Chapter 10  Writing Classes
10.1 Designing Classes

- As programs get larger, good design becomes increasingly important.
- One of the most popular techniques for designing Java programs is known as object-oriented design (OOD).
- The objective of OOD is to design a program by determining which classes it will need.
Object-Oriented Design

- The first step of OOD is to identify an initial set of classes, along with the methods that these classes will need to provide.
- During the design of the initial classes, the need for other classes will often arise.
- Classes that might be discovered during the design of a bank account class:
  - A class that stores information about the owner of an account
  - A class that stores a series of transactions
Object-Oriented Design

- Each class may give rise to more classes, which in turn give rise to still more classes, and so on.
- The process of identifying new classes continues until no more classes are needed beyond the ones provided by the Java API.
Object-Oriented Analysis

- The OOD process begins with a specification of what the program needs to do.
- The initial classes are then identified by a process called object-oriented analysis.
- One of the simplest ways to do object-oriented analysis involves finding nouns and verbs in the program’s specification.
- Many of the nouns will represent classes, and many of the verbs will represent methods.
- This simple technique doesn’t work perfectly, but it can be useful for generating ideas.
Object-Oriented Analysis

- **Nouns in the specification for a banking system:**
  - Customer
  - Account

- **Verbs in the specification for a banking system:**
  - Open (an account)
  - Deposit (money into an account)
  - Withdraw (money from an account)
  - Close (an account)

- **This analysis suggests the need for** Customer and Account **classes, with the Account class having open, deposit, withdraw, and close methods.**
Design Issues

- Issues that arise when designing a class:
  - What should the name of the class be?
  - What variables will the class need?
  - What methods should the class provide?

- Although there’s no algorithm for designing classes, there are a number of guidelines to follow.
Naming Classes

- When naming a class, start with a noun (Account, Font, Fraction, Graphics, String).
- If the noun isn’t descriptive enough by itself, include an adjective (FontMetrics, SavingsAccount).
- A participle (a word ending in “ing”) can also be used to clarify a noun (CheckingAccount).
Instantiable or Not?

- Java’s classes fall into one of two categories:
  - *Used for object creation.* Classes in this group contain instance variables and (in most cases) instance methods.
  - *Not used for object creation.* Classes in this category contain class variables and/or class methods, but no instance variables or methods.

- A class that will be used for creating objects is said to be *instantiable.*
Classes that aren’t instantiable have relatively few uses. Most fall into one of two categories:

- **Collections of related constants and/or methods.** The Math and System classes belong to this category.
- **“Main programs.”** Many classes are created solely as a place to put main and its helper methods.
Mutable or Immutable?

- The instances of an instantiable class can be either mutable or immutable.
- The state of a **mutable** object can be changed (its instance variables can be assigned new values).
- The state of an **immutable** object cannot be changed after the object has been created.
- Instances of most classes are mutable. However, some classes (including String) produce immutable objects.
Mutable or Immutable?

- The advantage of making objects immutable is that they can be shared safely.

- Assigning one `Account` variable to another is risky because both will refer to the same `Account` object:

  ```java
  Account acct1 = new Account(1000.00);
  Account acct2 = acct1;
  ```

  Performing an operation such as
  ```java
  acct1.deposit(100.00);
  ```
  will affect `acct2` as well.

- Because `String` objects are immutable, assigning one `String` variable to another is perfectly safe.
Mutable or Immutable?

- The disadvantage of making objects immutable is that programs are potentially less efficient.
- Since objects can’t be changed, new objects will have to be created frequently, which can be expensive.
- The class designer must weigh the safety of immutable objects against the greater efficiency of mutable objects.
10.2 Choosing Instance Variables

- Unlike other kinds of variables, instance variables are stored in every instance of a class.

- **Instance variables can be declared** public or private.
  - **private** variables are visible only to methods in the same class.
  - **public** variables are visible to methods in other classes.
How Many Instance Variables?

- Deciding what the instance variables should be depends on what information is necessary to represent the state of an object.
  - Some objects (fractions, for example) have very little state.
  - Others (such as bank accounts) must store a large amount of information.
- It’s usually best not to have a large number of instance variables in a class.
- Often the number of instance variables can be reduced by creating additional classes.
How Many Instance Variables?

- It’s not a good idea for an `Account` class to use separate instance variables to store the owner’s name, address, telephone number, and so on.
- A better choice: create a `Customer` class and have one of the instance variables in the `Account` class be a reference to a `Customer` object.
- To help reduce the number of instance variables, avoid storing unnecessary data in objects.
- There’s no point in storing an interest rate in every `SavingsAccount` object if all accounts have the same interest rate.
Deciding which instance variables to put in a class is not that critical, because of an important principle known as information hiding—limiting access to the variables in a class by making them private.
Public Instance Variables

- Declaring instance variables to be private rather than public has a number of advantages.
- However, in some cases it makes sense for a class to have public instance variables.
- Consider the Point class, which belongs to the java.awt package.
- Every Point object contains two instance variables, x and y, which represent x and y coordinates.
- Both variables have type int, and both are public.
Public Instance Variables

- The value of a public instance variable can be accessed by writing an object name, a dot, and then the name of the variable:

  ```java
  Point p = new Point(10, 20);
  System.out.println("x coordinate: " + p.x);
  System.out.println("y coordinate: " + p.y);
  ```

- The Dimension class also belongs to the java.awt package.

- A Dimension object contains two public instance variables, width and height, which are both of type int.
Public Instance Variables

- Dimension objects are convenient for keeping track of the size of windows and other visual components.
- The getSize method returns a Dimension object containing the width and height of any window, including a DrawableFrame object.
- An example of determining the width and height of a frame stored in the df variable:

```java
Dimension dim = df.getSize();
System.out.println("Width of frame: "+ dim.width);
System.out.println("Height of frame: "+ dim.height);
```
Public Instance Variables

- Public instance variables make classes such as `Point` and `Dimension` easy to use.
- For example, the `x` coordinate for a point `p` can be obtained by writing `p.x` instead of `p.getX()`.
- Public instance variables also save a little time, because no method call is required to access or change the value of an instance variable.
Information Hiding

- Despite the convenience of public instance variables, it’s usually best for instance variables to be private.
- This technique, known as information hiding, has several significant advantages.
- The most important one is the ability to make changes to those variables.
- Since private instance variables cannot be used outside their own class, it’s possible to rename them, change their types, or even remove them without affecting program code outside the class.
Information Hiding

- Other important advantages of making variables private:
  - Private variables are protected against corruption from methods outside the class.
  - Classes are easier to test.
  - Methods that modify private variables can check the new values for validity.
Information Hiding

- In a well-designed class, methods that modify instance variables will check the validity of any values that are assigned to those variables.
- In addition, constructors will ensure that instance variables are initialized to reasonable values when an object is created.
Information Hiding

- Information hiding is also relevant for methods.
- Rules for getting the greatest benefit from information hiding:
  - **Make variables private.** If access to a variable is needed outside the class, provide a method that returns the value of the variable and/or a method that changes the value of the variable.
  - **Make a method private if it’s used only by other methods in the class.**
The “Hidden Variable” Problem

- Each variable and parameter in a Java program has a **scope**: the region of the program in which it can be used.
- The variable or parameter is **visible** within this region.
- **Java’s visibility rules:**
  - Variables declared inside a class declaration—including both instance variables and class variables—are visible throughout the class declaration.
  - A parameter is visible throughout the body of its method.
  - A local variable is visible from the line on which it is declared to the end of its enclosing block.
The “Hidden Variable” Problem

- A scope example:

```java
public class ScopeExample {
    ...
    private int i;
    ...
    public void f(int j) {
        ...
        int k;
        ...
        if (i > j) {
            ...
            int n;
            ...
        }
    }
}
```
The “Hidden Variable” Problem

- A variable in an inner scope with the same name as a variable in an outer scope will “hide” the outer one.

- An example of the “hidden variable” problem:

```java
public class HiddenVariableExample {
    private static String str = "Original value";

    public static void main(String[] args) {
        String str = "New value";
        displayString();
    }

    private static void displayString() {
        System.out.println(str);
        // Displays "Original value"
    }
}
```
The “Hidden Variable” Problem

- Accidentally hiding a class variable or instance variable is a common bug in Java.
- Hidden variables aren’t illegal, so the compiler won’t point out the problem.
- A parameter can also hide an instance variable or class variable.
- A variable declared in a block can’t hide a variable in an enclosing block, however.
10.3 Designing Instance Methods

- Deciding which instance methods to put in a class is more important—and more difficult—than deciding which instance variables to include.
- The public methods are especially important because they’re the only ones that methods in other classes will be able to call.
- The public methods form the “interface” to the class, or the “contract” between the class and other classes that rely on it.
Completeness and Convenience

- The public methods provided by a class should have two properties:
  - **Complete.** Any reasonable operation on instances of the class should be supported, either by a single method call or by a combination of method calls.
  - **Convenient.** Common operations should be easy to perform, usually by a single method call.
Completeness and Convenience

- A version of the `Fraction` class with no instance methods:

  ```java
  public class Fraction {
      private int numerator;
      private int denominator;

      public Fraction(int num, int denom) {
          ...
      }
  }
  ```

- This version of `Fraction` is clearly not complete: `Fraction` objects can be created, but no operations can be performed on them.
Completeness and Convenience

Adding two methods gives Fraction a complete set of methods:

```java
public int getNumerator() {
    return numerator;
}

public int getDenominator() {
    return denominator;
}
```
Completeness and Convenience

- The Fraction class still isn’t convenient to use, however.

- A statement that computes the product of f1 and f2:

```java
f3 = new Fraction(
    f1.getNumerator() * f2.getNumerator(),
    f1.getDenominator() * f2.getDenominator());
```
Categories of Methods

- Most instance methods fall into one of several categories:
  - Manager methods—Methods that initialize objects and convert them to other types.
  - Implementor methods—Methods that implement the important capabilities of the class.
  - Access methods—Methods that allow access to private instance variables.
  - Helper methods—Methods that are called only by other methods in the same class.

- A class that lacks an adequate selection of methods in each category probably won’t be convenient.
Manager Methods

- Constructors fall into the category of manager methods.
- Most classes will need at least one constructor.
- Having additional constructors makes a class easier to use.
- Manager methods also include methods that convert an object to some other type, such as `toString`.
Manager Methods

- A `toString` method for the `Account` class:
  ```java
  public String toString() {
      long roundedBalance =
          Math.round(balance * 100);
      long dollars = roundedBalance / 100;
      long cents = roundedBalance % 100;
      return "$" + dollars +
          (cents <= 9 ? ".0" : ".") + cents;
  }
  ```
  `toString` is called automatically when an object is printed or concatenated with a string:
  ```java
  System.out.println(acct);
  System.out.println("Balance: " + acct);
  ```
Manager Methods

- `toString` may not be the only conversion method that a class needs to provide.
- The `Fraction` class should provide methods that convert fractions to float and double values:

  ```java
  public float toFloat() {
      return (float) numerator / denominator;
  }
  
  public double toDouble() {
      return (double) numerator / denominator;
  }
  ```
Implementor Methods

- Implementor methods represent useful operations that can be performed on instances of the class.
- In many cases, implementor methods change the state of an object or create a new object.
- The `Account` class has three implementor methods: deposit, withdraw, and close.
Implementor Methods

- For convenience, the `Fraction` class needs at least four implementor methods: `add`, `subtract`, `multiply`, and `divide`.

- Other possible implementor methods:
  - Compare fractions
  - Find the reciprocal of a fraction
  - Compute the absolute value of a fraction
  - Determine the larger of two fractions

- The more implementor methods there are, the more convenient the class will be.
Access Methods

- Types of access methods:
  - **Accessors** (or **getters**) — methods that *return* the value of a private variable.
  - **Mutators** (or **setters**) — methods that *change* the value of a private variable.

  - By convention, names of getters start with the word *get*.
  - Names of setters begin with *set*. 
Access Methods

- Providing a getter and/or a setter for a variable is usually better than making the variable public.
- The advantages of information hiding normally outweigh the inconvenience of having to call a method to access the variable.
- There’s usually no need to provide an access method for every instance variable.
Helper Methods

- Section 7.7 showed how to write helper methods for `main`.
- Instance methods can also benefit from helpers.
- A helper for an instance method can either be an instance method or a class method.
- The constructor for the `Fraction` class could benefit from a helper method.
The Fraction constructor as given in Section 4.6:

```java
public Fraction(int num, int denom) {
    // Compute GCD of num and denom
    int m = num, n = denom;
    while (n != 0) {
        int r = m % n;
        m = n;
        n = r;
    }

    // Divide num and denom by GCD; store results
    // in instance variables
    if (m != 0) {
        numerator = num / m;
        denominator = denom / m;
    }
}
```
Helper Methods

// Adjust fraction so that denominator is never negative
if (denominator < 0) {
    numerator = -numerator;
    denominator = -denominator;
}

To simplify the constructor, the statements that reduce the fraction to lowest terms can be moved to a reduce helper method.

The call

f.reduce();

will reduce f to lowest terms.
Helper Methods

- **Code for the reduce method:**

```java
private void reduce() {
    // Compute GCD of numerator and denominator
    int m = numerator, n = denominator;
    while (n != 0) {
        int r = m % n;
        m = n;
        n = r;
    }

    // Divide numerator and denominator by GCD
    if (m != 0) {
        numerator /= m;
        denominator /= m;
    }
}
```
// Adjust fraction so that denominator is never negative
if (denominator < 0) {
    numerator = -numerator;
    denominator = -denominator;
}

- The Fraction constructor can now call reduce:
  public Fraction(int num, int denom) {
      numerator = num;
      denominator = denom;
      reduce();
  }
Helper Methods

- The call of `reduce` doesn’t specify which object is being reduced:
  ```java
  reduce();
  ```

- When a constructor calls an instance method without specifying an object, the compiler assumes that the method is being called by the same object that the constructor is currently working with.

- The same property holds when an instance method calls another instance method in the same class.
Using Class Methods as Helpers

- Instance methods are allowed to call class methods in the same class, so a class method can serve as a helper for an instance method.
- The `reduce` method uses Euclid’s algorithm to find the GCD of the fraction’s numerator and denominator.
- To simplify `reduce` (and create a potentially useful helper method), the code for Euclid’s algorithm can be moved to a new method named `gcd`. 
Using Class Methods as Helpers

- gcd won’t need access to the numerator and denominator variables, so it can be a class method:

  ```java
  private static int gcd(int m, int n) {
      while (n != 0) {
          int r = m % n;
          m = n;
          n = r;
      }
      return m;
  }
  ```
The new version of reduce:

```java
class Rational {
    private int numerator, denominator;

    // Constructor
    public Rational(int numerator, int denominator) {
        this(numerator, denominator, 1);
    }

    // Constructor
    public Rational(int numerator, int denominator, int scale) {
        this(numerator * scale, denominator * scale);
        reduce();
    }

    // Methods
    public void simplify() {
        reduce();
    }

    // Private methods
    private void reduce() {
        // Compute GCD of numerator and denominator
        int g = gcd(numerator, denominator);

        // Divide numerator and denominator by GCD
        if (g != 0) {
            numerator /= g;
            denominator /= g;
        }

        // Adjust fraction so that denominator is never negative
        if (denominator < 0) {
            numerator = -numerator;
            denominator = -denominator;
        }
    }
}
```
Method Overloading

- Giving the same name to two or more methods in a class is known as **overloading**; the methods themselves are said to be **overloaded**.

- Both instance methods and class methods can be overloaded.

- Overloading is normally done when two or more methods perform essentially the same operation.

- The advantage of overloading is that there are fewer method names to remember.
Method Overloading

- The String class contains four methods named `indexOf`:
  - `int indexOf(int ch)`
  - `int indexOf(int ch, int fromIndex)`
  - `int indexOf(String str)`
  - `int indexOf(String str, int fromIndex)`

- All four search for a key (a single character or a string) and return the index at which it was found.
Method Overloading

- Every method has a *signature*, which consists of the name of the method and the number and types of its parameters.
- Methods within a class must have different signatures.
- Two methods with the same name must have different numbers of parameters or a difference in the types of corresponding parameters.
Method Overloading

- The `print` and `println` methods are heavily overloaded. Versions of `print`:
  - `void print(boolean b)`
  - `void print(char c)`
  - `void print(int i)`
  - `void print(long l)`
  - `void print(float f)`
  - `void print(double d)`
  - `void print(String s)`
Method Overloading

- In many cases, a call of an overloaded method could apply to any of several methods.
- A call of `print` with an `int` argument could (in theory) be a call of `print(int), print(long), print(float), or print(double)`.
- All these versions of `print` are said to be applicable.
- The Java compiler chooses the version that is most specific.
- In this example, `print(int)` is the most specific, because an `int` can be used as a `long, float, or double`, but not the other way around.
Method Overloading

- The Java compiler can’t always find a most specific method.
- Suppose that a class contains the following two methods:
  
  ```java
  void f(int i, long j)
  void f(long i, int j)
  ```
- Calling `f` with two `int` values as arguments is illegal, because neither version of `f` is more specific than the other.
An instance method must be called by an object.

For example, the `getBalance` method must be called by an `Account` object:
```
double balance = acct.getBalance();
```

Although an instance method never knows the name of the object that called it, the keyword `this` can be used to refer to the object.
Using **this** to Access Hidden Variables

- An instance method may sometimes have a local variable (or a parameter) with the same name as a variable that belongs to the enclosing class.
- **this** can be used to access the variable in the class, which would otherwise be hidden.
Using **this** to Access Hidden Variables

- **Consider the** `Account` **constructor:**

  ```java
  public Account(double initialBalance) {
      balance = initialBalance;
  }
  ```

- **Using the name** `balance` **for the parameter would be legal, but the assignment would have no effect:**

  ```java
  public Account(double balance) {
      balance = balance;
  }
  ```
Using **this** to Access Hidden Variables

- **A corrected version using** **this**:  
  ```java
  public Account(double balance) {
    this.balance = balance;
  }
  ```

- **Some programmers always use** **this** **when referring to**  
  **an instance variable, even when it’s not needed:**  
  ```java
  public double getBalance() {
    return this.balance;
  }
  ```
Using **this** to Call Instance Methods

- Normally, a call of an instance method is preceded by an object:
  
  ```
  object. f();
  ```

- If one instance method calls another in the same class, it’s not necessary to specify the calling object.

- If `g` is another method in the same class as `f`, the body of `f` can call `g` as follows:

  ```
  g();
  ```
Using **this** to Call Instance Methods

- **An example from the** Account class:

  ```java
  public void close() {
    withdraw(balance);
  }
  ```

- **The calling object can be made explicit by using the word** this:

  ```java
  public void close() {
    this.withdraw(balance);
  }
  ```
Using **this** as an Argument

- Sometimes an instance method will need to pass the object that called it to another method.
- **Using this** as the argument to the call solves the problem of naming the object.
- A method that uses **this** to compute the reciprocal of a fraction:

```java
public Fraction reciprocal() {
    Fraction one = new Fraction(1, 1);
    return one.divide(this);
}
```
Using **this** as an Argument

- A better version of **reciprocal**:  
  ```java
  public Fraction reciprocal() {
    return new Fraction(denominator, numerator);
  }
  ```
Using **this** as a Return Value

- **this is also useful in return statements**, when a method wants to return the object that called it.
- **Suppose that the** Fraction class **had a max method** that returns the larger of two fractions:

  ```java
  f3 = f1.max(f2);
  ```
Using **this** as a Return Value

- The `max` method will compare the two fractions and return the larger one.
- If the calling object is the larger of the two fractions, `max` will return the value of **this**:

```java
public Fraction max(Fraction f) {
    if (numerator * f.denominator >
        denominator * f.numerator)
        return this;
    else
        return f;
}
```
Using **this** as a Return Value

- Sometimes a method will return **this** so that method calls to be chained.
- The **deposit** and **withdraw** methods of the **Account** class could be modified to return **this**:

```java
public Account deposit(double amount) {
    balance += amount;
    return this;
}

guard Account withdraw(double amount) {
    balance -= amount;
    return this;
}
```
Using **this** as a Return Value

- A single statement can now perform multiple deposits and withdrawals on the same `Account` object:
  ```java
  acct.deposit(1000.00).withdraw(500.00).deposit(2500.00);
  ```
10.5 Writing Constructors

- If a class has no constructors, Java automatically supplies a default constructor (a constructor that does nothing).
- Most of the time, it’s a good idea to write one or more constructors for a class.
- If a class has $n$ instance variables, it will often need a constructor with $n$ parameters, one for each instance variable.
Example: The `Complex` Class

- A class whose instances represent complex numbers:
  ```java
class Complex {
    private double realPart;
    private double imaginaryPart;
    ...
}
```
- A constructor with one parameter for each instance variable:
  ```java
public Complex(double real, double imaginary) {
    realPart = real;
    imaginaryPart = imaginary;
}
```
Example: The **Complex** Class

- Although this constructor is the only one the `Complex` class needs, it’s often a good idea to provide additional constructors for convenience.

- A second constructor that allows a `Complex` object to be created from a single real number:

```java
public Complex(double real) {
    realPart = real;
    imaginaryPart = 0.0;
}
```
Using **this** in Constructors

- It’s not unusual for a Java class to have a large number of constructors that are very similar.
- To make it easier to write such constructors, Java allows one constructor to call another by using the word **this**:
  ```java
  public Complex(double real) {
      this(real, 0.0);
  }
  ```
- Any other statements in the constructor must come after the call of **this**.
“No-Arg” Constructors

- A **“no-arg” constructor** is a constructor that has no arguments.
- It’s usually a good idea to include a no-arg constructor that assigns default values to the instance variables of a newly created object.
- A no-arg constructor for the `Complex` class:

```java
public Complex() {
    realPart = 0.0;
    imaginaryPart = 0.0;
}
```
“No-Arg” Constructors

- Another way to write the no-arg constructor:
  ```java
  public Complex() {
    this(0.0, 0.0);
  }
  ```

- Writing a no-arg constructor may seem pointless, because Java should automatically create a default constructor.

- Problems with this reasoning:
  - It’s often better to initialize variables explicitly rather than let them assume default values assigned by Java.
  - Java won’t create a default constructor for a class if the class contains any other constructors.
Constructors Versus Instance Variable Initialization

- It may be possible to simplify the no-arg constructor for a class by specifying initial values in the declarations of instance variables.

- Consider the `Fraction` class:

  ```java
  public class Fraction {
    private int numerator;
    private int denominator;

    public Fraction() {
      numerator = 0;
      denominator = 1;
    }
  }
  ```
The assignments can be removed from the constructor if numerator and denominator are initialized at the time they’re declared:

```java
public class Fraction {
    private int numerator = 0;
    private int denominator = 1;

    public Fraction() {}
    ...
}
```
Constructors Versus Instance Variable Initialization

- When an instance of a class is created, the following steps are done (in order):
  1. Default values are assigned to all instance variables. Numeric variables—including `char` variables—are set to zero. `boolean` variables are set to `false`. Instance variables of a reference type are set to `null`.
  2. For each instance variable whose declaration contains an initializer, the initializer is evaluated and assigned to the variable.
  3. The constructor is executed.
Constructors Versus Instance Variable Initialization

- **A modified version of the** `Fraction` **class:**

  ```java
  public class Fraction {
    private int numerator = 1;
    private int denominator = 1;
    
    public Fraction() {
      numerator = 0;
      denominator = 1;
    }
  }
  ...
  }
  ```

- **Steps performed when a** `Fraction` **object is created:**
  1. `numerator` **and** `denominator` **are set to 0.**
  2. `numerator` **and** `denominator` **are set to 1.**
  3. `numerator` **is set to 0;** `denominator` **is set to 1.**
10.6 Example: The *Fraction* Class

- The *Fraction* class can now be more fully developed.
- **Constructors needed:**
  - A constructor that initializes both the numerator and denominator to specified values.
  - A constructor that allows the denominator to be given a default value.
  - A constructor that allows both the numerator and denominator to be given default values.
Design of the Fraction Class

- Manager methods:
  - `toString`, `toDouble`, `toFloat`

- Implementor methods:
  - `add`, `subtract`, `multiply`, `divide`

- Access methods:
  - `getNumerator`, `getDenominator`

- Helper methods:
  - `reduce`, `gcd`
Fraction.java

// Class name: Fraction
// Author: K. N. King
// Written: 1998-05-28
// Modified: 1999-07-12

// Represents a mathematical fraction, consisting of a numerator and a denominator, both of which are integers.

public class Fraction {
    // Instance variables
    private int numerator;    // Numerator of fraction
    private int denominator;  // Denominator of fraction
public Fraction(int num, int denom) {
    numerator = num;
    denominator = denom;
    reduce();
}

public Fraction(int num) {
    this(num, 1);
}
public Fraction() {
    this(0, 1);
}

public Fraction add(Fraction f) {
    int num = numerator * f.denominator +
              f.numerator * denominator;
    int denom = denominator * f.denominator;
    return new Fraction(num, denom);
}
public Fraction divide(Fraction f) {
    int num = numerator * f.denominator;
    int denom = denominator * f.numerator;
    return new Fraction(num, denom);
}

public int getDenominator() {
    return denominator;
}
public int getNumerator() {
    return numerator;
}

public Fraction multiply(Fraction f) {
    int num = numerator * f.numerator;
    int denom = denominator * f.denominator;
    return new Fraction(num, denom);
}
public Fraction subtract(Fraction f) {
    int num = numerator * f.denominator -
              f.numerator * denominator;
    int denom = denominator * f.denominator;
    return new Fraction(num, denom);
}

public double toDouble() {
    return (double) numerator / denominator;
}
/*****

// NAME: toFloat
// BEHAVIOR: Converts this fraction into a float value
// PARAMETERS: None
// RETURNS: A float value obtained by dividing the fraction's numerator by its denominator

public float toFloat() {
    return (float) numerator / denominator;
}
public String toString() {
    if (denominator == 1)
        return numerator + "";
    else
        return numerator + "/" + denominator;
}
private void reduce() {
    // Compute GCD of numerator and denominator
    int g = gcd(numerator, denominator);

    // Divide numerator and denominator by GCD
    if (g != 0) {
        numerator /= g;
        denominator /= g;
    }

    // Adjust fraction so that denominator is never negative
    if (denominator < 0) {
        numerator = -numerator;
        denominator = -denominator;
    }
}

90
private static int gcd(int m, int n) {
    while (n != 0) {
        int r = m % n;
        m = n;
        n = r;
    }
    return m;
}
Order of Variables and Methods

- There’s no universal agreement about the order in which variables and methods should appear in a class declaration.

- One possible arrangement:
  1. Instance variables
  2. Class variables
  3. Constructors
  4. Public instance methods
  5. Public class methods
  6. Private instance methods
  7. Private class methods
Order of Variables and Methods

- Putting the methods in each group into alphabetical order makes it easier to locate a particular method.
- Another way to arrange methods within a group is by role, with related methods kept together.
- For example, `add`, `subtract`, `multiply`, and `divide` would be kept together, because they perform similar operations.
10.7 Adding Class Variables and Methods

- Instantiable classes often contain class variables and methods in addition to instance variables and methods.
Class Variables in Instantiable Classes

- Sometimes a class will need to keep track of information about the instances that have been created so far.
- For example, the `Account` class might need to remember how many accounts have been created and the total amount of money stored in those accounts.
- These items of data aren’t related to any particular account, but to all accounts in existence, so they would logically be stored as class variables.
The Account class with the addition of class variables named totalDeposits and accountsOpen:

```java
public class Account {
    // Instance variables
    private double balance;

    // Class variables
    private static double totalDeposits = 0.0;
    private static int accountsOpen = 0;

    ...
}
```
Class Variables in Instantiable Classes

- If a class variable is public, it can be accessed by writing the class name, a dot, and the variable name.

- **If accountsOpen** were public, it could be accessed by writing `Account.accountsOpen`:
  ```java
  System.out.println("Number of accounts open: " + Account.accountsOpen);
  ```

- **Class variables should usually be private:**
  - Public class variables are vulnerable to being changed outside the class.
  - Changing the name or type of a public class variable may affect other classes.
Class Variables in Instantiable Classes

- Constructors and instance methods have full access to all class variables in the same class.

- A version of the Account class in which the constructors and instance methods keep totalDeposits and accountsOpen up-to-date:

  ```java
  public class Account {
    // Instance variables
    private double balance;

    // Class variables
    private static double totalDeposits = 0.0;
    private static int accountsOpen = 0;
  }
  ```
Class Variables in Instantiable Classes

// Constructors
public Account(double initialBalance) {
    balance = initialBalance;
    totalDeposits += initialBalance;
    accountsOpen++;
}

public Account() {
    balance = 0.0;
    accountsOpen++;
}

// Instance methods
public void close() {
    totalDeposits -= balance;
    balance = 0.0;
    accountsOpen--;
}
Class Variables in Instantiable Classes

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

public double getBalance() {
    return balance;
}

public void withdraw(double amount) {
    balance -= amount;
    totalDeposits -= amount;
}
```
Class Methods in Instantiable Classes

- Not every class method in an instantiable class is a helper method.
- Class methods also provide the ability to access and/or change class variables.
The new version of the Account class will need class methods that return the values of the totalDeposits and accountsOpen variables:

```java
public static double getTotalDeposits() {
    return totalDeposits;
}

public static int getAccountsOpen() {
    return accountsOpen;
}
```

Examples of calling these methods:

```java
double deposits = Account.getTotalDeposits();
int numAccounts = Account.getAccountsOpen();
```
# Restrictions on Class Methods

<table>
<thead>
<tr>
<th><strong>Instance Methods</strong></th>
<th><strong>Class Methods</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Are allowed to use <code>this</code> to refer to the calling object.</td>
<td>Are not allowed to use <code>this</code>.</td>
</tr>
<tr>
<td>Have access to all instance variables and class variables declared in the enclosing class.</td>
<td>Have access to class variables in the enclosing class, but not instance variables.</td>
</tr>
<tr>
<td>Can call any instance method or class method declared in the class.</td>
<td>Can call class methods, but not instance methods.</td>
</tr>
</tbody>
</table>
Restrictions on Class Methods

- A class method isn’t called by an object, so it has no access to instance variables and can’t call instance methods.

- However, if a class method is supplied with an instance of the class (through a parameter or class variable), the method does have access to that object’s instance variables, and it can use the object to call instance methods.
10.8 Reusing Code

- One of the biggest advantages of object-oriented programming is the ease with which code can be reused in other programs.
- When properly written, class methods and entire classes can be reused.
- To make a method or class reusable, it should be written in a general fashion, not tied too closely to the program under development.
- A class can be given extra methods that aren’t needed in the current program.
Reusing Class Methods

- For a class method to be reusable, it needs to be placed in an appropriate class and declared to be public.

- For example, the gcd method could be put in a class named MathAlgorithms:

```java
public class MathAlgorithms {
    public static int gcd(int m, int n) {
        while (n != 0) {
            int r = m % n;
            m = n;
            n = r;
        }
        return m;
    }
}
```
Reusing Class Methods

- It's a good idea for methods that read and validate user input to be reusable.
- For example, the `readInt` method of Section 8.1 could be put in a class named `ValidIO`:

  ```java
  public class ValidIO {
      public static int readInt(String prompt) {
          while (true) {
              SimpleIO.prompt(prompt);
              String userInput = SimpleIO.readLine();
              try {
                  return Integer.parseInt(userInput);
              } catch (NumberFormatException e) {
                  System.out.println("Not an integer; try again");
              }
          }
      }
  }
  ```
Reusing Classes

- Packages are an important tool for making classes reusable.
- A package is a collection of classes.
- Every Java class belongs to a package.
- If the programmer doesn’t specify which package a class belongs to, the Java compiler adds it to a default package that has no name.
Reusing Classes

- A visual representation of the `jpb`, `java.awt`, `java.lang`, and default packages:

<table>
<thead>
<tr>
<th>default package</th>
<th>jpb</th>
<th>java.awt</th>
<th>java.lang</th>
</tr>
</thead>
<tbody>
<tr>
<td>All the classes</td>
<td>Convert</td>
<td>Color</td>
<td>Character</td>
</tr>
<tr>
<td>we've written</td>
<td>DrawableFrame</td>
<td>Font</td>
<td>Integer</td>
</tr>
<tr>
<td>so far</td>
<td>SimpleIO</td>
<td>FontMetrics</td>
<td>Math</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Graphics</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>System</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reusing Classes

- Classes that belong to the same package have additional access privileges.
- If the access modifier is omitted in the declaration of an instance variable or class variable, that variable is available to all classes in the same package.
- The same is true of instance methods and class methods.
- Classes in other packages, however, have no access to such a variable or method.
Reusing Classes

- Each file in a package may contain multiple classes, but only one may be declared `public` (the one whose name matches the file name).
- Classes that aren’t declared `public` aren’t accessible to classes in other packages.
- A strategy for making a class easily reusable in other programs:
  - Make sure that the class is declared `public`.
  - Put the class into a package with related classes.
Writing Packages

- Steps required to create a package:
  1. Add a package declaration to each file that contains classes belonging to the package.
  2. Put the files in a directory with the same name as the package.
  3. Make sure that the Java compiler and interpreter can find the directory.

- Form of a `package declaration`:
  ```
  package package-name ;
  ```

- A package declaration must come before any import lines and before the classes themselves.
Example: adding `ValidIO` to the `jpb` package:

```java
package jpb;

public class ValidIO {
    public static int readInt(String prompt) {
        while (true) {
            try {
                SimpleIO.prompt(prompt);
                String userInput = SimpleIO.readLine();
                return Integer.parseInt(userInput);
            } catch (NumberFormatException e) {
                System.out.println("Not an integer; try again");
            }
        }
    }
}
```
Writing Packages

- The files that belong to a package must be put in a directory with the same name as the package.
- The `ValidIO.java` file would go in the `jpb` directory, along with the other classes that belong to the `jpb` package.
- Package names may contain dots, which indicate directory structure.
- For example, the files for the `java.awt` package would be kept in a subdirectory named `awt`, which is located inside a directory named `java`.
On the Windows platform, the Java compiler and interpreter locate package directories via the `CLASSPATH` environment variable.

A definition of `CLASSPATH` that causes a search for package names in the current directory and in `C:\MyPkgs`:

```
SET CLASSPATH=.;C:\MyPkgs
```

`CLASSPATH` must list the name of the directory in which the package directory is located, not the name of the package directory itself.
10.9 Case Study: Playing Blackjack

- The Blackjack program will play the card game known as Blackjack.
- Rules for the game:
  - Blackjack is played with standard playing cards. The value of each numbered card is the card’s number. The value of the jack, queen, and king is 10. The value of the ace can be either 1 or 11.
  - The goal of the game is to accumulate cards whose total values come as close to 21 as possible, without exceeding 21.
Rules of Blackjack

- **Additional rules:**
  - At the beginning of each hand, the dealer (the program) and the player (the user) are each dealt two cards. If the player’s cards total to 21, the hand is over and the player wins (unless the dealer also has 21, in which case the hand is a tie).
  - Otherwise, the player may now draw additional cards in an attempt to get closer to 21. If the player exceeds 21, the hand is over and the player loses.
  - Once the player “stands” (stops drawing cards), the dealer begins drawing cards until reaching a total of 17 or more. If the dealer exceeds 21, the player wins. If the player’s final count exceeds the dealer’s final count, the player wins. If the dealer and the player have identical counts, the hand is a tie. Otherwise, the dealer wins.
Program Behavior

- The program will pick two cards randomly for both the dealer and the player.
- It will then display the player’s cards and ask the user whether to stand or “hit” (request a card).
- The user will enter \( S \) (for stand) or \( H \) (for hit).
- Any input other than \( h \) or \( H \) is assumed to mean “stand.”
- The process repeats until the player’s count exceeds 21 or the player decides to stand.
Program Behavior

- If the player’s count did not go over 21, the program displays the dealer’s cards and draws additional cards until reaching 17 or exceeding 21.
- At the end of each hand, the program will display the total number of wins for both the dealer and the player.
- The program will then ask the user whether or not to play again.
- Any input other than \( y \) or \( Y \) will cause program termination.
User Interface

A sample session with the Blackjack program:

Your cards: 6S 5H
(S)tand or (H)it? h
You drew: 6H
(S)tand or (H)it? s
Dealer's cards: QC QS
Dealer wins
Dealer: 1 Player: 0
Play again (Y/N)? y
User Interface

Your cards: 3C 2C
(S)tand or (H)it? h
You drew: JC
(S)tand or (H)it? s
Dealer's cards: 2D AS
Dealer drew: 2H
Dealer drew: AH
Dealer drew: KC
Dealer drew: 3C
Dealer wins
Dealer: 2 Player: 0
Play again (Y/N)? y
User Interface

Your cards: JS AH
Dealer's cards: 5H TC
Player wins!
Dealer: 2 Player: 1
Play again (Y/N)? y

Your cards: 3H TS
(S) tand or (H) it? h
You drew: QH
Dealer wins
Dealer: 3 Player: 1
Play again (Y/N)? y
User Interface

Your cards: 7C TD
(S)tand or (H)it? s
Dealer's cards: 4D 8C
Dealer drew: 2S
Dealer drew: 2D
Dealer drew: TH
Player wins!
Dealer: 3 Player: 2
Play again (Y/N)? y
User Interface

Your cards: JS 7C
(S)tant or (H)it? s
Dealer's cards: TS 5D
Dealer drew: 2S
Tie
Dealer: 3 Player: 2
Play again (Y/N)? n
Design of the **Card** Class

- The program will need two classes:
  - **Blackjack**. Contains the main method and its helpers.
  - **Card**. Each instance represents a single playing card.

- **The Card class will need rank and suit instance variables** to store the rank and suit of a card.

- To represent ranks and suits, integer codes can be used (0 to 12 for ranks; 0 to 3 for suits).
Design of the **Card** Class

- The **Card** class will need a constructor to initialize the **rank** and **suit** variables.

- **Methods needed by the Card class:**
  - `getRank`
  - `getSuit`
  - `toString`
  - A class method that generates a random card
// Class name: Card
// Author: K. N. King
// Written: 1998-05-26
// Modified: 1999-07-11

// Represents a playing card with a rank and a suit.

public class Card {
    // Instance variables
    private int rank;
    private int suit;

    // Names for ranks
    public static final int TWO = 0;
    public static final int THREE = 1;
    public static final int FOUR = 2;
    public static final int FIVE = 3;
    public static final int SIX = 4;
public static final int SEVEN = 5;
public static final int EIGHT = 6;
public static final int NINE = 7;
public static final int TEN = 8;
public static final int JACK = 9;
public static final int QUEEN = 10;
public static final int KING = 11;
public static final int ACE = 12;

// Names for suits
public static final int CLUBS = 0;
public static final int DIAMONDS = 1;
public static final int HEARTS = 2;
public static final int SPADES = 3;

// Constants for use within the class
private static final String RANKS = "23456789TJQKA";
private static final String SUITS = "CDHS";
public Card(int rank, int suit) {
    this.rank = rank;
    this.suit = suit;
}

public int getRank() {
    return rank;
}
public int getSuit() {
    return suit;
}

public String toString() {
    return RANKS.charAt(rank) + "" + SUITS.charAt(suit);
}
public static Card pickRandom() {
    return new Card((int) (Math.random() * 13),
                    (int) (Math.random() * 4));
}
Discussion of the **Card** Class

- The **Card** class includes constants that provide names for all the possible ranks and suits.
- These constants make it easier to work with ranks and suits, as well as improving readability:

  ```java
  Card c = new Card(Card.ACE, Card.SPADES);
  ```

  is easier to read than

  ```java
  Card c = new Card(12, 3);
  ```

- The **Blackjack** program doesn’t need names for suits. By providing these names, the **Card** class becomes easier to reuse in other programs.
Discussion of the **Card** Class

- The `pickRandom` method doesn’t keep track of which cards have been generated in the past.
- It behaves like a dealer with an infinite number of decks that have been shuffled together.
- The **Card** constructor will accept any integer as a rank or suit value, which is undesirable.
- The constructor should probably throw an exception if it is given an illegal argument.
Design of the **Blackjack** Class

- Since the program can play multiple hands, the game-playing code will need to be inside a loop.

- Design of loop body:
  
  *Choose two cards for both player and dealer;*
  
  *Display player’s cards;*
  
  *Compute initial counts for player and dealer;*
  
  ```java
  if (player’s count is 21) {
      if (dealer’s count is not 21)
          Dealer loses;
  } else {
      Ask player to draw additional cards and determine new value of player’s hand;
  }
  ```
Design of the **Blackjack** Class

```java
if (player’s count exceeds 21)
    Player loses;
else {
    Show dealer’s cards;
    Draw additional cards for dealer and determine new value of dealer’s hand;
    if (dealer’s count exceeds 21)
        Dealer loses;
}

Compare player’s count with dealer’s count to determine the winner; display the outcome and update the win counts;
Display the win counts;
See if user wants to play again; exit from loop if answer is no;
```
Design of the **Blackjack** Class

- One of the trickier issues is how to keep track of whether the dealer or the player has lost during an early stage of the game.

- *boolean* variables could be used, but there’s a better way.

- To indicate that the dealer has lost, the program can simply set the dealer’s count to 0.

- Similarly, the player’s count can be set to 0 to indicate that the player has lost.
Design of the **Blackjack** Class

- To simplify the `main` method, two helper methods are needed:
  - `getDealerCards`: Contains a loop that draws cards until the dealer’s count reaches 17.
  - `getPlayerCards`: Similar, but repeatedly asks the user whether or not to draw a card. Returns when the player’s count exceeds 21 or the user decides to stand.

- The hardest part of writing these methods is getting them to handle aces properly.
Design of the **Blackjack** Class

- One technique is to assume that the value of each ace is 11. Whenever a count exceeds 21, it is reduced by 10 if an ace is present.

- Since a hand may contain more than one ace, a variable will be needed to keep track of the number of aces that are still being valued at 11.

- The *Blackjack* class will need one more helper method: `getCount`, whose job is to determine the Blackjack value of a card.
Blackjack.java

// Program name: Blackjack
// Author: K. N. King
// Written: 1998-05-26
// Modified: 1999-07-11

// Plays the game of Blackjack. At the beginning of each hand, the dealer (the program) and the player (the user) are each dealt two cards. If the player's cards total to 21, the hand is over and the player wins (unless the dealer also has 21, in which case the hand is a tie). The player may now draw additional cards in an attempt to get close to 21. If the player exceeds 21, the hand is over and the player loses. Otherwise, the dealer begins drawing cards until reaching a total of 17 or more. If the dealer exceeds 21, the player wins. If the player's final count exceeds the dealer's final count, the player wins. If the dealer and the player have identical counts, the hand is a tie. Otherwise, the dealer wins.
At the end of each hand, the program will display the total number of wins for both the dealer and the player. The player will then be asked whether or not to play another hand.

```java
import jpb.*;

public class Blackjack {
    public static void main(String[] args) {
        int playerWins = 0;
        int dealerWins = 0;

        while (true) {
            // Choose two cards for both player and dealer
            Card playerCard1 = Card.pickRandom();
            Card playerCard2 = Card.pickRandom();
            Card dealerCard1 = Card.pickRandom();
            Card dealerCard2 = Card.pickRandom();
```
// Display player's cards
System.out.println("Your cards: " + playerCard1 + " " + playerCard2);

// Compute initial counts for player and dealer
int playerCount = getCount(playerCard1) + getCount(playerCard2);
int dealerCount = getCount(dealerCard1) + getCount(dealerCard2);

// Check whether player's count is 21. If so, dealer
// must have 21 or lose automatically.
if (playerCount == 21) {
    if (dealerCount != 21)
        dealerCount = 0;
} else {
    // Player's count was not 21. Ask player to draw
    // additional cards and determine new value of
    // player's hand.
    playerCount = getPlayerCards(playerCard1, playerCard2);
```java
// Player loses if new count exceeds 21
if (playerCount > 21)
    playerCount = 0;
else {
    // Player's count does not exceed 21. Show dealer's cards.
    System.out.println("Dealer's cards: " +
                       dealerCard1 + " " + dealerCard2);

    // Draw additional cards for dealer and determine new value of dealer's hand
    dealerCount = getDealerCards(dealerCard1, dealerCard2);

    // Dealer loses if new count exceeds 21
    if (dealerCount > 21)
        dealerCount = 0;
}
```
// Compare player's count with dealer's count to
determine the winner; display the outcome and
update the win counts
if (playerCount > dealerCount) {
    System.out.println("You win!");
    playerWins++;
} else if (playerCount < dealerCount) {
    System.out.println("Dealer wins");
    dealerWins++;
} else
    System.out.println("Tie");

// Display the win counts
System.out.println("Dealer: " + dealerWins +
                " Player: " + playerWins);
// See if user wants to play again; exit from loop if
// answer is no
SimpleIO.prompt("Play again (Y/N)? ");
String userInput = SimpleIO.readLine();
if (!userInput.equalsIgnoreCase("Y"))
    break;
System.out.println();


// NAME:       getDealerCards
// BEHAVIOR:   Adds cards to the dealer's hand until the
//             value reaches 17 or more
// PARAMETERS: card1 - One of dealer's original two cards
//             card2 - The other original card
// RETURNS:    Value of the dealer's hand, including
//             original cards and new cards


}
private static int getDealerCards(Card card1, Card card2) {
    int dealerCount = getCount(card1) + getCount(card2);
    int aceCount = 0;

    // Determine number of aces among original pair of cards
    if (card1.getRank() == Card.ACE) 
        aceCount++;
    if (card2.getRank() == Card.ACE) 
        aceCount++;

    while (true) {
        // If the dealer's count exceeds 21 and the hand
        // contains aces still valued at 11, then reduce the
        // number of aces by 1 and reduce the count by 10
        if (aceCount > 0 && dealerCount > 21) {
            aceCount--;
            dealerCount -= 10;
        }
    }
}
// Return if dealer's count is at least 17
if (dealerCount >= 17)
    return dealerCount;

// Pick a new card and update the dealer's count
Card newCard = Card.pickRandom();
System.out.println("Dealer drew: " + newCard);
dealerCount += getCount(newCard);

// Check whether the new card is an ace
if (newCard.getRank() == Card.ACE)
    aceCount++;
}
private static int getPlayerCards(Card card1, Card card2) {  
    int playerCount = getCount(card1) + getCount(card2);  
    int aceCount = 0;  

    // Determine number of aces among original pair of cards  
    if (card1.getRank() == Card.ACE)  
        aceCount++;  
    if (card2.getRank() == Card.ACE)  
        aceCount++;  

    // Calculate the player's hand value  
    int handValue = playerCount;  
    if (aceCount > 0)  
        handValue -= aceCount * 10;  
    return handValue;  
}
while (true) {
    // If the player's count exceeds 21 and the hand
    // contains aces still valued at 11, then reduce the
    // number of aces by 1 and reduce the count by 10
    if (aceCount > 0 && playerCount > 21) {
        aceCount--;  
        playerCount -= 10;  
    }

    // Return if player's count exceeds 21
    if (playerCount > 21)
        return playerCount;

    // Ask user whether to stand or hit
    SimpleIO.prompt("(S)tand or (H)it? ");
    String userInput = SimpleIO.readLine();
    if (!userInput.equalsIgnoreCase("H"))
        return playerCount;
// Pick a new card and update the player's count
Card newCard = Card.pickRandom();
System.out.println("You drew: " + newCard);
playerCount += getCount(newCard);

// Check whether the new card is an ace
if (newCard.getRank() == Card.ACE)
    aceCount++;
}

/////////////////////////////////////////////////////////////////////////////
// NAME: getCount
// BEHAVIOR: Returns the Blackjack value of a particular card
// PARAMETERS: c - a Card object
// RETURNS: The Blackjack value of the card c. The value of a card is the same as its rank, except that face cards have a value of 10 and aces have a value of 11.
/////////////////////////////////////////////////////////////////////////////
private static int getCount(Card c) {
   switch (c.getRank()) {
      case Card.TWO:   return 2;
      case Card.THREE: return 3;
      case Card.FOUR:  return 4;
      case Card.FIVE:  return 5;
      case Card.SIX:   return 6;
      case Card.SEVEN: return 7;
      case Card.EIGHT: return 8;
      case Card.NINE:  return 9;
      case Card.ACE:   return 11;
      default:         return 10;  // TEN, JACK, QUEEN, KING
   }
}
}
10.10 Debugging

- When a program consists of more than one component, it’s important to test each component before testing the entire program.

- Testing a single component is known as **unit testing**.

- Testing the entire program, after each component has been tested individually, is known as **integration testing**.

- Unit testing helps locate bugs and fix them at an early stage.
Unit Testing in Java

- In Java, unit testing consists of verifying that the constructors and methods in each class work properly.
- One way to test the Fraction class would be to write a small test class (TestFraction) whose main method creates two Fraction objects and performs operations on them.
- TestFraction will ask the user to enter the numerators and denominators of the fractions.
// Tests the behavior of the Fraction class.

import jpb.*;

public class TestFraction {
    public static void main(String[] args) {
        while (true) {
            // Prompt the user to enter the numerator and
denominator of two fractions
            int num = ValidIO.readInt("Enter first numerator: ");
            int denom =
                ValidIO.readInt("Enter first denominator: ");
            Fraction f1 = new Fraction(num, denom);
            System.out.println();

            num = ValidIO.readInt("Enter second numerator: ");
            denom = ValidIO.readInt("Enter second denominator: ");
            Fraction f2 = new Fraction(num, denom);
            System.out.println();
        }
    }
}
// Display the fractions after they've been stored in objects
System.out.println("First fraction: " + f1);
System.out.println("Second fraction: " + f2 + "\n");

// Compute and display the sum, difference, product, and quotient of the two fractions
System.out.println("Sum: " + f1.add(f2));
System.out.println("Difference: " + f1.subtract(f2));
System.out.println("Product: " + f1.multiply(f2));
System.out.println("Quotient: " + f1.divide(f2));

// Ask the user whether or not to test more fractions
SimpleIO.prompt("\nTest more fractions (Y/N)? ");
String userInput = SimpleIO.readLine();
if (!userInput.equalsIgnoreCase("Y"))
    break;
} } }
User Interface

- **User interaction with TestFraction:**
  - Enter first numerator: 3
  - Enter first denominator: 6
  - Enter second numerator: -100
  - Enter second denominator: -50
  - First fraction: 1/2
  - Second fraction: 2
  - Sum: 5/2
  - Difference: -3/2
  - Product: 1
  - Quotient: 1/4

- Test more fractions (Y/N)? n
Test Harnesses

- A class such as `TestFraction` is often called a test harness.
- Test harnesses should be retained for future use.
- If the `Fraction` class is modified in the future, `TestFraction` should be run again to make sure that the changes haven’t broken the `Fraction` class.
- Retesting a component after a change is known as regression testing.
Using the `main` Method as a Test Harness

- In Java, there’s an easy way to create a test harness and make sure that it won’t be lost in the future.
- The secret is to put a `main` method in the class that’s being tested.
- This method will play the role of a test harness by creating instances of the class and performing operations on them.
Using the **main** Method as a Test Harness

- **For example**, TestFraction’s **main method could be moved to the** `Fraction` **class**:

  ```java
  public class Fraction {
    // Rest of Fraction class goes here

    public static void main(String[] args) {
      // Same code as in TestFraction
    }
  }
  ```
Using the **main** Method as a Test Harness

- The Java interpreter must be given the name of the class whose **main** method is to be used.
- **main** methods in other classes are ignored.
- Suppose that the **Card** class has a **main** method that’s used for testing purposes.
- Command to play a game of Blackjack:
  - `java Blackjack`
- Command to test the **Card** class by executing its **main** method:
  - `java Card`