user if the components were situated at random positions within the container. It is possible to specify an exact position within the container for each component; but suppose we later add or delete components? What would happen when the user resizes the container? Our GUI would have to recalculate the position of each component to maintain a good appearance for the container.

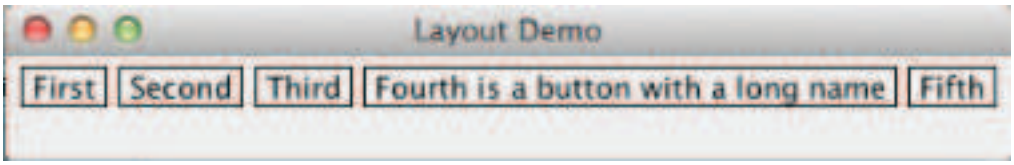


Figure 10.4: Frame with five buttons, using flow layout

There is a better solution to the problem of positioning components in a container: the *layout manager*. `LayoutManager` is an interface in java.awt, and every container has a layout manager. The layout manager will position components in the container in a fairly 'intelligent' way. The layout manager will also reposition, or resize, the components at appropriate times (e.g. the container's size changes). There are several classes which implement the `LayoutManager` interface; each has its own algorithm for positioning components. If you don't like the look of your container, you can use a different layout manager which may give it a better appearance. In this section we will examine some of the more commonly used layout managers.

10.4.1 Flow Layout

The easiest layout manager to use is called *Flow Layout*. A `FlowLayout` is the default layout manager for many kinds of containers, including `JPanel`. When you create a `JPanel`, its layout manager will be `FlowLayout`:

```
JPanel buttonPanel = new JPanel(); // layout mgr is flow layout
```

To see how the components are positioned by a FlowLayout, we have developed a simple GUI with five buttons. Each button has a caption, but the fourth button has a much longer caption, which makes the default size of that button significantly larger than the other buttons. The GUI initially appears as shown in Figure 10.4. Note that the buttons are positioned in the order in which they were added to the container (i.e. the contentPane). When the window is resized, the buttons are automatically shifted so as to have a 'nice' appearance, as shown in Figure 10.5. With flow layout, the components are arranged horizontally as long as they fit in the container; but if they do not fit, they will be moved vertically to available space. The components seem to flow to available space in the container, hence the name `FlowLayout`.

10.4.2 Grid Layout

A *Grid Layout* will position the components in a rectangular array, or grid, with rows and columns (think of a spreadsheet, or a checkerboard). When components are added, the columns of the first row are filled with components before moving to the second row; i.e. the container is filled in *row-major* order, not in *column-major* order. When instantiating a GridLayout, you can specify the number of rows and columns it will have. Then you can set the layout

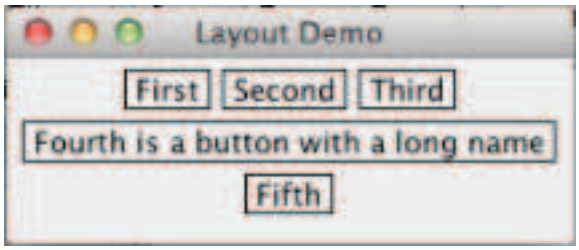


Figure 10.5: Frame with five buttons, using flow layout, after resizing the frame

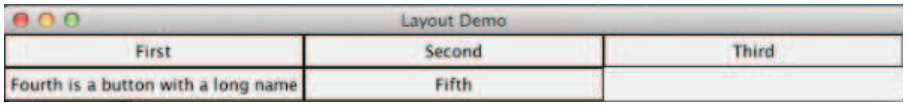


Figure 10.6: Frame with five buttons, using grid layout, 2 rows and 3 columns

manager when instantiating the container:

```
// 2 rows and 3 columns
LayoutManager gridMgr = new GridLayout(2,3);
JPanel inputPanel = new JPanel(gridMgr);
```

The first parameter in the constructor for `GridLayout` is the number of rows, and the second parameter is the number of columns in each row. An example of a frame with grid layout is shown in Figure 10.6. This frame has five buttons (same as the example for flow layout).

If you wish the components to be arranged in a horizontal grid with one row and a variable number of columns, specify 0 as the number of columns:

```
// One row
LayoutManager gridMgr = new GridLayout(1,0);
```

In this case the number of columns will increase as components are added to the container. Alternatively, to arrange the components in a vertical column, use 0 for the number of rows:

```
// One column
LayoutManager gridMgr = new GridLayout(0,1);
```

In general, if the number of rows (columns) is 0, the grid will have as many rows (columns) as are needed to accommodate the components which have been added.

As you experiment with Grid Layout, you will notice a few things:

- The components are arranged in a rectangular grid
- If the number of components added to the grid is less than the product of the number of rows and the number of columns, unused grid positions are empty.
- As you resize the container, the components also resize to fill their respective grid positions (this may be undesirable, and we will address this issue

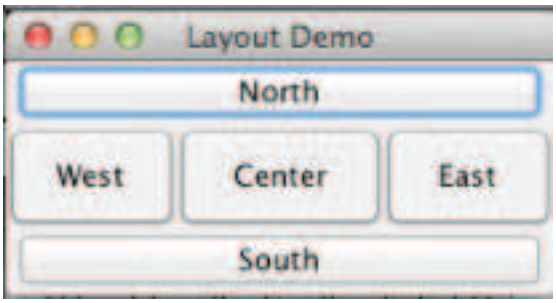


Figure 10.7: Frame with five buttons, using border layout

below)

- There can be at most one component in each position of the grid. If you add a second component to a position, it replaces the first component.

10.4.3 Border Layout

Border Layout is a very commonly used layout manager. It allows you to arrange the container into five regions. A single component may be placed in any of those regions. The five regions are: NORTH, SOUTH, EAST, WEST, and CENTER, as shown in Figure 10.7.

To create a container with a BorderLayout manager:

```
LayoutManager borderMgr = new BorderLayout();
JPanel outPanel = new JPanel(borderMgr);
```

Then to add a component to a particular region, use a class constant from the `BorderLayout` class to specify the region:

```
outPanel.add(new JLabel("hi"), BorderLayout.NORTH);
outPanel.add(new JButton("there"), BorderLayout.CENTER);
```

When using a Border Layout:

- There can be only one component in each region (as with GridLayout). If you add a second componenet to a region, it will replace the first component.

- Each component will expand to fill the entire region as the container is resized (as with Grid Layout).

- If no component is added to a particular region, that region will not be shown at all. The other regions will expand to fill up the entire container.

- The default region is CENTER. If you do not specify a region, the component will be placed in the center region.

- BorderLayout is the default layout manager for the contentPane. This explains why our UniversityGUI in the previous section did not show both buttons in the contentPane. The layout manager was BorderLayout, by default, and both buttons were added to the center region. The displayButton replaced the admitButton.

10.4.4 Nested containers and summary of layout managers

The layout manager for a particular container can be changed at execution time, using the `setLayout(LayoutManager)` method. For example:

```
contentPane.setLayout(new GridLayout(4,3));
```

changes the layout manager for the contentPane to GridLayout.

We have been using the words 'container' and 'component' in a somewhat general sense without giving precise definitions of these words. `Container` and `Component` are both classes in java.awt. Moreover, `Container` is a subclass of `Component`. This means that every `Container` *is-a* `Component`. This provides for the nesting of containers. You can put components into a container, but since every component *is-a* container, you can put a container into a container. This recursive notion is very common in computer science (think of the folders on your computer, which may contain other folders, which in turn may contain other folders, ...). When we nest containers inside other containers, each of those containers may have its own layout manager; we will make use of this property in our UniversityGUI class.

We mentioned that only one component may be placed into a position when using GridLayout, and only one component may be placed into a region when using BorderLayout. If you wish to put more than one component into a position or region, consider adding a container to that position or region. Then several components can be added to the nested container, as shown in Figure 10.8.

This diagram shows a contentPane with BorderLayout. Its five regions contain:

- NORTH – A container with Flow layout
- SOUTH – A container with Border layout. Its five regions contain:
  - NORTH – Nothing
  - SOUTH – Nothing
  - EAST – A component which is not a container, perhaps a button or label, not shown here for lack of space
  - WEST – A component which is not a container, perhaps a button or label, not shown here for lack of space
  - CENTER – A component which is not a container, perhaps a button or label, not shown here for lack of space
- EAST – Nothing

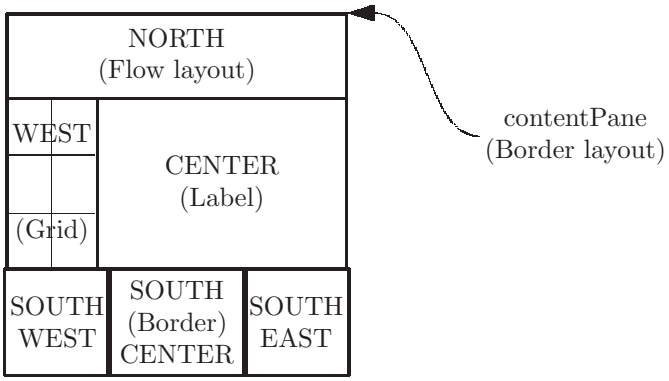


Figure 10.8: A contentPane with BorderLayout, in which containers have been placed in the north, west, and south regions – the container in the south region also uses BorderLayout

- WEST – A container with Grid layout, 3 rows and 2 columns. Its six positions could be components which are not containers, perhaps buttons or labels, not shown here for lack of space
- CENTER – A label

In summary, layout managers may not always perform in an ideal way, but most programmers feel the advantages of using layout managers far outweigh the disadvantages. There are other layout managers in java.awt which we have not mentioned; if your GUI doesn't look just right, you are probably using the wrong layout manager.

10.4.5 University Information System - version 1

Version 1 of our University Information System is in the project university-v1 in the repositry for this chapter. At this point there is only one class in the project: `UniversityGUI` which will define and place all the components in the graphical user interface for our information system.

The frame is instantiated in the constructor, with the title "State U". The constructor also obtains a reference to the contentPane and calls a helper method `makeFrame()` to do all the work involved in placing components into the contentPane.

In the `makeFrame()` method we note that the default layout manager for the contentPane is BorderLayout. We make use of only three regions in the BorderLayout:

- North - contains a JPanel called `buttonPanel` with FlowLayout (default) into which we place three buttons:
  - searchButton, to search for a Student

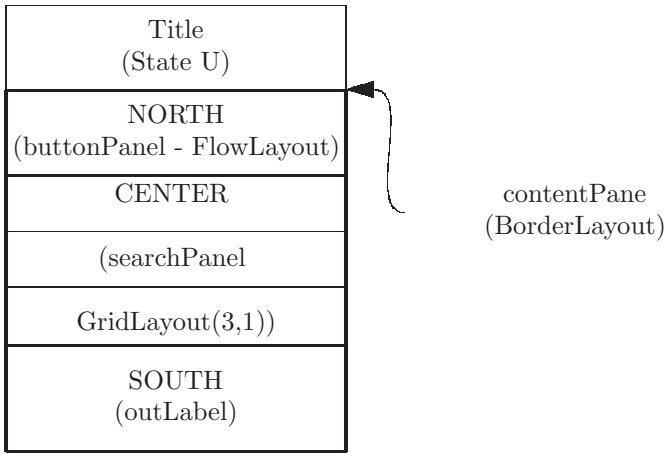


Figure 10.9: Diagram of the GUI for the University Information System, version 1

  - admitButton, to admit a new Student to the University
  - displayButton, to display all students in the University

- Center - contains a JPanel called `searchPanel` which contains check boxes to select a search by name or by ssn. It also contains a text field in which the user may enter a name or ssn for search purposes.

- South - A label which will display output: either search results or a display of all students.

Figure 10.9 shows a diagram of the layout of our graphical user interface at this point.

Students often ask about the sequence in which components are added to the contentPane, and the sequence in which we specify attributes (Should the frame be made visible before or after adding all the components?) In most cases it really doesn't matter; we are simply building a structure and filling in references. As an example, we show a hypothetical object diagram for our frame in Figure 10.10. We call it a hypothetical diagram because we are just guessing at the names of the private fields in many of these classes (as we did with Sets, Maps, etc). To fit this diagram on the page we are not showing some of the referenced objects (and we are probably omitting many fields as well) but Figure 10.10 can be helpful in understanding the internal structure of a GUI.

10.4.6 Exercises

1. Give the name of the LayoutManager with the following properties:

   (a) Components are repositioned vertically and horizontally as the container is resized.

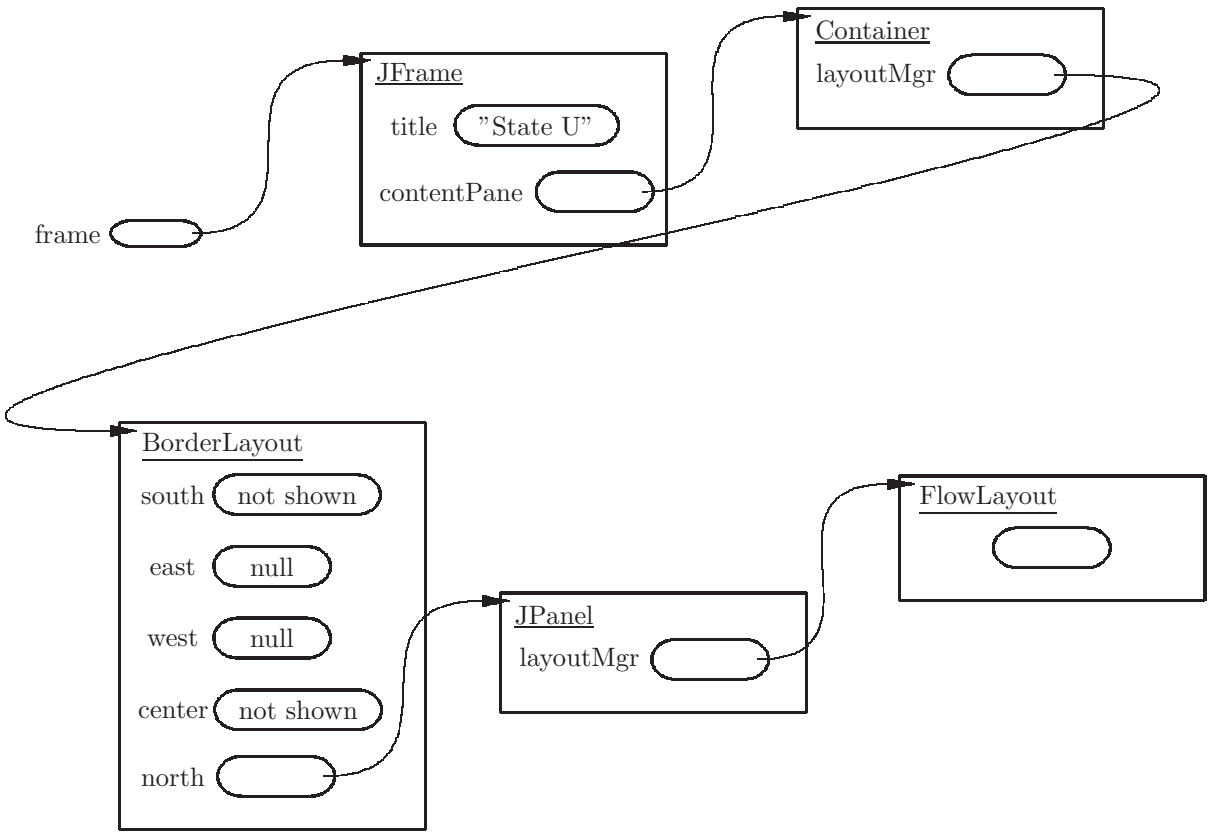


Figure 10.10: A hypothetical object diagram showing the value of the variable `frame` after the `makeFrame()` method has terminated

(b) Is capable of storing at most five components.

(c) Will arrange components into rows and columns.

(d) Has regions named North, South, East, West, and Center.

2. Identify the apparent error, if any, in each of the following:

(a)
```
JPanel myPanel = new JPanel();
myPanel.setLayout (new BorderLayout());
myPanel.add (new JButton ("Click"), BorderLayout.NORTH);
myPanel.add (new Label ("Hi"), BorderLayout.SOUTH);
myPanel.add (new Label ("There"), BorderLayout.NORTH);
```

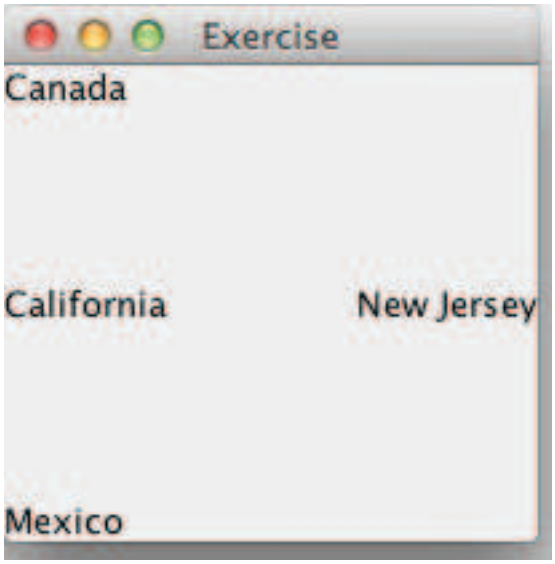
(b)
```
JPanel myPanel = new JPanel();
myPanel.setLayout (new GridLayout(3,2));
myPanel.add (new JButton ("Click"), BorderLayout.NORTH);
```

(c)
```
JPanel myPanel = new JPanel();
myPanel.setLayout (new GridLayout(0,2));
myPanel.add (new JButton ("Click"));
```

3. How many rows and columns of components will there be in `myPanel` after the code shown below has executed?

```
JPanel myPanel = new JPanel();
myPanel.setLayout (new GridLayout(0,2));
myPanel.add (new JButton ("Click"));
myPanel.add (new JButton ("Here"));
myPanel.add (new JButton ("Now"));
```

4. Define a class which will produce a GUI as shown below when instantiated.



10.5 Actions and Listeners

We now have a GUI for our University Information System. The only problem is that it does not do anything. If you click on a button, nothing happens. We need to use Actions and Listeners; thus we will introduce *event-driven* programming. Up to this point, all appications that we hve developed started up from a `main` method, or from an IDE, and ran to completion (perhaps pausing for input from stdin). With GUIs the sequence of events is much different. Once the GUI has been initialized, it waits for a user action. Examples of actions are:

- The user clicks on a button
- The user selects an item from a menu
- The user moves the mouse
- The user types on the keyboard
- The user provides some other form of interaction with the computer

When any of these actions occur, our application can be programmed to handle them in an appropriate way. Alternatvely, we may wish our application to ignore certain actions (such as a mouse movement). Run time computations occur as a result of a particular action or event, thus the phrase 'event-driven' is used to describe this kind of program.

In the package `java.awt.event` there are several kinds of *listeners*. Listeners are objects which detect a particular event, and are then capable of handling the event appropriately. Each listener is an interface in the package

`java.awt.event`; the listener must be implemented in order to handle actions. This means we must import needed classes from this package:

```
import java.awt.event.*;
```

Note that `import java.awt.*;` will not give us the classes from `java.awt.event` because the * matches class names only , not package names, in a package, which is essentially just a folder, or directory. (The . in java.awt.event is like a slash - forward or backward - in a unix or Windows directory path)

Examples of listeners from java.awt.event are:

- `ActionListener` - listen for an action such as a button click or a menu selection
- `MouseListener` - listen for a mouse movement, mouse button down, mouse button up, etc.
- `KeyListener` - listen for a keyboard strike
- `TextListener` - listen for a change to a text field

The most useful listener is `ActionListener`, and this is the one we will be using most. As we look at the API for the `ActionListener` interface, we see that it has only one method: `actionPerformed (ActionEvent)`. This means that any class which implements `ActionListener` must define the `ActionPerformed` method. When an action occurs, any listener which is listening for that action will automatically call the `actionPerformed` method. It is our responsibility to define this method to handle the action before returning control to the Java runtime environment. The program is event-driven: nothing happens until an event causes a listener to respond.

The parameter for the `actionPerformed` method is of type `ActionEvent`. This parameter provides us with everything that is known about the event, as shown in the API for `ActionEvent`. These include, but are not limited to:

- Text from the component which cause the event (such as a button's caption)
- Time that the event occurred
- A reference to the component which caused the event; returned by the method `getActionCmd()` (this will be used to identify which component caused the event; e.g. which button was clicked)

Any component for which we need to handle events, must be *registered* with an event listener. This can be done with the the method

`addActionListener(ActionListener)`.

One last item needs to be addressed before we try to apply all this. How do we instantiate listeners? The easiest way to do this is to make our GUI an ActionListener. We can do this by declaring that it implements ActionListener. Then the GUI object is itself an `ActionListener`, and can be used as the actual parameter in the call to `addActionListener`. For example:

```
public class MyGUI implements ActionListener
{  private JButton myButton = new JButton ("click me");

   ...

   private void makeFrame()
   {  myButton.addActionListener (this);  // this MyGUI object
      ...
   }

   public void actionPerformed (ActionEvent evt)
   {  if (evt.getActionCmd() == myButton)
         // code to handle the button click
   }
}
```

In the `makeFrame` method we register an ActionListener with a button. The usage of the key word `this` should be explained. The key word `this` stores a reference to the object on which the method was called. In this case it refers to `this` MyGUI, which *is an* ActionListener. This usage of `this` is the same as that which was described previously in chapter 3.

Note in the `actionPerformed` method that we obtain a reference to the component that caused the action (returned by `getActionCmd()`) and compare it with the reference in the field `myButton`. In this case the comparison is `==` and not `.equals(Object)` because we are comparing references rather than the objects to which those references refer.

10.5.1 University Information System - version 2

We are now ready to build a GUI for our information system. We wish it to be capable of admitting students, displaying admitted students, and searching for a particular student by name or by ssn.

At this point we should clarify the purpose of a user interface; it should be used only to communicate with the user. All computations and processing of data should be done separately in what is often referred to as an *engine* or *kernel*. There should be a clear separation of the user interface and the engine. If this is done properly, we should be able to remove the graphical user interface for our information system, and plug in a command-line user interface, without making any changes to the engine.

To further clarify, we now define the engine, and we call it `UniversityInfoSys`. This class will have a set of students. Version 2 will be able to do the following:

- Add a student to the set
- provide a reference to the set of students
- search for a student by name or by ssn

Our `UniversityInfoSys` class is shown in the project `university-v2`. In this class we note that

- There is one field, a set of students.
- The constructor instantiates the set, it is now an empty set. The constructor also instantiates the GUI. This will cause the GUI frame to be initialized and to pop up on the monitor.
- The `addStudent` method adds one student to the set of students.
- The `getStudents` method is an accessor method which returns a reference to the set of students who have been admitted.
- The `searchByName` method will search for all students who have a given name, and return those students as a set.
- The `searchBySSN` method will search for the one student who has the given SSN.

10.5.2 Exercises

1. For each of the following interfaces in the package `java.awt.event` show all methods which must be defined in any class which implements the interface.
   (a) MouseListener
   (b) KeyListenerr
   (c) TextListener
2. Show the code necessary to print the caption of a button named `goButton` to stdout when the button is clicked. Assume the button has been registered with `this ActionListener`.
3. In the project `university-v2` in this chapter's repository the program will throw a NullPointerException and crash if the user cancels input for the name and/or ssn when attempting to admit a student. Correct this problem so that the student is not admitted, and an appropriate message is displayed on the output label.
4. In the project `university-v2` in this chapter's repository if the user attempts to enter two students with the same ssn, the first one is expelled and replaced by the second one. Modify this project so that the first student is retained, and an appropriate message is displayed on the output label.

10.6 Menus

One useful way of communicating commands to a GUI is through the use of *Menus*. A menu is normally a drop-down list of items, one of which can be selected with the mouse. Menus typically allow users to open or save to a disk file, terminate an application, edit data, etc. You can define menus to perform any action you wish. We will include two menus in the GUI in our information system: a File menu and a Help menu.

10.6.1 Adding menus to the frame

If you wish to include menus in your GUI, your frame must have a *menu bar* (there can be only one menu bar). You may then add several menus (e.g. File, Help) to the menu bar. Each menu may in turn have several menu items (e.g. Save, Save As, Open, Quit). A diagram of the menu bar for our University GUI is shown in Figure 10.11.

Menus are included in version 3 of our university information system. We will have a File menu which allows us to save our data to a disk file, and to retrieve our data from a disk file. We will also have a Help menu for the clueless users. The menus are set up in a separate method: `setMenus()`.

10.6.1.1 Setting the MenuBar

If your frame is to make use of menus, it must have a *menu bar* (We will use `JMenuBar` from `javax.swing` to be up-to-date). Think of this as a container, appearing as a rectangle as shown in Figure 10.11, for all the menus. A frame cannot have more than one MenuBar. The method used to include a JMenuBar is `setJMenuBar(JMenuBar)` as shown below:

```
JMenuBar menuBar = new JMenuBar();
frame.setJMenuBar (menuBar);
```

Note that this method is *not* `addJMenuBar`; the word 'add' would imply that a frame can have more than one menu bar.

10.6.1.2 Adding Menus to the MenuBar

One or more *menus* may be added to the menu bar. When a menu is created, it is provided with a String, which is the text shown on the menu. The menu can then be added to the menu bar with the method `add(JMenu)` as shown below:

```
JMenu fileMenu = new JMenu("File");
menuBar.add (fileMenu);
```

10.6.1.3 Adding the MenuItems to a Menu

Each menu may have 0 or more *menu items*. A menu item may be selected by the user to generate an event (e.g. open a file, quit, edit text, etc.). Again,

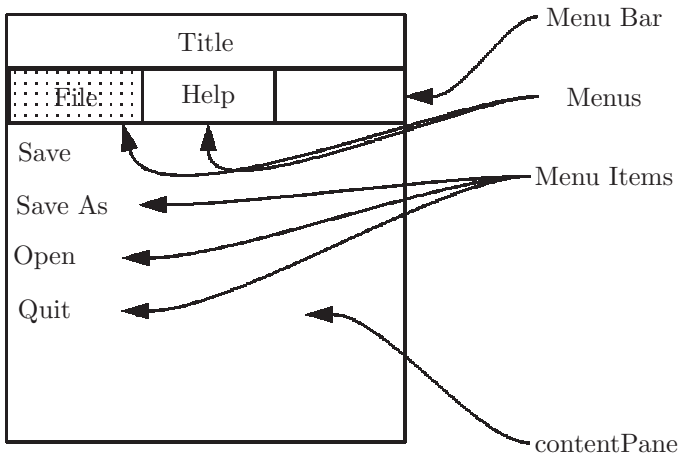


Figure 10.11: A simple frame with a title, menu bar, and contentPane

we will be using JMenuItem rather than MenuItem. When the menu item is created, it is provided with a String, which is the text shown on the menu item. To add a menu item to a menu, use the method `add(JMenuItem)`. A menu item which is supposed to save our data to disk can then be added to the File menu as shown below:

```
JMenuItem saveItem = new JMenuItem("Save");
fileMenu.add (saveItem);
```

The menu items in our GUI are declared as fields rather than as local variables because we may wish to refer to them from other methods in this class.

10.6.1.4 Other features of menus

Menu items may be *disabled* (or greyed-out) by using the method `setEnabled(boolean)`. For example:

```
saveItem.setEnabled(false); // disable save
saveItem.setEnabled(true); // enable save
```

The disabling of a menu item is desirable when its selection would not make sense, particularly if the user can cause an error by selecting the item. A good GUI will protect the novice user from making careless mistakes.

The java class library has a number of other features involving menus:

- Menu items may themselves contain other menu items
- menus can pop up at places in the frame other than the menu bar
- The menus on a menu bar, and the menu items on a menu can be changed as the program executes, to ensure that the user is always seeing options which are currently of interest

- Even the menu bar may be changed at run time, to ensure that the user is always seeing options which are currently of interest (at any one time there can be no more than one menu bar)

10.6.2 Listening for menu selection

At this point our GUI has menus, but they do not do anything; the user can make selections, but nothing happens as a result of the selection. Our GUI needs to *listen* for these actions and handle them appropriately.

The handling of a menu item selection is exactly the same as the handling of a button click; it is merely an action. If there is an action listener registerd with the menu item, the method `actionPerformed(ActionEvent)` is called automatically. Just as we did with buttons, we can use the `addActionListener(ActionListener)` to register an action listener with a particular menu item. In our example, the GUI class implements the interface `ActionListener`, so we can register the Save menu item as follows:

```
saveItem.addActionListener(this);
```

Having done this, when the user selects the Save menu item, the method `actionPerformed(ActionEvent)` is called automatically.

To handle this event in the `actionPerformed` method, we compare the reference provided by the parameter, `evt`, with a reference to the Save menu item. If they are equal (same reference), we know that `actionPerformed` was called as a result of the Save menu item selection. This is done as shown below:

```
public void actionPerformed (ActionEvent evt)
{   ...
    if (evt.getSource() == saveItem)
       infoSys.saveFile();
    ...  // handle other menu items here
}
```

Note that the actual saving to disk is *not* done in the user interface; it is done with a call to `saveFile()`, a method in the information system engine. In general all computation and processing should be done in the engine (or in other classes; we could have a separate class dedicated to input/output of data). A user interface should be used only for communication with the user.

10.6.3 Menus for the University Information System - version 3

The GUI for version 3 of our university information system has a menu bar with two menus:

- File menu - has 4 menu items:
  - Save menu item: save the set of students in the current disk file. Prompt for a file name if necessary.

  - Save As menu item: prompt the user for a new file name to be used as the current disk file, and save the set of students. Provide an error message if the file cannot be opened for output.
  - Open menu item: read a set of students from the current file; provide an error message if the file does not exist or cannot be opened for input.
  - Quit menu item: terminate the application.

- Help menu - intended to provide assistance for the novice user. Implementation of the menu items contained here is left as an exercise for the reader.

After we set up the frame and add the menus, we should tell the frame to resize itself appropriately to accommodate all the components which have been added, as well as the menubar. This is done with the `pack()` method:

`frame.pack();` // resize the frame

The engine for this project, UniversityInfoSys, is expanded to include methods for saving to and retrieving from disk. The class has a String, `fileName`, which stores the name of the file currently storing a permanent copy of our data. This file is in the same folder as the project, by default, though we could specify a path to a different folder. The file is a plain text file in which each student name and each student ssn is on a separate line. We will not go into the details of these methods here because they are not relevant to the subject of this chapter, but they apply the concepts of input and output as covered in chapter 9.

10.6.4 Exercises

1. Point out the error, if any, in each of the following:

   (a)
   ```
   JMenu menu = new JMenu ("File");
   frame.setJMenu (menu);
   ```

   (b)
   ```
   JMenuBar menuBar = new JMenuBar ();
   frame.addJMenuBar (menuBar);
   ```

   (c)
   ```
   JMenu menu = new JMenu ("File");
   JMenuItem menuItem = new JMenuItem ("Quit");
   menu.addItem (menuItem);
   ```

   (d) Assume the GUI class implements `ActionListener`

   ```
   JMenu menu = new JMenu ("File");
   JMenuItem menuItem = new JMenuItem ("Quit");
   menu.add (menuItem);
   menuItem.addActionListener (menu);
   ```

   (e) Assume there is a button named `goButton` and a method named `go()`.

```
public void actionPerformed (ActionEvent evt)
{  if (evt.getSource().equals (goButton))
       go();
}
```

2. (a) Which method in the `ActionEvent` class can be used to find out the exact time that the event occurred?

   (b) What are the units of time used in that method's result?

3. The project university-v3 in this chapter's repository includes a Help method which does not do anything. Improve the Help menu so that the user will understand the meanings of the three buttons. (Do NOT print to stdout nor to stderr. All communication with the user must go through the GUI.)

   Hint: See the API for `JOptionPane` in the javax.swing package.

4. Add a menu to the menubar which will ...

10.7 Projects

1. Build an application with a GUI which will test a student on integer arithmetic skills. Include the operations:

   - Addition
   - Subtraction
   - Multiplication
   - (Integer) quotient on division
   - (Integer) remainder on division

   The student should be rewarded for attempting more difficult operations and for quick responses. Include a menu to set the level of difficulty. An example of this program can be executed from the project `games` in this chapter's repository. Instantiate the Arithmetic class to start it up.

2. Build a java application that works like the one in the project `chase` in the code repository for this chapter. To execute it, simply instantiate the `Chase` class.

   Hints:

   - See the API for the `MouseListener` interface in the `java.awt.events` package.
   - Use a List of buttons, all of which are not visible, except for one button.
   - Use a GridLayout for the contentPane. Add all the buttons to the contentPane, and register each one with a MouseListener.

- In the `MouseEntered` method, make the selected button invisible, and choose some other button to become visible. (This can be done with a random number generator, or with clever use of the mod % operator)

Chapter 11

Abstract Data Types

In chapter 2 we discussed the various data types provided in java. Some examples of primitive data types are: int, float, double, char, boolean. In addition any class may be regarded as a data type - we can declare a variable that stores a reference to an object of a defined class.

In this chapter we examine classes which are designed to be used in the same way that primitive data types are normally used. In this way we can use software to improve, or enhance, the properties of the primitive data types provided by the hardware. We call these classes *abstract data types*.

Some programming languages, such as C++, provide features which make abstract data types even more attractive. One such feature, called *operator overloading* allows the programmer to define the semantics of primitive operators to include programmer-defined abstract data types. Unfortunately java does not allow operator overloading. Nevertheless we can define very useful abstract data types, but the user must understand the proper syntax for their use.

11.1 The Rational ADT

In mathematics a *rational* number is one that can be expressed as the ratio of two whole numbers. Some examples of rational numbers are: 3/2, 2/1, 4/2, 0/4, 5/-3.

In this section we develop an ADT for rational numbers. This ADT is motivated by some of the shortcomings of the floating point types.

11.1.1 Some problems with float and double

Java provides two floating point types: `float` and `double`. These types presumably reflect data types of the underlying hardware. I.e. most computers have a 32-bit floating point numeric format (which java calls `float`) and a 64-bit floating point numeric format (which java calls `double`).

These floating point types can exhibit unexpected behavior (which we will investigate further in the next section). Here are a few of the problems we might encounter:

- 1.0/3.0 produces the result 0.3333333333333333 which is close but not perfect. This repeating decimal cannot be represented exactly in base 10 (nor in its internal representation on the computer).
- $0.1 + 0.1 + 0.1 == 0.3$ is false. If you don't believe this, try it. The reason has to do with the fact that 0.1 cannot be represented exactly on a binary machine, for the same reason that 1.0/3.0 cannot be represented exactly in decimal.
- 1.0E200 + 1.0 == 1.2E200 is true. If you don't believe this, try it. This problem results from the limited precision available in the floating point formats. The Rational ADT will not address this problem, but we will address it in a later section.

Smart programmers will generally avoid comparing floats for equality to avoid some of these unexpected results. In cases where it is necessary to compare floats for equality, it is better to determine whether the two values differ by more than a very small tolerance. For example, if one wishes to compare the floats x and y for equality we wish to avoid:

```
if (x == y)
```

Instead we will use a small tolerance, such as:

```
float epsilon = 1.0e-7
```

and instead compare x and y as follows:

```
if (Math.abs (x - y) < epsilon)
```

In this statement we determine whether the absolute value of the difference between x and y is smaller than epsilon; if so, we conclude that the values of x and y are sufficiently close that they can be considered equal.

11.1.2 Defining the Rational ADT

Some of the problems described above can be alleviated with a *Rational* ADT. Our ADT will store two ints to represent a rational number: a numerator and a denominator (the denominator must not be zero). In this way we can store exact values in cases where floats are not perfect. A numerator value of 1 and a denominator value of 3 is a perfect representation of one third.

Our ADT should include operations; we will define addition, subtraction, multiplication, and division of rationals to provide the user with a complete and useful ADT. We'll start with the class definition shown below:

```
/** Rational ADT
  * A Rational has a numerator and a denominator.
  * Rationals can be added, subtracted, multiplied, and divided
  */
public class Rational
```

```
{   private int num;             // numerator
    private int denom;           // denominator, cannot be 0

    /** @param d must not be 0
      */
    public Rational (int n, int d)
    {  num = n;
       denom = d;
    }

}
```

This means that we can create Rational objects such as:

```
Rational r;
r = new Rational (1,3);           //   1/3
r = new Rational (-7,2);          //   -7/2 = -3 1/2
r = new Rational (2,6);           //   2/6 = 1/3
r = new Rational (17,1);          //   17
```

Note that there can be more than one representation for the same rational number: $1/3 = 2/6$
If at some point we wish to compare rational numbers for equality, or we exceed the precision of ints, this could be a problem, which we will address here. The constructor should call a method to simplify this Rational, or reduce it to lowest terms.

```
    /** @param d must not be 0
      * Zero is always represented as 0/1
      */
    public Rational (int n, int d)
    {  num = n;
       denom = d;
       if (n == 0)
          d = 1;            //   normal form for 0
       else
          simplify();
    }
```

In this way Rationals should always be simplified when created (and we should avoid setting the numerator or denominator of an existing Rational). In other words, do this: `r = new Rational (n,d);` not this:

```
r = new Rational();
r.num = n;
r.denom = d;
```

The `simplify` method will utilize the fact that we can divide two numbers by their greatest common divisor to eliminate the factors which they have in

common. For example, gcd(70,21) is 7 because: $70 = 2 \cdot 5 \cdot 7$ and $21 = 3 \cdot 7$. To simplify 70/21 we divide both numerator and denominator by 7 to obtain 10/3. The `simplify` method is shown below:

```
/**  Simplify this Rational, eliminating common factors
 *   from the numerator and denominator.
 */
private void simplify()
{   int g = gcd(num, denom);   // greatest common divisor of num and denom
    num = num / g;
    denom = denom / g;
}
```

To define the gcd method, we use a classic algorithm known as the *Euclidean* algorithm.

```
/** @return the greatest common divisor of a and b.
  * @param a must greater than or equal to 0
  * @param b must greater than 0
  */
private int gcd (int a, int b)
{  int r = a % b;
   while (r > 0)
       {  a = b;
          b = r;
       }
   return b;
}
```

Note that our `simplify` method works only for non-negative numbers. (Improvement is left as an exercise.)

We can now proceed to define some of the arithmetic operations for our Rational ADT. With all of our operations, we'll assume that the left operand is *this* Rational, and the right operand is the parameter. If we had operator overloading and the client wanted to multiply the Rationals r1 and r2, it would simply use: `r1*r2`. But since we do not have operator overloading it will be done this way:

```
r1.mult(r2)
```

Multiplication is the easiest so we'll start with that. Recall that when multiplying rational numbers we simply multiply the numerators and multiply the denominators: $(a/b) * (c/d) = (ac)/(bd)$

```
/** @return The product, this Rational multiplied by the parameter r
 */
public Rational mult (Rational r)
{  return new Rational (num * r.num, denom * r.denom); }
```

We next implement addition of Rationals. Recall that we can add the numerators only if the denominators are equal. To ensure the denominators are equal, we can multiply each operand by a number equal to 1:

$a/b + c/d = a/b * d/d + c/d * b/b = ad/bd + bc/bd = (ad + bc)/bd$

We define the addition method using this formula.

```
/** @return The sum, this Rational plus the parameter r
 */
public Rational add (Rational r)
{  return new Rational ((num*r.denom + denom*r.num)/
                        denom * r.denom); }
```

Subtraction and division will be left as exercises for the student. Note that we can now compute $0.1 + 0.1 + 0.1$ with perfect accuracy:

```
Rational tenth, result;
tenth = new Rational (1,10);            //  1/10
result = tenth.add(tenth);              //  2/10
result = result.add(tenth);             //  3/10
```

To print out a Rational we should have a `toString()` method:

```
/**  @return this Rational as a String
 *   Show numerator over denominator
 */
public String toString()
{  return num + "/" + denom;  }
```

To determine whether two Rationals are equal:

```
/** @return true only if this Rational represents
 *  the same number as obj.
 *  Pre: this Rational and obj have both been simplified.
 */
public boolean equals (Object obj)
{  if (obj==null | (! (obj instanceof Rational))
      return false;
   Rational r = (Rational) obj;
   // numerators and denominators must be equal
   return num == r.num && denom == r.denom;
}
```

If the user wishes to create a HashSet or HashMap of Rationals, we should supply a hashCode method, which agrees with the equals method: two Rationals which are equal should have the same hashCode.

11.1.3 Exercises

1. Define a division method for the Rational class:

```
/** @return the result of dividing this Rational by
    the parameter, r.
    @param r is not zero.
  */
public Rational divide (Rational r)
```

   Hint: $a/b \div c/d = ad \div bc$

2. Define a subtraction method for the Rational class:

```
/** @return the result of subtracting
    the parameter, r, from this Rational.
  */
public Rational subtract (Rational r)
```

3. Improve the `simplify` method called by the constructor, to allow for negative values for numerator and/or denominator.

4. Define additional constructors in the Rational class to allow convenient ways of constructing Rational numbers which are whole, and Rational numbers representing zero:

```
Rational r = new Rational (17);            //    17/1
Rational p = new Rational ();              //    0/1
```

   Hint: Eliminate duplicated code in your constructors by calling another constructor using the keyword *this*. A call to another constructor must be the first statement in your constructor.

5. Define a copy constructor for the Rational class. It should have one parameter, the Rational being copied.

6. Define a method in some class, to return the purchase price (including tax) on a purchase. Assume the retail price and the tax rate are both Rationals. For example, a tax rate of 7/100 is a rate of 7%. You may use the copy constructor defined in the previous exercise.

```
/** @return the total purchase price, including tax.
  * @param retail The retail price, in dollars.
  * @param taxRate The tax rate
  */
public Rational getCost (Rational retail, Rational taxRate)
```

7. Define a method in the Rational class which will round this Rational up to a given whole fraction. For example, if fraction is 100, it will return the Rational rounded up to the next higher one hundredth. Test your method in conjunction with your getCost method to obtain a purchase price rounded up to a penny (fraction = 100).

```
/** @return this Rational rounded up to the next higher
 *  fraction.
 *  If fraction is 100, it will round up to the
 *  one hundredth.  Can be used in financial applications
 *  to round to the (higher) penny,
 *  Pre: this Rational is not negative.
 */
public Rational roundUp (int fraction)
```

8. Show how to create Rational objects representing each of the following numbers.

   (a) 7

   (b) 4.1

   (c) -12.04

9. Define a toString() method for Rationals. The easiest way to do this is simply to include the character '/' between the numerator and denominator. An example could be `"13/2"`.

```
/** @return A String representation of this
 *  Rational.
 */
public String toString()
```

10. Improve your toString() method to return a representation in *composite* form, such as 2 1/3 rather than 7/3. Hint: Watch for negative values; in normal form, it is the numerator which will be negative.

11. Define an `equals (Object)` method for Rationals, so that the client can have a Set of Rationals. You may assume that this Rational and the parameter are both in normal form.

```
/** @return true only if the parameter obj
 *  is a Rational and is equal to this Rational.
 */
public boolean equals (Object obj)
```

12. Define a hashCode method for Rationals, so that the client can have a HashSet of Rationals. You may assume this Rational is in normal form.

```
/** @return A hashCode for this Rational.
 *  Pre: this Rational is in normal form.
 */
public int hashCode ()
```

13. We would like the Rational class to implement the `Comparable` interface, so that the client can have a TreeSet of Rationals. Change the class definition to read as follows:

    `public class Rational implements Comparable<Rational>`
    This means that a Rational can be compared with any other Rational for order as well as equality (i.e. greater than, less than). The compiler will require you to include a compareTo method:

```
/** @return a negative value if this Rational is less than
 *  the parameter r, a positive value if this Rational
 *  is greater than the parameter r, and 0 if they
 *  are equal.
 *  Pre: Both this Rational and the parameter r
 *  are in normal form.
 */
public int compareTo (Rational r)
```

11.2 MyFloat

In the previous section we examined some anomolies of floating point arithemetic. These resulted from the fact that certain values (such as 0.1) cannot be represented exactly on a binary machine, and from the limited precision of the floating point hardware. In order to gain a better understanding of floating point types, we will implement our own floating point ADT in this section. In doing so, we will use only whole numbers (i.e. ints). The presumption here is that floating point values (floats and doubles) do not exist, and that we are building this data type *from scratch*. In doing so, we will see many similarities with the Rational ADT of the previous section, and will use what we learned there.

We will call our ADT *MyFloat*, to distinguish it from the existing wrapper class in the java.lang package, Float. We begin by pointing out that every MyFloat can be represented by two whole numbers, and we call these the *mantissa* and the *exponent*. These whole numbers correspond to the two parts of a java constant when written in scientific notation, as in the following examples:

```
125e0 = 125.0
12e3 = 12000.0
12e-3 = 0.012
0e0 = 0.0
```

The mantissa of a MyFloat corresponds to the part of a floating point number before the `e`, and the exponent of a MyFloat corresponds to the part of a floating point number after the `e`. The exponent is always assumed to be an exponent of 10. In this way, any floating point value can be represented with two whole numbers.

11.2.1 Constructor for MyFloat

We begin our class definition as follows:

```
/** Every MyFloat has a mantissa and an exponent (of 10).
 *  MyFloat operations are addition, subtraction,
 *  multiplication, and division.
 */
public class MyFloat
{    int mant;
     int exp;        // exponent of 10

     public MyFloat (int m, int e)
     {  mant = m;
        exp  = e;
     }
}
```

As we noticed with the Rational ADT, MyFloat also has the property that there can be more than one representation for any number:
103e0 = 1030e-1 = 10300e-2 = 103000e-3 ... = 103.0
602e21 = 6020e20 = 60200e19 = 602000e18 ... = 6.02×10^{23}

We can eliminate trailing zeros in the mantissa:

```
/** Eliminate trailing zeros in the mantissa of this
  * MyFloat.  Adjust the exponent accordingly
  */
private void simplify ()
{  if (mant == 0)
      { exp = 0;               // Handle zero as a special case
        return;
      }
   while (mant % 10 == 0)      // Low order digit is zero?
      { mant = mant / 10;
        exp++;
      }
}
```

In the constructor we can now include a call to the `simplify()` method. As with the Rational ADT, we can be sure that all MyFloats are in normal form if created with our constructor:

```
/** Every MyFloat has a mantissa and an exponent (of 10).
 *  MyFloat operations are addition, subtraction,
 *  multiplication, and division.
 *  Constructed MyFloat will be in normal form.
 */
public class MyFloat
{    int mant;
     int exp;        // exponent of 10

     public MyFloat (int m, int e)
     {  mant = m;
        exp  = e;
        simplify();   // Put into normal form
     }
}
```

11.2.2 Arithmetic operations for MyFloat

We will now define the same four arithmetic operations for MyFloat which were defined for the Rational ADT: Addition, Subtraction, Multiplication, and Division. Again, we will be assuming that the left operand is `this` Myfloat, and the right operand is the parameter. Thus, if x and y are MyFloat objects, they can be multiplied to produce a product:

```
product = x.mult(y);
```

We'll start with multiplication because that is the easiest operation. When multiplying two MyFloats we multiply the mantissas and add the exponents, as shown in the following examples:

`123e3 * 2e4 is 246e7`, i.e. `123000.0 * 20000.0 = 2460000000.0`
`3e-3 * 2e1 is 6e-2`, i.e. `0.003 * 20.0 = 0.06`
`123e0 * 1e-4 is 123e-4`, i.e. `123.0 * 0.0001 = 0.0123`
`5e0 * 6e0 is 3e1`, i.e. `5.0 * 6.0 = 30.0`

In the last example, note that the result has been simplified to normal form (i.e. it has been *normalized*).

Our multiply operation can be defined as shown below:

```
/** @return the product of this MyFloat multiplied
  * by the parameter, f.
  */
public MyFloat mult (MyFloat f)
{  int mantissa = this.mant * f.mant;
   int exponent = this.exp + f.exp;
```

```
   return new MyFloat (mantissa, exponent);
}
```

Note that we have taken care to do this so that the result has been simplified to normal form (i.e. the `simplify()` method is called from the constructor).

Division is similar to multiplication, but is not as easy. The following examples show that to divide MyFloats, we divide the mantissas and subtract the exponents:

`12e4 / 3e1 = 4e3`, i.e. `120000.0 / 30.0 = 4000.0`

`4e0 / 1e3 = 4e-3`, i.e. `4.0 / 1000.0 = 0.004`

These examples are straightforward and easy because the mantissa of the left operand is a divisor of the mantissa of the left operand in both cases. The next two examples expose the case where division of the mantissas will not produce the desired result

`1e0 / 4e3 = 25e-5`, i.e. `1.0 / 4000.0 = 0.00025`

`1e0 / 3e0 = 333333333e-8`, i.e. `1.0 / 3.0 = 0.33333333`

Recall that integer division produces an integer quotient; for example 1/4 is 0, thus dividing the mantissas will not give the desired result. To correct this problem in the first example above, we must *adjust* the left operand by making the mantissa larger, and the exponent smaller, to form an alternate representation of the same number.

`1e0 = 10e-1 = 100e-2`

We can now perform the division without loss of precsion:

`100e-2 / 4e3 = 25e-5`

In the case of the second example above, we need to adjust the left operand even more. Notice that the larger we make the mantissa of the left operand, the more precision we will get in the result:

`1e0 / 3e0 = 0e0`, i.e. `1.0 / 3.0 = 0.0`, not correct.

`10e-1 / 3e0 = 3e-1`, i.e. `1.0 / 3.0 = 0.3`, a little better.

`100e-2 / 3e0 = 33e0`, i.e. `1.0 / 3.0 = 0.33`, even better.

`1000e-3 / 3e0 = 333e0`, i.e. `1.0 / 3.0 = 0.333`, even better.

The solution is to make the mantissa of the left operand as big as possible (without exceeding the bounds of the int primitive data type). This value can be obtained from the wrapper class for ints: `Integer.MAX_VALUE`. This is done in the private helper method, `adjust`, shown below, leading to the following definition of the divide operation:

```
/** @return the quotient when this MyFloat is divided
  * by the parameter, f.
  * @param f Must not be a representation of zero
  */
public MyFloat divide (MyFloat f)
{  MyFloat temp = new MyFloat (mant, exp); // temp copy of this
   temp.adjust (f);                              // adjust with respect to f
   return new MyFloat (temp.mant / f.mant, temp.exp - f.exp);
```

```
}

/** Adjust this MyFloat with respect to the parameter f,
  * to provide maximum precision when dividing
  */
private void adjust (MyFloat f)
{   while (mant % f.mant != 0 && mant < Integer.MAX_VALUE/10)
      {  mant = mant * 10;
         exp--;
      }
}
```

In the adjust method above, we wish to stop the loop when the mantissa of the parameter, f, divides the mantissa of this MyFloat without a remainder. We also wish to stop the loop before overflowing the capacity of an int. Also note that we made a temporary copy of this MyFloat in the divide method; we do not wish to make any alterations to either operand when dividing.

The `adjust` method shown above presumes that the mantissas of this MyFloat is positive. We need to allow for the fact that it could be negative. To do this we will make use of the absolute value method provided in the Math class: Math.abs(int).

```
/** Adjust this MyFloat with respect to the parameter f,
  * to provide maximum precision when dividing
  * @param f must not be a representation of 0
  */
private void adjust (MyFloat f)
{   while (mant % f.mant != 0 && Math.abs(mant) < Integer.MAX_VALUE/10)
      {  mant = mant * 10;
         exp--;
      }
}
```

We are now ready for addition and subtraction. In order to add or subtract the operands must have the same exponent. For example, to add `12e3 + 3e4`, i.e. `12000.0 + 30000.0` we cannot simply add the two mantissas (this would produce a mantissa of 15, clearly not correct). Instead we must adjust one of the operands so that the exponents are equal. We choose to adjust the operand with the larger exponent, 3e4 in this case: `3e4 = 30e3`. We can now perform the addition by adding the mantissas and using the exponent of either operand (they are equal) for the exponent of the result:

`12e3 + 30e3 = 42e3`, i.e. `12000 + 30000 = 42000`.

Our addition method for MyFloat is shown below:

```
%%%%%%%%%%%%%% TEST THIS
/** @return the sum when this MyFloat
```

```
  * and the parameter, f, are added.
  */
public MyFloat add (MyFloat f)
{  MyFloat temp = new MyFloat (mant, exp);         // temp copy of this
   MyFloat tempF = new MyFloat (f.mant, f.exp);    // temp copy of f
   temp.adjustAdd (tempF);                         // adjust either this or f
   return new MyFloat (temp.mant + tempF.mant, temp.exp);
}

/** Adjust either this MyFloat or f
  * so that they have the same exponent.
  */
private void adjustAdd (MyFloat f)
{  while (exp > f.exp)
      {   exp--;
          mant = mant*10;
      }
   while (f.exp > exp)
      {   f.exp--;
          f.mant = f.mant*10;
      }
}
```

The subtract method will be almost identical to the add method.

11.2.3 Exercises

1. Show how to create MyFloat objects representing each of the following numbers.
   - 7
   - -12.04
   - 0.00032
   - 6.02x10 sup 23
   - 6.02e23
2. What is the value of 12e23 * 3e10 / 4e12 ?
3. Define a toString() method for MyFloats. The easiest way to do this is simply to include the character 'e'or 'E' between the mantissa and the exponent, producing a result such as `"32e-4"` or `"1e3"`.
4. Improve your toString() method so as to produce more conventional results such as `"0.0032"` instead of `"32e-4"` and `"1000.0"` instead of `"1e3"`. You'll need to decide on an arbitrary criterion to produce output with the "e" notation.

```
/** @return A String representation of this
 *  MyFloat.
 */
public String toString()
```

Hint: Convert the mantissa to a String, then handle each of the following cases separately, in the order shown (LIMIT is a constant (perhaps 6) representing the number of zeroes to be inserted before resorting to the 'e' notation):

Case	mant	exp	result
$exp == 0$	123	0	"123.0"
$exp > LIMIT$	123	12	"1.23e14"
$exp > 0$	123	3	"123000.0"
$-exp > mant.len + LIMIT$	12345	-10	"1.2345e-6"
$-exp > mant.len$	123	-4	"0.0123"
$default$	123	-2	"1.23"

5. Define a subtract method for MyFloats:

```
/** @return the difference produced when
 *  the parameter f is subtracted from
 *  this MyFloat
 */
public MyFloat subtract (MyFloat f)
```

6. Define additional constructors in the MyFloat class to allow convenient ways of constructing MyFloats which which have an exponent of 0, and MyFloats representing zero:

```
MyFloat r = new MyFloat (17);            //    17e0
MyFloat p = new MyFloat ();              //    0e0
```

Hint: Eliminate duplicated code in your constructors by calling another constructor using the keyword *this*. A call to another constructor must be the first statement in your constructor.

7. Define an `equals (Object)` method for MyFloats, so that the client can have a Set of MyFloats. You may assume that this MyFloat and the parameter are both in normal form.

```
/** @return true only if the parameter obj
 *  is a MyFloat and is equal to this MyFloat.
 *  Pre: Both this MyFloat, and the parameter,
 *  are in normal form.
 */
public boolean equals (Object obj)
```

8. Define a hashCode method for MyFloats, so that the client can have a HashSet of MyFloats. You may assume this MyFloat is in normal form.

```
/** @return A hashCode for this MyFloat.
 *  Pre: this MyFloat is in normal form.
 */
public int hashCode ()
```

9. We would like the MyFloat class to implement the `Comparable` interface, so that the client can have a TreeSet of MyFloats. Change the class definition to read as follows:

   `public class MyFloat implements Comparable<MyFloat>`
   This means that a MyFloat can be compared with any other MyFloat for order as well as equality (i.e. greater than, less than). The compiler will require you to include a compareTo method:

```
/** @return a negative value if this MyFloat is less than
 *  the parameter f, a positive value if this MyFloat
 *  is greater than the parameter f, and 0 if they
 *  are equal.
 *  Pre: Both this MyFloat and the parameter f
 *  are in normal form.
 */
public int compareTo (MyFloat f)
```

10. Try the following operation in Java (if using BlueJ, you can use the codepad feature).

    3e200 + 1e0

    - Explain why the result is not perfectly accurate.
    - Try this same operation with MyFloats. If an error occurs, correct it so that a reasonable result is produced.

11. Try the following operation in Java (if using BlueJ, you can use the codepad feature).

    123456789e0 * 123456789e0

    - Explain why the result is not perfectly accurate.
    - Try the same operation with MyFloats. If an error occurs, correct it so that a reasonable result is produced.

11.3 BigNumber

Our third example of a useful ADT is one which allow arithmetic with unlimited precision. A java `int` is represented by 32 bits, and thus has limits (see Integer.MAX_VALUE and Integer.MIN_VALUE). When the result of an arithmetic operation exceeds these limits (known as *overflow*), no Exception is thrown; the program continues to execute with values that are almost certainly not valid. Even if we use the primitive type `long` (64 bits) instead, there are still limits to the values which can be computed and stored.

In this section we will define an ADT named *BigNumber* which will be used to represent whole numbers that are not limited in size. [1]. Numbers with unlimited precision have many important applications in today's world, most notably in the areas of cryptography and computer security. BigNumbers are keeping the internet alive today.

Our strategy is to represent a whole number as a list of decimal digits. Each digit will be in the range [0..9]. For example, the number 423,502 will be represented as the list [4,2,3,5,0,2]. With BigNumbers we will need to find a suitable representation for negative numbers, and here we will follow the example of computer chip design; we will use a representation similar to two's complement.

11.3.1 Constructing BigNumbers

Our BigNumber class will need only one field: the List of digits. We need to decide what kind of List this should be in the Constructor. Recall that ArrayLists are designed to be efficient when accessing specific elements directly, but not efficient when the size of the List is often changed. As far as we can see now, with BigNumbers we will not need to access digits in the middle of the List, but instead will process the digits in sequentially, from low order digit to high order digit. For this reason we choose to implement our List of digits with a LinkedList, noting that if we made the wrong choice it should be easy to change it to an ArrayList.

```
/** A BigNumber represents a whole number with unlimited
 *  precision.
 */
public  class BigNumber
{   private List<Integer> digits
        = new LinkedList <Integer> ();

    /** Default constructor is used in the add
     *  method.
```

[1] Of course every computer has a finite memory, hence there will be limits on the size of a BigNumber, yet our software will be designed so as to impose no limit on the size of a BigNumber, and the client can potentially allow for larger numbers by adding more memory to the computer

```
     */
    BigNumber ()
    {   }

    /** Construct a BigNumber from a String
     *  of numeric characters.
     *  @param num is a String of numeric
     *  characters [0..9]
     */
    public BigNumber (String num)
    {  // Process the digits from low-order to high-order
       for (int i=num.length()-1; i>=0; i--)
           digits.add (num.charAt(i));
    }

}
```

The client can now create very large numbers such as:

```
BigNumber huge = new BigNumber ("234809903939392034982043983 4");
```

Note that the constructor will put the low-order digit (the one at the right, as normally viewed, into position 0 of the List. If you are accustomed to seeing position 0 at the left end of a List, the digits of a BigNumber will appear to be in reverse order.

11.3.2 Adding BigNumbers

We now wish to be able to add BigNumbers. We will do this by adding corresponding digits, beginning at the low-order digit, and working toward the high-order digit, adding in a carry (0 or 1) at each position.

In our add method, we use three loops (not nested). The first loop continues as long as there are more digits in both operands. The second loop continues as long as there are more digits in this BigNumber, and the third loop continues as long as there are more digits in the parameter. At least one of these loops will repeat 0 times. We use `carry` to carry a 1 into the next 'column'. When all loops terminate, we can add the carry at the high order position of the result. At this point we are assuming BigNumbers are not negative.

```
/** @return The sum of this BigNumber and the
 *  parameter, b.
 */
public BigNumber add (BigNumber b)
{   int carry = 0;
    int sum;
    BigNumber result = new BigNumber();
    Iterator<Integer> ittyThis = digits.iterator();
```

```
    Iterator<Integer> ittyB = b.digits.iterator();

    while (ittyThis.hasNext() && ittyB.hasNext())
    {  sum = ittyThis.next() + ittyB.next() + carry;
       carry = sum / 10;
       result.add (sum % 10);
    }
    // This loop may execute 0 times
    while (ittyThis.hasNext())
    {  sum = ittyThis.next() + carry;
       carry = sum / 10;
       result.add (sum % 10);
    }
    // This loop may execute 0 times
    while (ittyB.hasNext())
    {  sum = ittyB.next() + carry;
       carry = sum / 10;
       result.add (sum % 10);
    }
    if (carry == 1)
       result.add (carry);
    return result;
}
```

11.3.3 Subtracting BigNumbers

When subtracting, it is possible to obtain a negative result; consequently we need to think about how to represent negative BigNumbers. Here are a few possibilities:

1. Sign and Magnitude - Store the magnitude of the BigNumber as we are doing currently; also store a boolean which indicates whether the BigNumber is negative.

2. Nines Complement - Negate a BigNumber by subtracting each digit from 9 (similar to ones complement for binary numbers). For example, -04096 would be stored as 95903. Then when adding we would need to add 1 to the result, only if one of the operands is negative.

3. Tens Complement - Similar to two's complement for binary numbers, -1 would be represented by all 9's (9999), -2 would be 9998, -3 would be 9997, etc.

We will use tens complement representation for a few reasons:

- This will simplify subtraction: $a - b = a + (-b)$. All we need now is a negate method, to find -b (though we will need to modify our add method).

- Tens complement is analogous to twos complement for binary numbers. Hence, tens complement will reinforce your understanding of twos complement representation which is used in virtually all digital hardware.

In tens complement representation, the number is negative if and only if the high order digit is greater than 4. To determine the value of a negative number, we can negate it. Here are three algorithms for negating a number in Tens Complement:

- Subtract it from 0. For example, to negate 345:

```
  000          Extend the zeroes
 -345
-------
  655  =  -345
```

- Find the nines complement, and add 1. For example, to negate 345:

```
  999
  345      Nines complement, subtract each digit from nine
 ----
  654
 +  1      Add 1
 ----
  655 =   -345
```

- Scan the digits right to left:

  1. copy zeros
  2. subtract first non-zero digit from ten
  3. subtract all remaining digits from nine

  For example, to negate 3405000:

```
      3405000
          000        copy zeros
         5           subtract first non-zero digit from ten
      659            subtract remaining digits from nine
--------------
      6595000  =  -3405000
```

  If it seems strange to you that 6595000 should be equal to -3405000, remember that 6595000 represents a negative number because its high order digit is greater than 4. Try adding 6595000 + 3404000 (and discard the carry out of the high order digit). The result should be 0, proving that they are complements of each other.

  We prefer this algorithm, and recommend it be used in our BigNumber class.

Here are some other examples of tens complements:

```
-280 = 720
-0509 = 9491
-0500 = 500 = 9500 = 99500
```

Note that a non-negative number may have leading zeros, and a negative number may have leading nines; thus there are multiple represenations of the same number (as with our other abstract data types Rational and MyFloat).

Number	Tens Comp	Alternates	
+3	3	03	003
+10930	10930	010930	0010930
-3	7	97	997
-8	92	992	9992
+499	499	0499	00499
-499	501	9501	99501
+500	0500	005000	000500
-500	500	9500	99500
-501	599	9599	99599

A normal form for BigNumber objects is described as:

- A non-negative BigNumber has a minimum number of leading zeros.
- A negative BigNumber has a minimum number of leading nines.

Before we can implement subtraction of BigNumbers we will need to define the *negate* method. We will use the third negate algorithm described above.

```
/**@return this BigNumber negated, tens complement */
private BigNumber negate()
{   BigNumber result = new BigNumber();
    Iterator <Integer> itty = digits.iterator();
    int d = itty.next();
    while (d==0 && itty.hasNext())      // copy zeros
        {   result.digits.add (0);
            d = itty.next();
        }
    result.digits.add ( (10-d) % 10);           // copy 10 - digit
                                        // ensure digits is not empty.
    while (itty.hasNext())
        {   result.digits.add (9-itty.next());  }    // copy 9 - digit
    return result;
}
```

Now that we have a negate method, the subtract method is easy: $a - b = a + (-b)$.

```
/**  @return the result of subtracting the parameter, b,
 *   from this BigNumber.
 */
public BigNumber subtract (BigNumber b)
// a-b = a+(-b)
{
   return this.add (b.negate());
}
```

However, we will need to make a few minor changes to our add method, now that we are working with negative numbers (this is left as an exercise for the student):

- In the loops which accomodate the fact that one of the operands may have fewer digits than the other operand, we assumed leading zeros for the shorter operand. Now we will have to determine whether the shorter operand is negative; if so, assume leading nines. This can be done easily using a fill digit:

  `int fill=0;`

  If the shorter operand is negative, change the fill digit to 9. Add in the fill digit on each iteration.

- When adding two non-negative operands, the result should be non-negative. If this is not the case (i.e. *overflow* has occurred), append a leading zero. For example, when adding the positive numbers 402+350 you will get 752, which is negative because the high order digit is greater than 4; append a leading zero to produce 0752.

- When adding two negative operands, the result should be negative. If this is not the case (i.e. *overflow* has occurred), append a leading nine. For example, when adding the negative numbers 702+650 you will get 352, which is non-negative because the high order digit is less than 5; append a leading nine to produce 9352.

- A private helper method to determine whether a BigNumber is negative may be helpful; it will merely compare the hight order digit with 5.

Having defined addition and subtraction, how can we define multiplication, division (and mod) using what we have already built? We leave these as exercises for the student; here are some hints:

- Multiplication could be done by repeated addition: $4 * 7 = 7 + 7 + 7 + 7$.

- Remember, both operands are BigNumbers. This means that a loop is needed, and it will be controlled by decrementing the left operand until zero is reached. In the body of the loop the right operand is added into the result.

- If the left operand is negative, work with its negation in order to control the loop.

This is a fairly slow algorithm for multiplication of BigNumbers. A faster algorithm is called *shift and add*:
The multiplier is this BigNumber, and the multiplicand is the parameter.

1. Define a private helper method to multiply this BigNumber by a decimal digit returning the product.

```
/** @return the product of this BigNumber multiplied by
 *  the parameter, i.
 *  @param i is in the range 0..9
 */
private BigNumber multByDigit (int i)
```

2. Use a ListIterator to iterate over the digits in this BigNumber from high order to low order digit.

3. Multiply the parameter by the high order digit, storing the product in a local BigNumber, result.

4. Continue to iterate over the digits in this BigNumber using your ListIterator (using methods hasPrevious() and previous()):

   (a) Shift the result (simply insert a 0 at position 0)

   (b) Multiply the parameter by the next (i.e. previous) digit of this BigNumber, storing the result in a temporary BigNumber.

   (c) Add the temporary BigNumber into the result.

A slow algorithm for division of BigNumbers is fairly easy. Division should in general produce two results: a quotient and a remainder, since we are working with whole numbers. We should design our software so that the client can obtain both of these values without repeating the division. To do this, the divide method should return a List of BigNumbers; the first element in the list is the quotient and the second element is the remainder.

```
/** @return Both the quotient and the remainder when this
 *  BigNumber is divided by the parameter b
 */
public List<BigNumber> divide (BigNumber b)
```

The algorithm for division (this BigNumber is the dividend, and the parameter is the divisor):

1. Copy the dividend to a local BigNumber, dividendTmp.

2. Loop while the dividendTmp is greater than the divisor (use a BigNumber counter, to count the number of times the loop repeats):

   (a) Subtract the divisor from the dividendTmp storing the result in dividendTmp.

3. When the loop terminates, the loop counter is the quotient, add it to the result list.

4. The dividendTmp is the remainder, add it to the resultlist.

Now that you have a BigNumber class it could be tested as shown below (commment out the lines which have not yet been implemented):

```
    public static void main()
    {   Scanner scanner = new Scanner (System.in);      // read from stdin
        BigNumber x,y;
        String input;
        System.out.println ("Enter a Big Number, or Enter to terminate");
        while (scanner.hasNextLine())
            {
// read a big number from keyboard
                input = scanner.nextLine();
                if (input.length() == 0)
                    return;
                x = new BigNumber (input);

// read a big number from keyboard
                System.out.println ("Enter another Big Number");
                input = scanner.nextLine();
                if (input.length() == 0)
                    return;
                y = new BigNumber (input);
                System.out.println ("x+y: " + x.add (y));
                System.out.println ("y+x: " + y.add (x));
                System.out.println ("x-y: " + x.subtract (y));
                System.out.println ("y-x: " + y.subtract (x));
                System.out.println ("Enter a Big Number, or Enter to terminate");
            }
        }
```

11.3.4 Exercises

1. Complete the following table using tens complement representation.

   Hint: When viewing a number in tens complement representation, the first decision to be made is whether the number is positive or negative.

Number	Tens Comp Using 4 digits, if possible	Tens Comp Using as many digits as needed, but no more
0	0000	0
2		
4		
5		
-4	9996	6
-5		
-6		
25		
92		
-25		
-92		
499		
500		
501		
-499		
-500		
-501		
4999		
5000		
-5000		
-5001		
-394920	Not Possible	

2. Complete each of the following operations, assuming tens complement representation (use as many digits as necessary for the result):

(a)

```
  0003 = +3
+ 0097 = +97
------------
```

(b)

```
  4004 = +4004
+ 3097 = +3097
--------------
```

(c)

```
  0003 = +3
- 0097 = +97
------------
```

(d)

```
  9999 = -1
+ 0100 = +100
------------
```

```
  9999 = -1
+ 9998 = -2
------------
```

3. Include a toString() method in your BigNumber class:

```
/** @return this BigNumber as a String. */
public String toString()
```

4. (a) Revise the toString method so that it will handle negative values, in tens complement representation.
   Hint: If the number is negative, negate it, produce the result string, and append a '-' at the beginning.

   (b) Revise the add method so that it works with tens complement values. Use the main method given at the end of this section to test your solution.
   Hint: Use a *fill digit* for the operand that has fewer digits. The fill digit should be 9 if the operand is negative, and 0 otherwise. Discard the carry out of the high order digit. Add a 0 (or 9) at the high order digit if necessary to ensure the correct sign of the result.

   (c) Implement the subtract method. Test subraction using the main method.
   Hint: a - b = a + (-b)

   (d) There are several tens complement representations for the same number:
   `+17 = 17 = 017 = 0017 = 00017 ....`
   `+93 = 093 = 0093 = 00093 ....`
   `-19 = 81 = 981 = 9981 = 99981 ....`
   `-82 = 918 = 9918 = 99918 = ....`
   We can define a *normal form* by choosing the representation which has no unnecessary leading zeros (or nines for negative numbers). In the examples given above, the normal form of each number is shown first. Define a method which will normalize this BigNumber, and test with the main method. All newly created BigNumbers should be normalized.

```
/** Put this BigNumber into normal form.
 *  Eliminate unnecessary leading zeros or nines
 */
private void normalize()
```

(e) Revise the toString() method, if necessary, so that it does not print unnecessary leading zeros.

5. Implement multiplication using repeated addition: 5*3 = 3+3+3+3+3

   The method signature should be:

   ```
   /** @return the product of multiplying this BigNumber by b. */
   public BigNumber multiply (BigNumber b)
   ```

   Hint: Check the signs of the operands first, if they are different you know the sign of the result should be negative. Then negate each operand which is negative, and do the multiplication with non-negative values. Then negate the result if necessary.

6. Improve your multiply method to use a shift and add algorithm as described in this section.

7. Implement division using repeated subtraction. Division should produce two BigNumbers as results: a quotient and a remainder. The signature is:

   ```
   /** @return the quotient and remainder (in that order)
        when this BigNumber is divided by b.
       @throws DivideByZeroException if b is 0.
    */
   public List<BigNumber> divide (BigNumber b)
   ```

   If the dividend and divisor have different signs, the quotient should be negative. The sign of the remainder should be the same as the sign of the dividend.[2]

8. Make improvements to your divide method so that it is faster; this is similar to the shift and add algorithm for multiplication, but instead it will be shift and *subtract*.

[2]Caution: Various platforms do not agree on the correct sign for the remainder when one or both operands is negative. The sign proposed here agrees with the Java virtual machine.

Chapter 12

Algorithms: Sorting and Searching

This chapter contains an introductory discussion of sorting and searching algorithms. This subject is covered more extensively in Data Structures textbooks. We include it here because sorting and searching are included in the College Board's Advanced Placement exam for Computer Science.

An *algorithm* is a well-defined series of steps leading to the solution of a given problem. An algorithm must terminate with a correct solution. Note that the concept of an algorithm is independent of a particular implementation, or programming language used, for that algorithm. For example, we discussed the *sequential search* algorithm in chapter7. The sequential search method could have been written in C++, Python, or any other programming language. Also, it could have been written in a very different way in Java; it could have been a recursive method instead of using a loop. These would all be implementations of the *same algorithm*.

There are some problems which become very complex as the size of the input increases. A classic example is the *traveling salesman* problem: Given a map showing cities and roads connecting the cities, find a shortest path which enables the salesman to visit every city exactly once. There are algorithms to solve this problem, and if the number of cities is 10, they work quite well. However, if there are 1000 cities, your program will take too long to execute. For problems such as this we may choose to use a method which is not guaranteed to produce a correct solution, but which executes quickly enough that we will see it terminate. This kind of solution is called a *heuristic*. It is important to understand that a heuristic is not an algorithm; for problems like the traveling salesman problem a heuristic can be more useful than an algorithm.

12.1 Searching: Binary Search

In chapter 5 we discussed the problem of searching a list for a given target value. This was the `indexOf(Object)` method in the list classes (for the API, see the List interface in the `java.util` package of the java class library). This method performs a *sequential* search for the given object (i.e. the target). It begins at the first element of the list, comparing with the target, and proceeding to each element of the list until it either finds an element which is equal to the target (in which case it returns its index in the list), or reaches the end of the list, in which case it returns -1. If it finds the target, it will therefore return the index of the *first* occurrence of that value (there may be duplicate values in a list).

In this section we discuss an improvement to the sequential search algorithm. If the list being searched is sorted in ascending (or descending) order we can use an algorithm known as *binary search* to locate a given target value.

Imagine that you have an old (paper) telephone book for a nearby city. If you needed to find out who lives at "332 N. Main St", you would have to begin by looking at every entry on page 1, and continue through every entry in the phone book until you either find that address, or reach the end of the book, in which case you would conclude that the address "332 N. Main St." is not in the phone book.

However, if you need to find "Smithson, John" in that phone book, it would be much faster. If the first entry you look at is "Potter, James", then you know that if "Smithson, John" is in the book, it would have to come after "Potter, James". Thus you immediately exclude from consideration those entries coming before "Potter, James". This is an informal description of the binary search algorithm. After each comparison, half of the values are eliminated from consideration; we can ignore them, speeding up the search considerably.

The reason that looking up a name is so much faster than looking up an address is that the phone book is *sorted* by name, but it is *not* sorted by address. Thus the binary search algorithm applies only when the data are sorted prior to beginning the search.

The binary search algorithm can be expressed as follows:
Search a given list for a given target value, given the starting and ending indices of the list being searched

1. If the starting index is greater than the ending index, terminate. The target is not in the list.

2. Calculate the index of the midpoint, by averaging the start and end indices:
   `mid = (start+end)/2`[1]

3. If the target is equal to the value at position `mid`, the target has been found; return `mid`, the position of the target.

[1] Note that if start+end is odd, the result is rounded down to an int. E.g. $15/2 = 7$. This works fine as the midpoint.

4. If the target is smaller than the value at position `mid`, then the target must be in the left portion of the list (if it is in the list). Search the left portion of the list (excluding the mid point) using the binary search algorithm, from `start` to `mid-1`.

5. If the target is greater than the value at position `mid`, then the target must be in the right portion of the list (if it is in the list). Search the right portion of the list (excluding the mid point) using the binary search algorithm, from `mid+1` to `end`.

Note that this algorithm is expressed recursively. The base cases are steps 1-3. The recursive cases, steps 4 and 5, reduce the size of the input; thus it satisfies the desired properties of recursion.[2]

The binary search algorithm is diagrammed in Fig 12.1, in which the sorted list is [-7,-4,2,3,5,7,8,8,11,11,12,13,17,17,22] and the target is 11. On the first iteration, `start=0` and `end=14`. The midpoint is calculated as `mid = (0+14)/2 = 7`. Since the value at position 7 is 8, and the target, 11, is greater than 8, the target must be in the right half of the list; search the portion of the list from `start=8` to `end=14`, using the same binary search algorithm.

At this point the midpoint is calculated as `mid = (8+14)/2 = 11`. The value at position 11 is 13, which is less than the target. Thus if the target is in the list, its position must be in the range [8..10]. Thus on the next iteration `start=8` and `end=10`.

When searching this part of the list, `mid = (8+10)/2 = 9`. At this point the algorithm finds the target,11, at the midpoint, and terminates by returning its position, 9.

Note that the algorithm did not find the position of the *first* occurrence of the target. It is sufficient to return the position of *any* occurrence of the target.

Each time this algorithm invokes the binary search algorithm recursively, the start and end indices get closer together. If the target is not in the list, then the start index becomes greater than the end index, and the algorithm terminates with the base case in step 1. This case is diagrammed in Fig 12.2, in which the target, 4, is not in the list. When `start`=4 and `end`=3, the algorithm determines that the target 4 is not in the list.

A java method for the binary search algorithm is shown in Fig 12.3. The list to be searched is defined as a field in the class, and is assumed to be initialized.[3] We use a recursive helper method, `binSrch` to search the portion of the list from position `start` to position `end` for the target. There are two base cases in `binSrch`:

- `start > end`: The target is not in the list.

- The value at the midpoint is equal to the target. The target has been found; return its position: `mid`.

[2]This algorithm could have been expressed just as easily with a loop; we use recursion simply to gain some additional experience with recursion.

[3]The list should be an ArrayList, for efficiency.

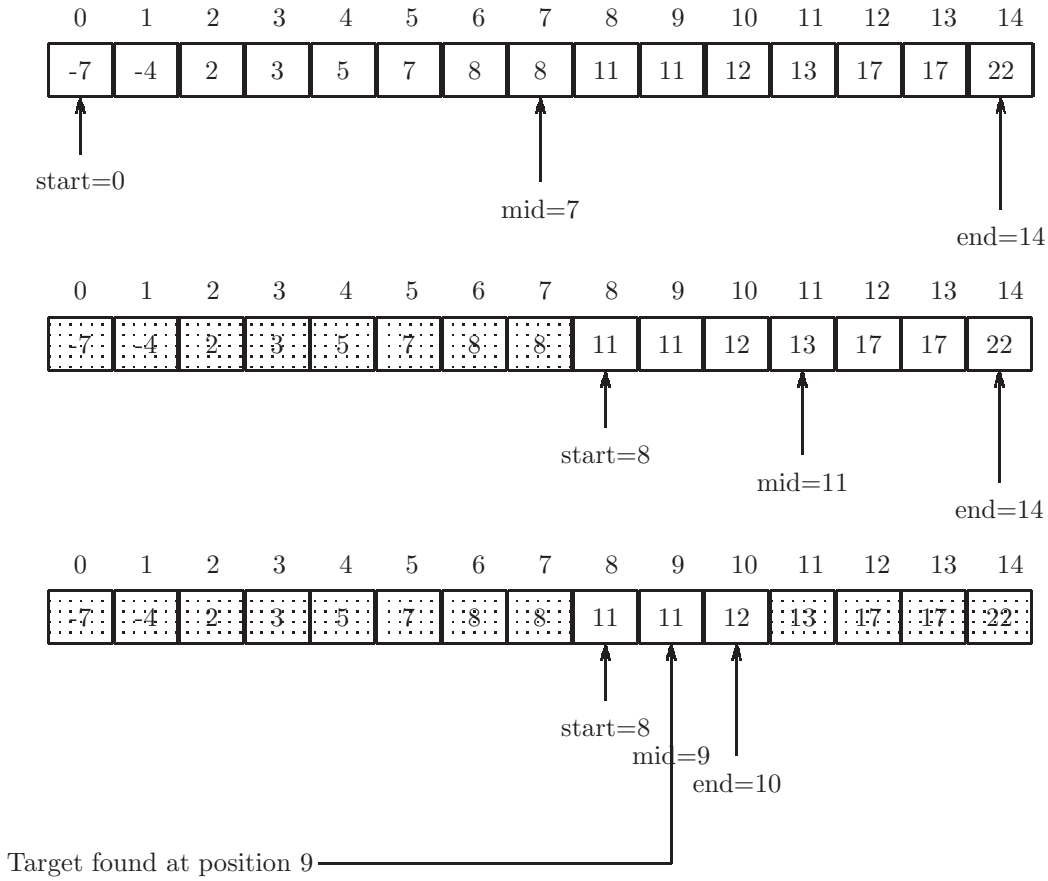
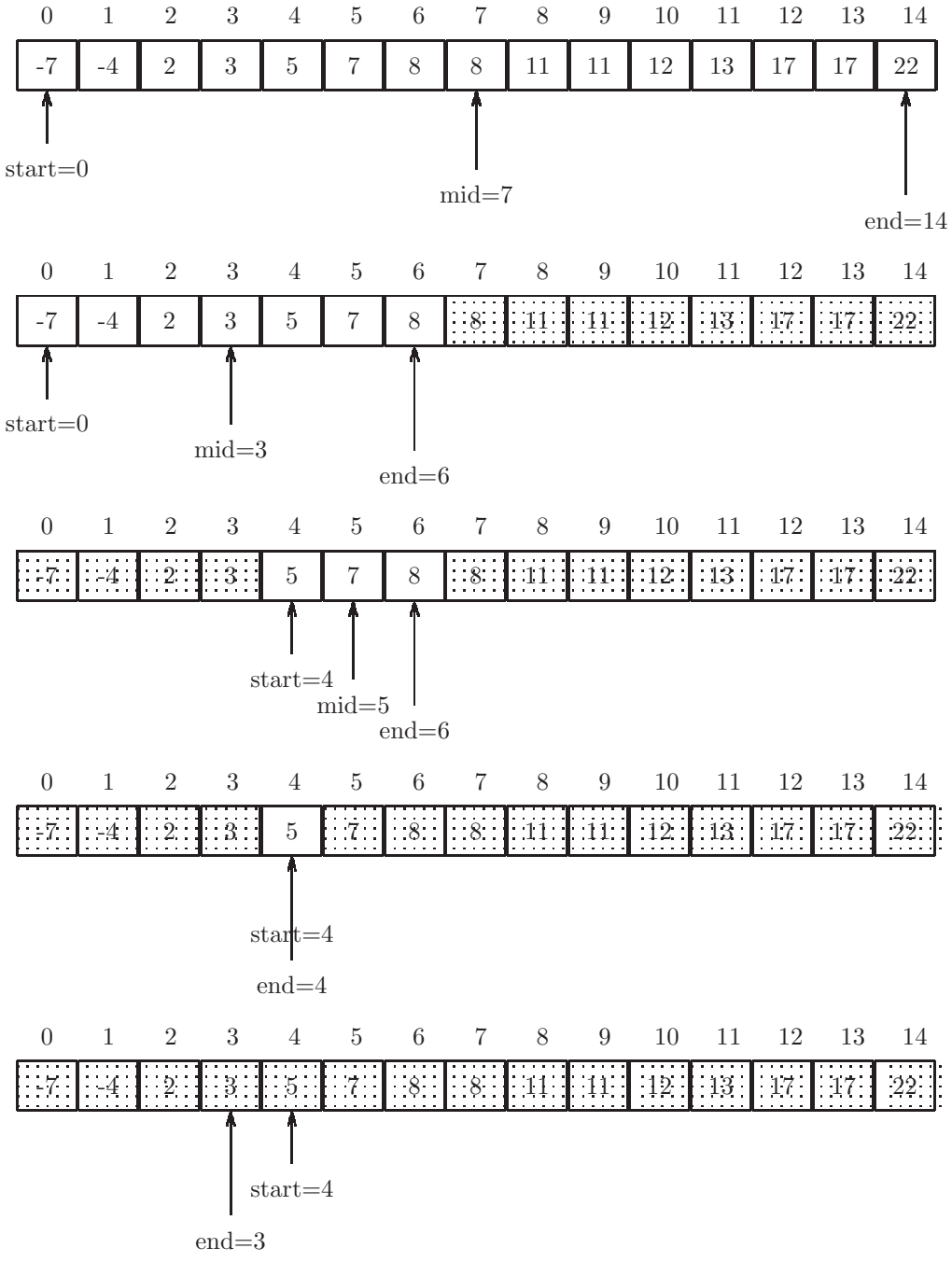


Figure 12.1: Binary Search algorithm, on a list of size 15. The target is 11, found at position 9. There is no need to search the shaded regions.



Target not found

Figure 12.2: Binary Search algorithm, on a list of size 15. The target, 4, is not found in the list.

```
List<Integer> list;     // Should be an ArrayList
   ...
public int search (int target)
   {    return binSrch (0, list.size()-1, target);   }

   /** @return A position of the target in the
    * portion of the list from start .. end, or
    *  -1 if not found,
    */
private int binSrch (int start, int end, E target)
   {    if (start > end)
            return -1;
        int mid = (start + end) / 2;
        if (target == list.get(mid))
            return mid;
        if (target < list.get(mid))
            return binSrch (start, mid-1, target);
        // target > list.get(mid)
        return binSrch (mid+1, end, target);
    }
```

Figure 12.3: Method to search a sorted list of Integers for a given target value, using the binary search algorithm.

There are two possible recursive cases in `binSrch`:

- The target is smaller than the value at the midpoint; search the list from positions `start` thru `mid-1`, inclusive.
- The target is larger than the value at the midpoint; search the list from positions `mid+1` thru `end`, inclusive.

.

12.1.1 Exercises

1. Given the List of Fig 12.1 show the values assigned to the variable `mid` for each of the following target values:
   (a) `target = 8`
   (b) `target = 13`
   (c) `target = 2`
   (d) `target = 15`
   (e) `target = -400`

2. Show a diagram similar to Fig 12.1 for each of the targets given in the previous exercise (and the list given in that figure).

3. Show a binary search method similar to Fig 12.3 in which we are searching a sorted *array* of ints, rather than a sorted List of Integers.

4. Show a binary search method similar to Fig 12.3 which uses a loop rather than a recursive helper method.

5. How many iterations (or calls to `binSrch`) would occur when searching for a target that is not in the given list if:

   (a) The size of the list is 7.

   (b) The size of the list is 15.

   (c) The size of the list is 1023.

   (d) The size of the list is 2^{k-1} for some integer k.

   (e) The size of the list is n.

6. (a) Rewrite the search method in Fig 12.3 to search a List of Strings for a given target String.
   Hints:

   - A String, s1, is smaller than another String, s2, iff s1 precedes s2 alphabetically.
   - See the compareTo method in the String class.

   (b) If you are familiar with generic types in java, and typed classes, rewrite the search method in Fig 12.3 assuming that it is in a class with generic type `E`. The `E` represents any class which implements the `Comparable` interface.

   ```
   public class Search<E extends Comparable>
   ```

12.2 Sorting a list

12.2.1 Rationale for Sorting

In this section we discuss an important problem known as the *sorting* problem: given a list of values which can be compared[4] for order, arrange the list in ascending (or descending) order.

Once a list has been sorted, it can be searched quickly with the Binary Search algorithm. Sorting a long list can take a lot of time, but it is probably worth it if the list is to be searched for many different values.

[4]Comparing two values to determine which is larger, or whether they are equal

12.2.2 Selection Sort Algorithm

There are many algorithms which can solve the sorting problem. We examine one of the easiest to understand and implement in this section. It is called *selection sort*. We describe the algorithm informally below, where `size` represents the size of the list.:

- Scan the list from left to right (`ndx` = 0,1,2,3,...`size`-2)
- Find the position, `p`, of the smallest value in the list beginning at position `ndx`, i.e. search the sublist in positions [ndx..size-1] for the smallest value.
- Swap (i.e. exchange) the values at positions `p` and `ndx`.
- Increment `ndx` and repeat from step 2 until `ndx` = `size`-1.

Fig 12.4 shows how this algorithm is applied to a List of size 5. The list is shown as the value of `ndx` ranges from 0 through 3. When the value of `ndx` is 0, the position of the smallest value to the right (-1) is position 3. Thus the algorithm swaps positions 0 and 3. Note that when the value of `ndx` reaches 1, the position of the smallest value starting from position 1, is also 1. That means that the algorithm swaps position 1 with itself - an unnecessary, but harmless, operation.[5] Note also that for a list of size 5, there are only 4 iterations. After the first 4 iterations, the first 4 values are correctly placed, and the remaining value must be in its correct position.

A Java method to sort a given List using the selection sort algorithm is shown in Fig 12.5. It uses a private helper method, `posSmallest(int start)` to return the position of the smallest value beginning at the given start position.

.

12.2.3 Insertion Sort Algorithm

The *insertion sort* algorithm is similar to the selection sort algorithm, in that it also requires n-1 passes over the input list, for a list of size n. On each pass, p=1..n-1, the algorithm will move the value in position p to the left as many places as needed so that it is inserted in the correct position. To do this, values to the left of position p must be shifted to their righ-hand neighbors, in order to make room for the inserted value.[6] Fig 12.6 shows how the algorithm sorts a List of size 5.

When p=2, the value at position 2 is 19, which is then removed and inserted at position 2 (essentially a no-operation). On that iteration the list is not changed. As with selection sort, we note that after i iterations the first i values of the list are correctly sorted. If n is the size of the list, then after n-1 iterations,

[5] Students often suggest checking for this condition, to avoid an unnecessary swap; however we feel that for random data it will not save a significant amount of time.

[6] We could remove the value at position p, and insert it at the correct spot, but the remove operation has a 'hidden loop' which we prefer to avoid, for reasons of run-time efficiency, and also to facilitate adapting the algorithm to operate on an array rather than an ArrayList.

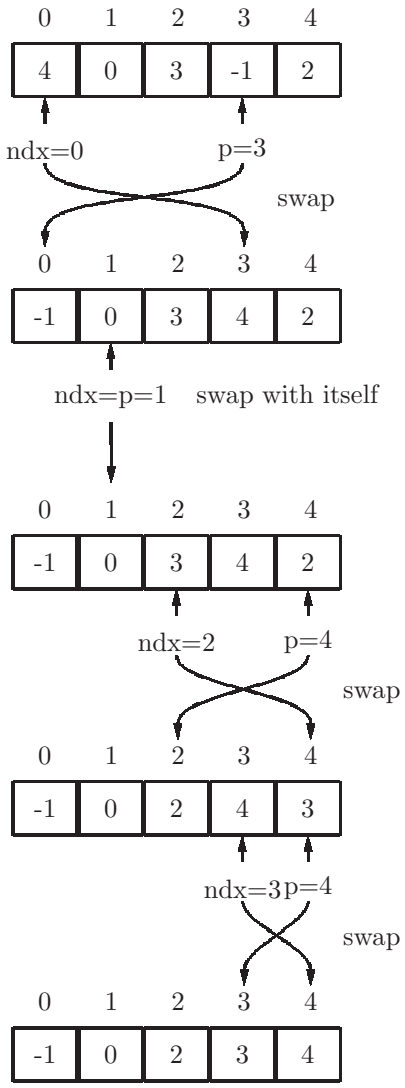


Figure 12.4: Selection sort algorithm, on a list of size 5

```
    /** Post: The values in the List will be arranged in ascending
              order
     */
    public void selectionSort (List <Integer> list)
    {   for (int i=0; i<list.size()-1; i++)
           swap (list, posSmallest (list, i), i);
    }

    /** @return the position of the smallest value,
     *   beginning at the given start position
     */
    private int posSmallest (List<Integer> list, int start)
    {   int smallestPos = start;
        for (int i=start+1; i<list.size(); i++)
            if (list.get(i) < list.get(smallestPos))
                smallestPos = i;
        return smallestPos;
    }

    /** Exchange values at positions i and j
     */
    private void swap (List <Integer> list, int i, int j)
    {   E temp;
        temp = list.get(i);
        list.set (i, list.get (j));
        list.set (j, temp);
    }

}
```

Figure 12.5: Java method implementing the selection sort algorithm, applied to a List of Integers

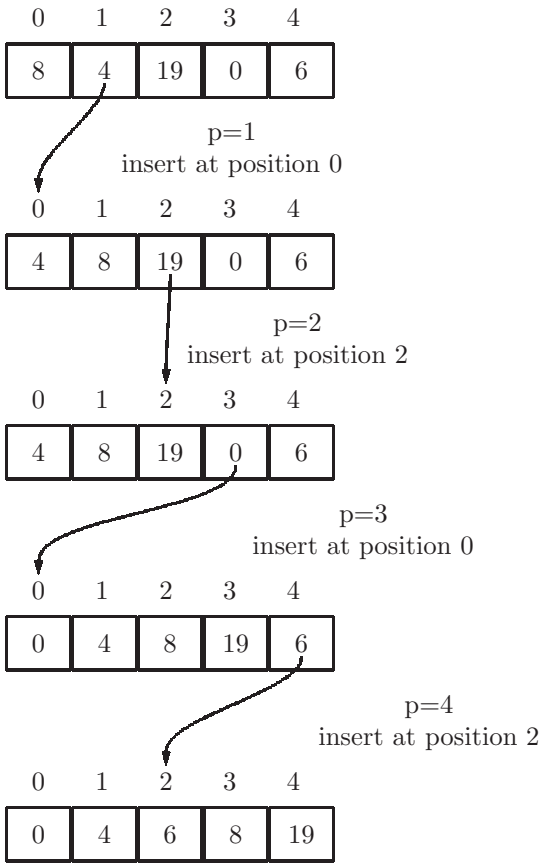


Figure 12.6: Insertion sort algorithm, on a list of size 5

```
/** Post: The values in the List will be arranged in ascending order
 */
    public void sort (List <Integer> list)
    {   int j, tmp;

        for (int p=1; p<list.size(); p++)
           {  tmp = list.get(p);
              for (j = p; j>0 && tmp < list.get(j-1); j--)
                list.set (j, list.get(j-1));         // shift right
              list.set(j, tmp);
            }
    }
```

Figure 12.7: Java method implementing the insertion sort algorithm, applied to a List of Integers

the first n-1 values are correctly sorted, so the last value must be in its correct spot; the loop repeats only n-1 times.

The code for the insertion sort algorithm is shown in Fig 12.7. Note that the inner `for` loop shifts values in the List to their right-hand neighbor:
`list.set (j, list.get(j-1));`
This allows insertion of the value from position p, stored in `tmp`, to be inserted at the correct position.

This example sorts a List of Integers, though it can be readily adapted to sort a List of any type which is Comparable.[7]

12.2.4 Merge Sort Algorithm

The Merge Sort algorithm makes use of one of the oldest techniques in computer science: that of merging two sorted lists. This was done decades ago when large datasets were stored on magnetic tape. If two tapes contained data in ascending order, they could be merged to a single third tape with a simple merge algorithm.[8]

12.2.4.1 Merge Algorithm

The *merge algorithm* takes 2 sorted lists as input; here we assume they are sorted in ascending order. The algorithm produces a sorted list from the values of the two given lists (refer to Fig 12.8 as we describe the algorithm). The two given lists are [3,5] and [2,3,6,7].

1. We compare the first value of the first list with the first value of the second list. The smaller value is added to the result list, and we move to the next

[7] With a generic type that extends Comparable, we can sort any List.

[8] Access to the data on a magnetic tape is sequential, similar to the access to data in a linked list.

value of that input list. In Fig 12.8 since $2 < 3$, we copy 2 to the result and move to the next position of the second list (value is 6).

2. Since $3 < 6$, we copy 3 to the result and move to the next position of the first list (value is 5).

3. Since $5 < 6$, we copy 5 to the result and move to the next position of the first list (reaching the end of the first list).

4. Since we have reached the end of the first list, we copy the remaining values of the second list (6,7) to the result.

The final result is the list [2,3,5,6,7] Fig 12.8 shows how the algorithm merges a sorted list of size 2 with a sorted list of size 3, to produce a sorted list of size 5.

A java method which implements the merge algorithm is shown in Fig 12.9. This method uses two Iterators, one for each input List. Thus it will run efficiently whether the input is an ArrayList or LinkedList. In the loop it chooses the smaller of the two values selected from list1 and list2, and adds it to the result List, after which it obtains the next value from that List. Note that a null value indicates that the end of the corresponding List has been reached. At that point the loop terminates, and the remaining values of the other list, if any, are copied to the result list.

This method uses a few helper methods, which are shown in Fig 12.10:

- `getNext(Iterator<Integer> it)`: Use the given iterator to obtain the next value from one of the input lists, or null if there is none.[9]

- `copyRemainingValues(int value, Iterator<Integer> it)`: Use the given iterator to copy all the remaining values to the result list.

12.2.4.2 Merge-in-Place Algorithm

Our goal is to expose a sorting algorthm which uses the merge algorithm that we have just seen. However, in order for it to be useful (and efficient) in our sorting algoirthm we need to modify the merge algorithm. We will consider the two lists to be merged as residing in the *same* list. We will need to provide the start position of the second list. For example, the lists [3,5] and [2,6,7] can be located in the same list as [3,5,2,6,7] with the start position of the second list at position 2. The result of our merge-in-place algorithm will be the list [2,3,5,6,7]. The Merge-in-Place is depicted with a diagram in Fig 12.11.

A Java method which performs the Merge-in-Place is shown in Fig 12.12. It has one parameter, the starting position of the second list to merged. The inner loop is used to shift values of the first list to the right to make room for the insertion of a value from the second list.

[9]This redesign of the next() method from the Iterator interface leads to a substantially cleaner solution.

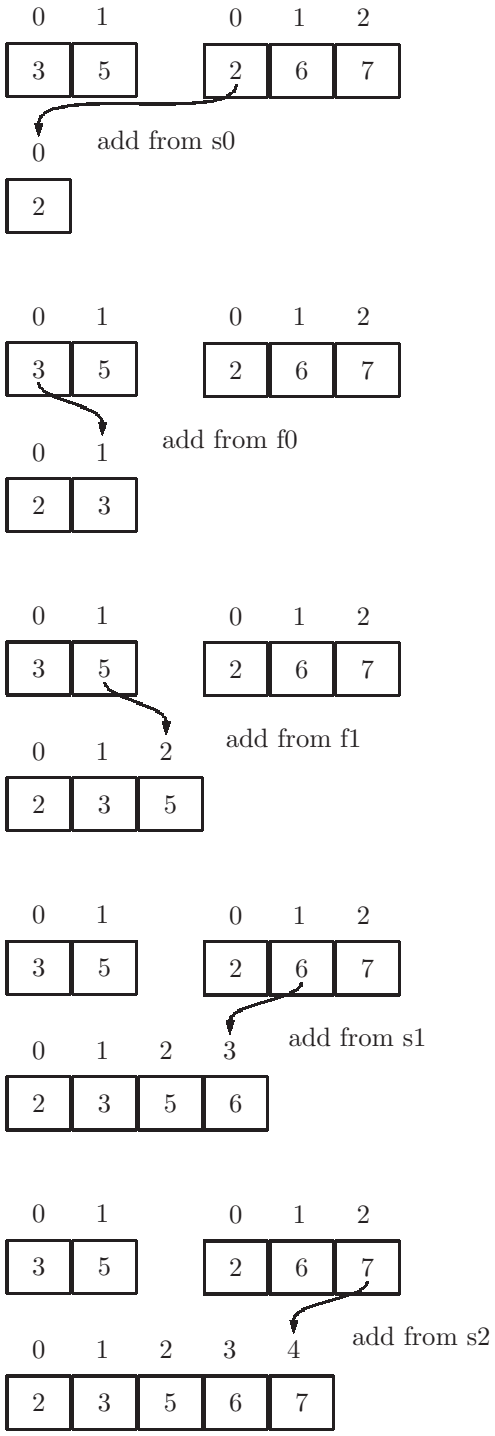


Figure 12.8: Merging two sorted lists into one sorted list. f = first list, s = second list.

```
    List<Integer> result = new ArrayList<Integer>();
    Iterator<Integer> it1, it2;
    Integer value1 = null, value2 = null;

 /**  @return a sorted ArrayList consisting of all values from
  *   the two given lists.
  *   @param first and second are both sorted in ascending order.
  */
public List<Integer> merge (List<Integer> first, List<Integer> second)
{   it1 = first.iterator();
    it2 = second.iterator();
    value1 = getNext(it1); value2 = getNext(it2);
    while (value1!=null && value2!=null)
     {  if (value1.compareTo(value2) < 0)    // add smaller value
             {  result.add(value1);          // to the result.
                value1 =getNext(it1);
             }
         else
             {   result.add(value2);
                 value2 = getNext(it2);
             }
     }
   copyRemainingValues(value1,it1);     // one of these will
   copyRemainingValues(value2,it2);     // do nothing.
   return result;
 }
```

Figure 12.9: Method to merge two sorted lists into one sorted list.

```
   /** Copy the given value to the result list if not null, then copy
     *  the remaining values using the given iterator.
     */
    private void copyRemainingValues(Integer value, Iterator<Integer> it)
   {   while (value!=null)
            {  result.add(value );
               value  = getNext(it);
            }
    }

    /** @return the next value using the given Iterator, or
     *  null if there is none.
     */
    private Integer getNext(Iterator<Integer> it)
     {  if (it.hasNext())
           return it.next();
        return null;
     }
}
```

Figure 12.10: Helper methods for the method which merges two sorted lists into one sorted list (Fig 12.9).

12.2.4.3 MergeSort Algorithm

We now have all the tools we need to implement a sort algorithm known as MergeSort. Given an ArrayList of values which can be compared for larger-vs-smaller,[10] we wish to sort the values in ascending order. Here is the MergeSort algorithm:

1. If the size of the list is 0 or 1, terminate; the list is sorted.
2. If the size of the list is 2, swap the two values, if necessary, so that the smaller is to the left of the larger, and terminate.[11]
3. Find the midpoint of the list. We now have a left part (which includes the midpoint) and a right part (which excludes the midpoint).
4. Sort the left part, using the MergeSort algorithm.
5. Sort the right part, using the MergeSort algorithm.
6. Merge-in-place the two parts, using the Merge-in-Place algorithm described above.

[10] In Java, any class which implements the `Comparable` interface must have a `compareTo(Object)` method, which enables the client to compare not only for equality, but for ordering, i.e. smaller or larger.

[11] Step 2 is actually not needed; we include it to simplify the diagrams.

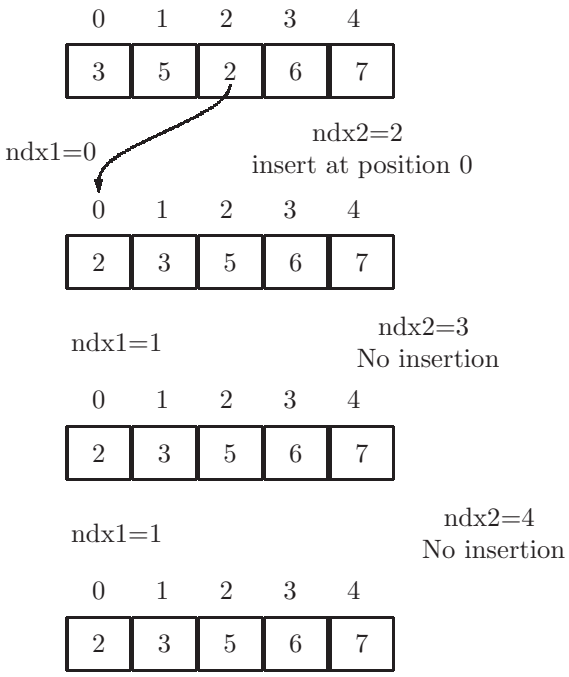


Figure 12.11: Merge in place. The two input lists are [3,5] and [2,6,7], contained in the same list. The second input list begins at position 2. Result is the merged list.

```
List<Integer> list;   //  list is a field
                      // initialized here ...
     /** Merge in place two lists stored in the same list.
      * @param start2 Position of first value in the second list.
      */
 public void mergeInPlace (int start2)
 {  int end = start2 - 1;              // end of first list
    int ndx = 0;                       // position in first list

     while (ndx<=end && end < list.size()-1)
       {   if (list.get(ndx) <=  list.get(start2))
                ndx++;
           else
             {  int value = list.get(start2);
                for (int i=start2; i>ndx; i--)
                     list.set(i, list.get(i-1));         // shift for insert
                list.set(ndx, value);          // insert value from second
                                               // list.
                ndx++;
                start2++;
                end++;
             }
       }
 }
```

Figure 12.12: Method to merge two sorted lists, both in one list, in place, using only one listt.

Notice that the description of the MergeSort algorithm involves a directive to use the MergeSort algorithm in steps 4 and 5. Thus it is a recursive algorithm (see chapter 3). This recursive method satisfies the two basic properties of recursion:

- There is a base case, which involves no recursive call (steps 1 and 2).
- In each recursive call, the size of the input is somehow reduced. In this case we are invoking the MergeSort algorithm on *half* of the given list (steps 4 and 5).

A diagram of this algorithm is shown in Fig 12.13. In that diagram we wish to sort the list [20,18,19,14]. To do that we first calculate the midpoint using the starting and ending positions. Thus the midpoint,
`mid = (0+3)/2 = 1`. The left half of the list is [20,18], and the right half is [19,14]. There is then a recursive call to MergeSort, using `start=0, end=1` to sort the left half. When that completes the left half is [18,20]. The same procedure is then applied to the right half, [19,14], which results in [14,19]. The complete list is now [18,20,14,19], which is merged using position 2 as the starting position of the second list. This merge results in [14,18,19,20], and the list is now sorted in ascending order.

A Java method for the MergeSort algorithm is shown in Fig 12.14. Note that since `mergeInPlace` is called from `msort`, and since `msort` is working on a *part* of the list, `mergeInPlace` will *also* need to know on what part of the list it is working. Therefore, `mergeInPlace` will need a second parameter, `end`, which is the index of the last value in the part of the list being considered.

This completes our discussion of the MergeSort algorithm. For long lists it is significantly faster than both SelectionSort and InsertionSort. However, a detailed analysis and explanation of the efficiency of MergeSort is beyond the scope of this book.

12.2.5 Exercises

1. Show which positions are swapped on each iteration of the main loop in the SelectionSort algorithm when sorting the list shown below:
   $[5, 4, 2, 4, 3, 1]$
2. Show a diagram similar to Fig 12.4 showing how the SelectionSort algorithm sorts the list shown below:
   $[5, 4, 2, 4, 3, 1]$
3. In the SelectionSort algorithm shown in Fig 12.5 there is a private helper method, `posSmallest(List,int)`. That method has a loop in which it compares two elements in the List being sorted. How many comparisons, total, are made if the size of the List, n, is:
   (a) n = 4
   (b) n = 5

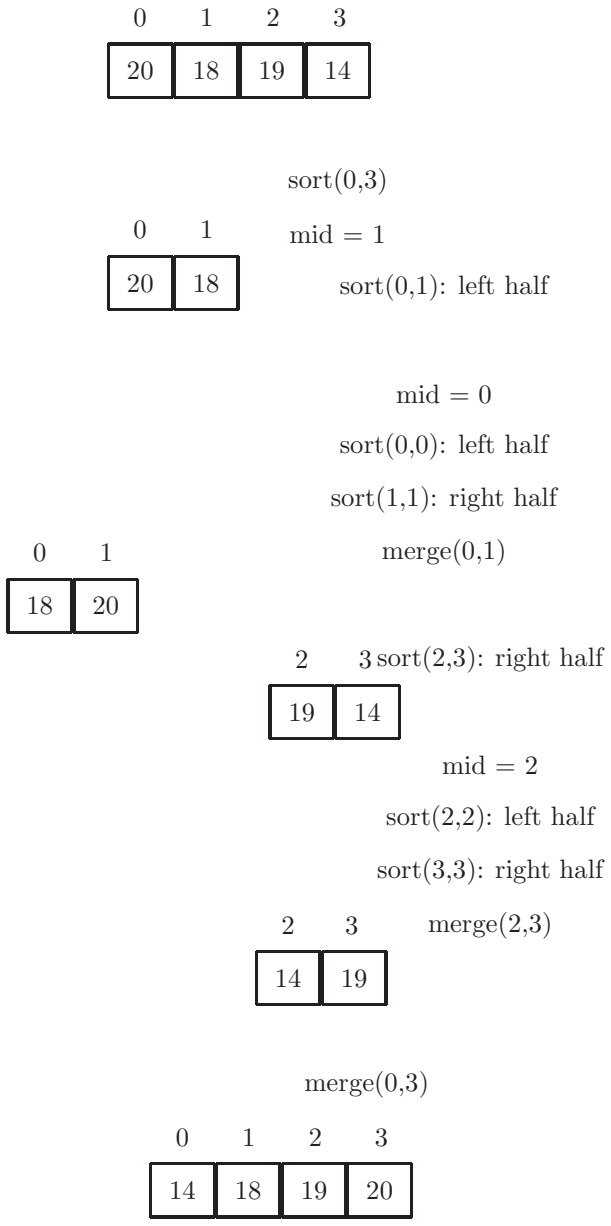


Figure 12.13: Merge sort algorithm, on a list of size 4

```
    // sort increasing, using mergeSort algorithm
    public void sort (List <E> list)
    {   this.list = list;
        msort (0, list.size()-1);
    }

   // Recursive helper method
   // Sort the portion of the list beginning at start
   // and ending at end.
    private void msort (int start, int end)
    {   if (end-start < 1)                    //  base case, size = 1
            return;
        if (end-start < 2)                    //  base case, size = 2
            if (list.get(start).compareTo(list.get(end)) > 0)
            {   swap (start,end);
                return;
            }
        int m = (start+end) / 2;              // midpoint
        msort(start,m);                       // sort left half
        msort(m+1,end);                       // sort right half
        mergeInPlace(start,end);              // merge the two halves
    }

private void mergeInPlace (int start, int end)
    {  int m = (start+end)/2;
       int ndx1 = start, ndx2 = m+1;

       while (ndx1<=m && m<end )
          {   if (list.get(ndx1).compareTo(list.get(ndx2)) <= 0)
                ndx1++;
              else
                {   E value = list.get(ndx2);
                    for (int i=ndx2; i>ndx1; i--)
                        list.set(i, list.get(i-1));         // shift for insert
                    list.set(ndx1, value);
                    ndx1++;
                    ndx2++;
                    m++;

                }
          }

     }
```

Figure 12.14: Method to sort a list using the MergeSort algorithm; mergeInPlace now requires two parameters.

(c) n = 6

(d) n = 1000

(e) n

4. Show an implementation of the SelectionSort algorithm which operates on an array of ints rather than a List of Integers.

5. Show an implementation of the SelectionSort algorithm which operates on a List of Strings rather than a List of Integers.

6. If you are familiar with generic types in java, and typed classes, rewrite the sort method in Fig 12.5 assuming that it is in a class with generic type `E`. The `E` represents any class which implements the `Comparable` interface.
   `public class Sort<E extends Comparable>`

7. Refer to the InsertionSort algorithm. Given the List shown below, show the position at which each value at position p is inserted as p is incremented from 1 through 8:
   $[9, 12, 4, 6, 9, 2, 0, 8, 1]$

8. Show a diagram, similar to Fig 12.6, showing how the List given below is sorted by the InsertionSort algorithm:
   $[9, 12, 4, 6, 9, 2, 0, 8, 1]$

9. In the InsertionSort algorithm shown in Fig 12.7 there is a a call to the `set` method on the List being sorted:
   `list.set(j,list.get(j-1))`.
   in the inner loop. How many times is that `set` method called for a List (initially in descending order) of size:

   (a) 3

   (b) 4

   (c) 5

   (d) 1000

   (e) n

10. Show an implementation of the InsertionSort algorithm which operates on an array of ints rather than a List of Integers.

11. Show an implementation of the InsertionSort algorithm which operates on a List of Strings rather than a List of Integers.

12. If you are familiar with generic types in java, and typed classes, rewrite the sort method in Fig 12.7 assuming that it is in a class with generic type `E`. The `E` represents any class which implements the `Comparable` interface.
   `public class Sort<E extends Comparable>`

13. The private helper method, `msort` in the MergeSort algorithm shown in Fig 12.14 is a recursive method, with parameters `start` and `end`. Show the parameter values and the value computed for the local variable `m` on each call to msort for the List [9,2,8,3,0,7,5]

14. The private helper method, `msort` in the MergeSort algorithm shown in Fig 12.14 is a recursive method. How many times is that method called for a List of size:

    (a) 7

    (b) 15

    (c) 31

    (d) n

15. Show an implementation of the MergeSort algorithm which operates on an array of ints rather than a List of Integers.

16. Show an implementation of the MergeSort algorithm which operates on a List of Strings rather than a List of Integers.

17. If you are familiar with generic types in java, and typed classes, rewrite the sort method in Fig 12.14 assuming that it is in a class with generic type `E`. The `E` represents any class which implements the `Comparable` interface.

    ```
    public class Sort<E extends Comparable>
    ```

Glossary

! - Logical NOT operator

& - Bitwise AND operator

&& - Logical AND operator

| - Bitwise OR operator

|| - Logical OR operator

˜ - Bitwise NOT operator

abstract - Having no evident details; non-concrete

abstract class - A class which has one or more abstract methods, and cannot be instantiated

abstract data type - A collection of data with associated operations; ADT

abstract method - A method which has no body and must be defined in a subclass

abstraction - The process of separting ideas from specific instances of those ideas

access mode - A specification of which classes can refer to a particular class, method or variable: public, [default], protected, and private

accessor method - A method with the purpose of obtaining the value of a particular field

Action - A class which is capable of generating an event in a GUI

ActionListener - An interface for the handling of actions, such as a button click or menu selection, in a GUI

actionPerformed - A method in the ActionListener interface which handles actions

actual parameter - A parameter value to be passed to a called method

ADT - Abstract data type

algorithm - A well-defined sequence of steps to solve a given problem, which terminates with a correct solution

AND - A boolean operation which results in `true` only if both operands are `true`

ArrayList - A List which can efficiently get and set a value at a particular position, or index

ASCII - American Standard Code for Information Interchange

American Standard Code for Information Interchange - An 8-bit numeric code for each character; a subset of Unicode

API - Application Program Interface

application program interface - The information needed to use an entity, such as a class, package, or program; API

ArithmeicException - An Exception indicating that a non-valid arithmetic operation, such as a division by zero, has occurred at run time

array - A homogeneous collection of values which is mapped directly to the computer's main memory

assertion - A statement of the program's current state, at run time, for purposes of verification

assignment - The binding of a data value with a variable

awt - Abstract window toolkit; package used for graphics applications

BigNumber ADT - An ADT for whole numbers with unlimited magnitude

binary search - A search algorithm in which the number of values which need to be examined is proporitional to the the logarithm of the size of the collection being searched; a fast search algorithm

BinaryTree - A Tree in which each value has two children

bit - A binary 0 or 1; a binary digit

BlueJ - An IDE used to develop java software

BorderLayout - A LayoutManager with five regions: NORTH, SOUTH, EAST, WEST, and CENTER

Byte - Wrapper class for the primitive type `byte`

byte - A data type for whole numbers using 8-bit two's complement representation

byte - 8 bits

boolean - A data type with only two possible values: `true` and `false`

Boolean - Wrapper class for the primitive type `boolean`

catch - The process of handling a thrown Exception

central processing unit - That portion of the computer's hardware which is capable of performing calculations and making decisions; CPU

char - A data type for characters, such as those on the keyboard, using ASCII

Character - Wrapper class for the primitive type `char`

checked Exception - An Exception which must either be caught (in a try/catch statement) or declared to be throwbn

class - A template defining the composition of a data object

ClassCastException - An Exception indicating that a reference is not being casted correctly at run time, to a subclass or subtype

class method - A (static) method which applies to a class

class variable - A (static) variable shared by all objects of a class

client - Software needing services from other software

close (a file) - Discontinue use of, or relinquish access to, a file which has been opened

collection - An object consisting of a variable number of objects

command line - A user interface in which words are typed on a keyboard for input

comment - A programmer-supplied description, ignored by the compiler

compile-time error - An error in a program which is detected by the compiler

compiler - A program which translates a program written in a high-level language to an equivalent program in machine language

Component (graphics) - A graphical entity, or Container, which may be included in a Container

compound statement - A block of statements enclosed in curly braces

concrete - Having exposed implementation details; not abstract

ConcurrentModificationException - An Exception indicating that a value in a collection is being changed as an iteration through the collection is occurring

console application - A program in which the user interacts with a keyboard and text display

constant - A data value supplied by the programmer

Container - An awt class which enables storage of multiple components and/or containers

contentPane - A Container for the components of a JFrame

control structure - A programming construct enabling an altered flow of execution

constructor - A method used to initialize the fields of an object when it is created

CPU - Central processing unit

data file - Information stored on a secondary storage device, such as disk or flash memory

De Morgan's Laws - Boolean identities: The negation of a conjunction is the same as the disjunction of the negations; The negation of a disjunction is the same as the conjunction of the negations

declaration - A definition of a variable or method

[**default**] **access** - Access is permitted from any class in the same package

double - A data type for numbers using 64-bit floating point representation

Double - Wrapper class for the primitive type `double`

do while statement - An iteration structure which does not specify the number of times the loop body is to be executed; a post-test loop

duplicated code - Program code which is duplicated verbatim in several places in a program

dynamic method lookup - The process of determining which of several functions having the same name is being called, at run time

Event - A state caused by an external action such as mouse move or keyboard entry

Exception - A class which is used to manage errors or other unexpected occurrences at run time

expression - A variable, a constant, or an operation on two expressions

extends - Specification of a subclass relationship

extremum problem - The problem of finding a minimum or maximum value in a collection of values

field - A data value belonging to an object (non-static)

FileReader - A class in the java.io package enabling input from a data file

FileWriter - A class in the java.io package enabling output to a data file

final - Cannot be changed as the program executes

finally - A clause in a try/catch statement which allows the handling of an Exception when no specified catches apply

file - Data stored on a secondary storage device, such as disk or flash memory

float - A data type for numbers using 32-bit floating pointrepresentation

Float - Wrapper class for the primitive type `float`

floating point - A approximate data representation for numbers which need not be whole numbers, and which may be very large, or very close to 0

FlowLayout - A LayoutManager which arranges components from one row to the next in available space

for statement - An iteration structure defining the number of times the loop body is to be repeated

for-each statement - An iteration structure associated with a collection

formal parameter - A parameter in a method declaration

Frame - See JFrame

free format - A lexical property: white space is ignored by the compiler

generic type - A variable type to be filled in at compile time

get - The operation of obtaining a value from a collection or Map

graphical user interface - A user interface in which icons, and other images on a display, and mouse or touchpad are used to interact with a program

GridLayout - A LayoutManager in which components are arranged in rows and columns

GridWorld - Case study developed by the Educational Testing Service for the Computer Science Advanced Placement course

GUI - Graphical User Interface

has-a - Composition relationshp; field within a class

HashMap - An implementation of the Map interface using a HashTable

HashSet - A Set implemented with a HashTable

HashTable - A structure storing many values enabling quick access

heuristic - A series of steps which attempts to solve a given problem but which may terminate with an incorrect, or approximate, solution

high-level language - A language such as Java which enables humans to develop software; a programmming language

IDE - interactive development environment

if statement - A one-way selection structure

if-else statement - A two-way selection structure

implements - Specification of a (java) interface to be implemented

IndexOutOfBoundsException - An Exception indicating a non-valid position being accessed in a collection

inheritance - The establishment of a class hierarcy resulting from a subclass-superclass relationship

initialization - The assignment of a value to a variable when it is first declared

input - Data supplied by the user of a program

instance (of a class) - An object

instance method - A (non-static) method which applies to an object

instance variable - A variable owned by a particular object

interactive development environment - software used to edit, compile, and test programs being developed

instantiate - To create an object, or instance, of a particular class

int - A data type for whole numbers using 32-bit two's complement representation

Integer - Wrapper class for the primitive type `int`

interface - An adapting layer between two or more entities (see applicaton program interface, java interface, and user interface

IO - Input and output

IOException - An Exception involving input and/or output; checked

is-a relationship - Subclass relationship

iteration structure - A selection structure permitting repeated exection of a statement; a loop

Iterator - A class which enables access to each of the elements of a Collection; also enables selective removal of values from a Collection

java - A high level programming language developed by Sun Microsystems, later acquired by the Oracle corporation; a command to execute a compiled java program

javac - A command to compile a java program

java interface - Specification of operations on data

JFrame - A swing class defining an application's extent and components

KeyListener - A class which can handle keyboard events in a GUI

LayoutManager - Class in awt which automatically arranges the components in a Container

LinkedList - A List which can change size efficiently (insert and/or remove values)

List - A Collection in which order is maintained, and duplicate values are permitted

Listener - A class which is capable of reacting to an event in a GUI

long - A data type for whole numbers using 64-bit two's complement representation

Long - Wrapper class for the primitive type `long`

loop - An iteration structure

loop body - The statement to be executed repeatedly in a loop

machine language - The language of binary coded instructions which can executed by the CPU

main - Name of the starting method for a program

Map - An interface defining an ADT for quick access to values using associated (unique) keys

method - A programmer-defined operation to be performed on an object or class

method abstraction - The ability to see the important aspects of a method, without being exposed to its underlying detailed code, which may involve calls to other methods

MouseListner - A class which can listen for Mouse events, such as movement, click buttonDown, in a GUI

multiline comment - A comment initiated with /* and ended with */

multiple inheritance - The ability, or property, that a class may have more than one superclass

mutator method - A method with the purpose of changing the value of a particular field

MyFloat ADT - An ADT (designed for this textbook) which mimics floating point data types and is capable of doing arithmetic

nested loop - A loop defined to be entirely in the loop body of another loop

NOT - A boolean operation which results in the logical complement of its operand

NullPointerException - An Exception caused by the dereferencing of a null reference

object - Data values in memory with a predetermined structure; an instance of a class

object diagram - A drawn diagram depicting the fields of an object, and their current values

one-way selection - A selection structure with only one possible choice of execution paths; an `if` statement

open (a file) - Determine correct access to a file and prepare for input and/or output

operation - A calculation on one or two data values, producing a new data value, with possible side effects

OR - A boolean operation which results in `false` only if both operands are `false`

output - Data produced by a program for a user

overriding methods - The process of redefining a method from a superclass

package - A group of associated classes

parameter - A variable used to send information to a method

pixel - One of the small dots making up an image; a picture element

post-test loop - An iteration structure in which the loop body is executed once before the termination condition is tested

polymorphism - The capability of exhibiting different behaviors at run time, generally enabled by inheritance

pre-test loop - An iteration structure in which the termination condition is tested before the first execution of the loop body

primitive type - A data type included in the java programming language

program - A sequence of binary coded instructions in the computer's memory

programming language - A language such as Java which enables humans to develop software; a high-level language

protected - Access is permitted from any subclass or any class in the same package

private - Access is not permitted from any other class

program - A sequence of binary coded instructions stored in the computer's memory

public - Access is permitted from any class

public static void main - Specification of the starting method for a program

put - The operation of adding a value to a HashTable or Map

Rational ADT - An ADT which can do arithmetic with non-whole numbers

recursive method - A method which calls itself

reference - The memory location of a data object

reference type - A data type defined by a class

return type - The type of data to be returned by a method

return statement - A statement intended to terminate the execution of a method, with a possible value to be sent to the calling method

run time - The execution of a program, as opposed to the compilation

run-time error - An error in a machine language error, detected when the program is executing

RunTimeException - An Exception which the programmer may ignore; unchecked

Scanner - A class in the java.util package used for input, pattern recognition, etc.

scope (of a variable) - The range of statements over which a variable has meaning

search - The problem of finding a given target value in a collection of values

selection structure - A programming construct enabling a program to take one of a few possible execution paths

sequential search - A search algorithm which examines all elements of a collection until the desired value is found, or determined not to be in the collection

server - Software providing data or computation for other software

Set - A Collection in which ordering of the values is not required, and in which there are no duplicates

set - The operaion of changing a value in a collection

short - A data type for whole numbers using 16-bit two's complement representation

Short - Wrapper class for the primitive type `short`

short circuit evaluation - An optimization of a selection structure resulting from the evaluation of a single operand

side effect - A change in a program's state, or output, resulting from an operation

signature - The part of a method defining the access mode, return type, method name, and parameter list

single line comment - A comment initiated with // and ended at line-end

sorting - The process of arranging the values in a collection in ascending (or descending) order

statement - An assignment operation, or method call, followed by a semicolon, a selection statement, a loop statement, or a compound statement

static - Describing a field or method which applies to a class, not an object

static field - A data value belonging to a class; a class variable

static method - A method invoked on a class; a class method

stderr - Standard error file; defaults to the user's console display

stdin - Standard input file; defaults to the user's keyboard

stdout - Standard output file; defaults to the user's console display

String - A class in the java.lang package containing operations on strings

string - Data consisting of a sequence of characters

swing - An updated package for graphics; in javax

TextListener - A class which can handle text entry from the keyboard in a GUI

throw - A statement which interrupts execution to indicate that an Exception has occurred at run time

toString() - A standard method used to convert a data object to a String representation

Tree - A storage structure in which each value is associated with other values, called the 'children'

TreeMap - An implementation of the Map interface using a BinaryTree

TreeSet - A Set implemented with a BinaryTree, in which natural ordering of the values is maintained

try - A statement which enables a thrown Exception to be handled at run time

type - classification of data, such as int, char, String, ...

two's complement - A binary representation system for negative, as well as positive, whole numbers

two-way selection - A selection structure with a choice of two possible execution paths; an `if - else` statement

type conversion - The transformation of a data value to a different type

user interface - Hardware and/or software used for human interaction with a device or program

variable - A name representing a memory storage location for a primitive value or a reference to a data object

visibility - Accessibility in a class

void method - No value is to be returned

while statement - An iteratioon structure which does not specify the number of times the loop body is to be executed; a pre-test loop

wrapper class - A predefined class in the java.lang package whose objects store only the value of a particular primitive type

Index