Java Programming: from the Beginning

Chapter 11 Subclasses
11.1 Inheritance

- In the real world, objects aren’t usually one-of-a-kind.
- Both cars and trucks are examples of vehicles. A car is a special type of vehicle; so is a truck.
- In Java, relationships such as the one between cars and vehicles are expressed using a feature known as inheritance.
- Inheritance allows a class to have a “parent” from which it inherits some of its state and some of its behavior.
Advantages of Inheritance

- There would be no point for a car designer to start from scratch when designing a new car.
- By assuming that a car has certain components and certain behaviors, a car designer can focus on what distinguishes the new model from older ones.
- Inheritance gives programmers a similar advantage.
- A new class created from an existing class inherits its variables and methods.
- The programmer can declare additional variables and methods to make the new class unique.
Class Extension

- In Java, inheritance is accomplished by extending an existing class.
- Extending a class is done by putting the word `extends` in a class declaration, followed by the name of the class that’s being extended:
  ```java
  public class Car extends Vehicle {
      ...
  }
  ```
- Car is said to be a `subclass` of Vehicle, and Vehicle is said to be the `superclass` of Car.
Class Extension

- A subclass *inherits* the variables and methods of its superclass, with the exception of constructors, private variables, and private methods.
- Inherited variables and methods behave as though they were declared in the subclass.
- The subclass may define additional variables and methods that were not present in the superclass.
Writing a Subclass

- A superclass can be any previously existing class, including a class in the Java API.
- The `Account` class of Section 3.3 can be extended to create a new `SavingsAccount` class.
- If different savings accounts can have different interest rates, each `SavingsAccount` object will need to store an interest rate.
Writing a Subclass

- A preliminary version of the SavingsAccount class:

```java
public class SavingsAccount extends Account {
    private double interestRate;

    public double getInterestRate() {
        return interestRate;
    }

    public void setInterestRate(double rate) {
        interestRate = rate;
    }
}
```
Writing a Subclass

- An instance of a subclass stores all the instance variables of the superclass, plus all the instance variables defined in the subclass.
- Even the superclass’s private variables, which aren’t inherited, are still stored in instances of the subclass.
Writing a Subclass

- A SavingsAccount object will contain two variables (balance and interestRate):

  SavingsAccount object
  
<table>
<thead>
<tr>
<th>balance</th>
<th>500.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>interestRate</td>
<td>4.25</td>
</tr>
</tbody>
</table>
Writing a Subclass

- Methods that can be applied to `SavingsAccount` objects:
  - `getInterestRate`
  - `setInterestRate`
  - `deposit (inherited)`
  - `withdraw (inherited)`
  - `getBalance (inherited)`
  - `close (inherited)`
If savingsAcct is a SavingsAccount variable, the following statements are all legal:

```java
System.out.println("Rate: "+ savingsAcct.getInterestRate());
savingsAcct.setInterestRate(4.25);
savingsAcct.deposit(500.00);
savingsAcct.withdraw(100.00);
System.out.println("Balance: "+ savingsAcct.getBalance());
savingsAcct.close();
```
Writing a Subclass

- When an instance method is called, Java looks first in the class to which the calling object belongs, then in the class’s superclass, then in the superclass of that class, and so on.

- Consider the following statement:

```java
savingsAcct.deposit(500.00);
```

Java first looks for the `deposit` method in the `SavingsAccount` class, then in the `Account` class.
Writing Subclass Constructors

- A subclass doesn’t inherit constructors from its superclass, so it will need its own constructors.
- The hard part of writing a constructor for a subclass is initializing the variables that belong to the superclass, which are likely to be private.
- The constructor for the SavingsAccount class will need to initialize both the balance and interestRate variables.
Writing Subclass Constructors

- A first attempt at writing the constructor:

  ```java
  public SavingsAccount(double initialBalance,
                         double initialRate) {
    balance = initialBalance;
    // WRONG; balance is private
    interestRate = initialRate;
  }
  ```

- This version of the constructor won’t compile, because balance was declared private in the Account class.
Writing Subclass Constructors

- There are two ways to solve this problem.
- One is for the SavingsAccount constructor to invoke the Account constructor by using the word super:

```java
public SavingsAccount(double initialBalance,
                        double initialRate) {
    super(initialBalance);
    // Invoke Account constructor
    interestRate = initialRate;
}
```

super must come first, before the other statements in the body of the subclass constructor.
Writing Subclass Constructors

- The `DrawableFrame` class extends `Frame`, a class in the Java API. A `Frame` object is a window with a title at the top.
- When a `DrawableFrame` object is created, a title is needed for the frame:
  ```java
  DrawableFrame df =
      new DrawableFrame("Title goes here");
  ```
- The `DrawableFrame` constructor uses `super` to pass the title to the constructor for its superclass:
  ```java
  public DrawableFrame(String title) {
      super(title); // Invoke Frame constructor
      ...
  }
  ```
Writing Subclass Constructors

- If a subclass constructor fails to call `super`, the compiler will automatically insert `super();` at the beginning of the constructor.

- If a subclass has no constructors at all, the compiler will create a no-arg constructor that contains `super();` but no other statements.
Illustrating Inheritance

- There are various ways to indicate visually that one class is a subclass of another.
- One common technique is to place the superclass above the subclass and draw a line connecting the two:

```
Account

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SavingsAccount</td>
</tr>
</tbody>
</table>
```

- Sometimes an arrow is drawn from the subclass to the superclass.
Illustrating Inheritance

- When two or more classes have the same superclass, a single diagram can be drawn with the superclass at the top and the subclasses below it:

![Diagram showing inheritance的关系]

- CheckingAccount is similar to Account, except that it allows checks to be written on the account.
Illustrating Inheritance

- To save space, the superclass can be put on the left and the subclasses on the right:

```
Account ┌──────────────────────────────────────
    │                                    *
    │                                    *
    │                                    *
    │                                    *
    │                                    *
    │                                    *
    └──────────────────────────────────────
       CheckingAccount
       └──────────────────────────────────────
          SavingsAccount
```

- A simpler notation can be used to show the ancestry of a particular class:

Account → CheckingAccount
Illustrating Inheritance

- Subclasses can have subclasses, making it possible to build up long chains of related classes.
- An example:
  
  Account → CheckingAccount → InterestCheckingAccount

- When a class is the result of a series of extensions, it will have a “direct superclass” (the class that it extends), as well as “indirect” superclasses.
When *Not* to Use Inheritance

- Inheritance is appropriate when the new class “is a” particular case of the old class (the *is-a* relationship):
  - A Car is a Vehicle.
  - A SavingsAccount is an Account.
- If the words “is a” don’t fit, then inheritance shouldn’t be used: it wouldn’t make sense to say that a SavingsAccount is a Vehicle.
- For the *is-a* relationship to exist, the subclass must have every property of the superclass.
When *Not* to Use Inheritance

- Suppose that instances of the `Money` class represent specific dollar amounts.
- If `Account` extends `Money`, every `Account` object would automatically store a monetary amount, but that’s not a good idea.
- `Money`’s public methods would be inherited by `Account`, making it possible to apply `Money` operations to `Account` objects.
- The *is-a* rule says that `Account` shouldn’t extend `Money`: it’s not true that an `Account` is a `Money`.
When *Not* to Use Inheritance

- Instead of having `Account` extend `Money`, it would be better to declare the type of the `balance` variable to be `Money` *instead of* `double`.
- Although the *is-a* relationship doesn’t hold for `Account` and `Money`, there’s clearly some connection between the two classes.
- This is called the *has-a* relationship, because an `Account` object *has* a `Money` object stored within it.
11.2 The **protected** Access Modifier

- **public** and **private** are the primary access modifiers in Java:
  - Declaring a variable or method to be **public** allows universal access to that variable or method.
  - Declaring a variable or method to be **private** limits access to the class in which the variable or method is defined.
- The public/private distinction is sometimes too restrictive, because it doesn’t distinguish a subclass from classes in general.
Using the `protected` Access Modifier

- A subclass can use the public variables and methods in its superclass, but it has no access to private variables and methods in the superclass.
- In Java, subclasses can be given access to superclass variables and methods by declaring them to be `protected` instead of `public` or `private`.
- **Using `protected` provides an intermediate level of access—one that’s more restrictive than `public` but less restrictive than private.**
Differences between Access Modifiers

- A summary of the differences between `private`, `protected`, and `public` when applied to a variable or method in a class:

<table>
<thead>
<tr>
<th>Access Modifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>Can be accessed only in the same class</td>
</tr>
<tr>
<td>protected</td>
<td>Can be accessed in the same class or in a subclass</td>
</tr>
<tr>
<td>public</td>
<td>Can be accessed in any class</td>
</tr>
<tr>
<td>None</td>
<td>Can be accessed in any class in the same package</td>
</tr>
</tbody>
</table>

- A class also has access to the protected variables and methods of all classes in the same package.
Properties of Protected Variables and Methods

- The ability to access protected variables and methods extends to subclasses of subclasses, their subclasses, and so on.
- Consider the following chain of classes:
  Vehicle → MotorizedVehicle → Car
- Methods in the Car class will have access to the protected variables and methods of both Vehicle and MotorizedVehicle.
An Example

- The protected keyword provides another way to solve the problem of writing the `SavingsAccount` constructor—declare the balance variable in the `Account` class to be protected rather than private:
  ```java
  public class Account {
    protected double balance;
    ...
  }
  ``

- The `SavingsAccount` constructor will now have direct access to balance:
  ```java
  public SavingsAccount(double initialBalance,
                        double initialRate) {
    balance = initialBalance;
    interestRate = initialRate;
  }
  ```
Protected Methods

- protected works with methods as well as variables.
- A protected method can be called within its own class and by methods in subclasses.
- When a class contains a helper method that might be useful to its subclasses as well, it’s a good idea to declare the method protected rather than private.
- protected can also be used with class variables and class methods.
Protected Class Variables

- In Section 10.7, the Account class had a private class variable named totalDeposits:
  ```java
  public class Account {
      private static double totalDeposits = 0.0;
      ...
  }
  ```

- Suppose that a creditInterest method is added to the SavingsAccount class:
  ```java
  public class SavingsAccount extends Account {
      public void creditInterest() { ... }
      ...
  }
  ```
Protected Class Variables

- Crediting interest will increase not only the balance in the account itself, but the total deposits for the entire bank.
- In order to allow `creditInterest` to update the `totalDeposits` variable directly, it can be declared `protected`:
  ```java
  public class Account {
      protected static double totalDeposits = 0.0;
      ...
  }
  ```
protected Versus private

- Declaring instance variables to be protected exposes them to all subclasses.
- For this reason, it’s usually best to avoid protected variables.
- It’s better to make variables private and provide access methods to fetch and/or modify their values.
- These methods can be declared protected if they’ll be needed only by subclasses, not by all classes.
11.3 Overriding

- Although most cars have steering wheels, some have had a steering bar instead.
- Fortunately, the object-oriented view of the world can accommodate variation through a mechanism known as **overriding**.
- A subclass can **override** an inherited instance method by supplying a new method with the same name and return type.
- In addition, both methods must have the same number of parameters, and the types of corresponding parameters must be the same.
An Example of Overriding

- Suppose that the `Account` class has a method named `printStatement` that prints an account statement.
- A `CheckingAccount` class might need a different `printStatement` method.
- To override the inherited `printStatement` method, `CheckingAccount` would have a `printStatement` method with the same return type and parameters as the one in `Account`.
Using `super` to Call an Overridden Method

- One tricky situation can arise when writing a subclass method: calling a method that’s inherited from the superclass but overridden in the new class.
- Calling it in the normal way won’t work, because the compiler will assume that the call refers to the new version.
Using `super` to Call an Overridden Method

- Suppose that the CheckingAccount version of printStatement tries to call the Account version:
  ```java
  void printStatement() {
      printStatement(); // Print regular statement
      ... // Then print list of checks
  }
  
  The compiler will assume that printStatement is calling itself, not the version that was inherited.
Adding `super` to the call of `printStatement` forces the compiler to use the superclass version of the method:

```java
void printStatement() {
    super.printStatement();
    // Print regular statement
    // Then print list of checks
}
```

When a method is called using `super`, Java looks first in the direct superclass.

If the method isn’t found there, Java looks in the superclass of that class, and so on.
Hiding Inherited Class Methods

- Although class methods can’t be overridden, a class method can hide an inherited class method.
- The requirements for hiding a class method are the same as for overriding an instance method:
  - The new method must have the same name and return type as the inherited method.
  - The methods must have the same number of parameters, and the types of corresponding parameters must be the same.
11.4 Polymorphism

- Inheritance becomes even more potent when combined with another concept: **polymorphism**.
- Consider giving instructions to someone to operate a vehicle:
  1. Start vehicle
  2. Release brake
  3. Accelerate
  4. Apply brake
  5. Stop vehicle
- These instructions work for any type of vehicle, not just a car.
Polymorphism

- The exact meaning of each instruction may vary, depending on the vehicle:
  - For a car, “accelerate” means “press on the accelerator.”
  - For a bicycle, “accelerate” means “turn the pedals.”
- In object-oriented programming, polymorphism refers to the ability of different kinds of objects to respond differently to the same commands, provided that the objects belong to classes with a common ancestor.
The Substitution Principle

- To allow polymorphism, Java has a rule that might be called the **Substitution Principle**:

  An instance of a subclass can take the place of an instance of any of its superclasses.
The Substitution Principle

Although the Substitution Principle may seem odd, there are good reasons for it:

- If the *is-a* relationship holds between the subclass and the superclass, then the subclass is a more specialized version of the superclass.
- An object that belongs to the subclass contains the same variables (plus possibly some more) as any instance of the superclass.
- An instance of the subclass responds to the same (public) methods as an instance of the superclass.
The Substitution Principle

- An example that uses the Substitution Principle:
  
  ```java
  Vehicle v = new Car();
  ```

- Although this statement looks wrong, it is valid as long as `Vehicle` is a superclass of `Car`.

- A car is a vehicle: it has all the properties and behaviors of a vehicle.

- Some behaviors were inherited and left unchanged, while others were overridden, but—either way—they’re guaranteed to exist.
A Polymorphic Algorithm in Java

- A Java version of the algorithm for operating a vehicle:
  ```java
  v.start();
  v.releaseBrake();
  v.accelerate();
  v.applyBrake();
  v.stop();
  ```
- It doesn’t matter what `v` is, as long as it’s some kind of vehicle.
- This algorithm is polymorphic: it works for a variety of vehicles, not a single kind.
Dynamic Binding

- In a call such as `v.start()`, the compiler can’t determine which version of `start` is being called; instead, `v` will have to be tested during program execution.

- This process is known as **dynamic binding**, because the exact method that’s being called won’t be known until the program is run.

- If different objects are assigned to `v` during execution, different versions of `start` may be called:

  ```java
  v = new Car();
  v.start(); // Calls start method in Car class
  v = new Truck();
  v.start(); // Calls start method in Truck class
  ```
Other Applications of the Substitution Principle

- The Substitution Principle applies to more than just assignment:
  - A method whose parameter type is a class can be passed an instance of a subclass instead.
  - A method whose result type is a class may return an instance of a subclass instead.
  - The $==$ and $!=$ operators can be used to compare references to instances of different classes, provided that one class is a subclass of the other.
Casting Object References

- Consider the following method:
  ```java
  static void updateAccount(Account acct) {
      ...
  }
  ```
- Thanks to the Substitution Principle, the argument in a call of `updateAccount` could be a `SavingsAccount` object or a `CheckingAccount` object.
- As long as `updateAccount` performs only operations on `acct` that are valid for any type of account, there won’t be any problems.
Casting Object References

- It’s possible that `updateAccount` needs to determine what type of account it’s dealing with.

- Java’s `instanceof` operator can be used to solve problems such as this. An expression of the form `object instanceof class` has the value `true` only if `object` is an instance of `class` or one of its subclasses.

- Suppose that `savingsAcct` contains a reference to a `SavingsAccount` object. Then:

  ```java
  savingsAcct instanceof SavingsAccount \Rightarrow true
  savingsAcct instanceof Account \Rightarrow true
  ```
Casting Object References

- Knowing that an object belongs to a particular class doesn’t necessarily make it possible to use the full set of methods defined for that class.
- Even if acct stores a SavingsAccount object, the compiler will allow only the use of Account methods, because acct is declared to be of type Account.
- Casting acct to type SavingsAccount will force the compiler to accept acct as a SavingsAccount object.
Casting Object References

- A statement that casts `acct` to type `SavingsAccount`:
  ```java
  SavingsAccount savingsAcct = (SavingsAccount) acct;
  ```
- The `savingsAcct` variable can now be used to call `creditInterest` or any other `SavingsAccount` method.
Casting Object References

- A statement that calls `creditInterest` if `acct` belongs to the `SavingsAccount` class:
  ```java
  if (acct instanceof SavingsAccount) {
      SavingsAccount savingsAcct =
          (SavingsAccount) acct;
      savingsAcct.creditInterest();
  }
  ```

- Be careful when trying to cast an object to a subclass. If the object isn’t an instance of the subclass, a `ClassCastException` will be thrown.

- It’s often a good idea to use `instanceof` to test an object before attempting to cast it.
Advantages of Polymorphism

- Polymorphism is especially useful when different types of objects are stored in the same data structure.
- It becomes possible to write general-purpose code that processes every object in the data structure without checking to see which class it belongs to.
- For example, statements can be printed for an array of Account objects without first checking the type of each account.
Advantages of Polymorphism

- Polymorphism becomes especially important during program maintenance.
- If a new subclass of Account is created, it won't be necessary to go through the entire program to see how this change might affect code that performs operations on accounts.
- If an existing subclass of Account is removed, most of the program should be unaffected.
Advantages of Polymorphism

- Excessive use of the `instanceof` operator can negate the advantages of polymorphism.
- Suppose that the `Account` class has an empty `creditInterest` method.
- The `SavingsAccount` class will override this method, but the `CheckingAccount` class will inherit the empty method from `Account`.
- Now, the `updateAccount` method won’t need to test whether `acct` refers to a `SavingsAccount` object before calling `creditInterest`:

```
acct.creditInterest();
```
11.5 The **Object** Class

- Every class—with the exception of a special class named **Object**—is required to have a superclass.

- If no superclass is specified in the declaration of a new class, **Java uses Object** as the default superclass.

- Because of this rule, all classes (other than **Object itself**) have **Object** as a superclass, either directly or indirectly.
The Java Class Hierarchy

Java’s classes belong to a single “family tree,” known as a class hierarchy:
The Java Class Hierarchy

- The existence of a single class hierarchy has some important consequences:
  - Every class (other than `Object` itself) inherits methods from the `Object` class.
  - A variable of type `Object` can store a reference to any object whatsoever.
  - A method with an `Object` parameter will accept any object as its argument.
Object Methods

- A partial list of methods in the `Object` class:
  - `clone()` — Returns a copy of this object.
  - `equals(obj)` — Indicates whether the object `obj` is “equal” to this object.
  - `toString()` — Returns a string representation of this object.

- These methods are inherited by the subclasses of `Object`, so that every class in Java has these methods.

- These methods are frequently overridden.
The **equals** Method

- The `equals` method is provided to solve the problem of testing whether two objects are equal.
- `equals` has one parameter, an arbitrary object, and returns a boolean result:
  ```java
  public boolean equals(Object obj) {
      ...
  }
  ```
- By default, the `equals` method behaves like the `==` operator. The call `x.equals(y)` returns `true` only if `x` and `y` refer to the same object.
The **equals** Method

- If the default behavior of the `equals` method is unsatisfactory, a class can override the method.

- **An equals method for the Fraction class:**

```java
public boolean equals(Object obj) {
    if (!(obj instanceof Fraction))
        return false;
    Fraction f = (Fraction) obj;
    return (numerator == f.numerator &&
            denominator == f.denominator);
}
```
The *equals* Method

- A statement that tests whether two *Fraction* objects are equal:
  ```java
  if (f1.equals(f2)) ...
  ```
- The inherited version of *equals* would have tested whether `f1` and `f2` refer to the same object.
- The new version will test whether `f1` and `f2` have matching numerators and denominators.
The `equals` Method

- The `equals` method needs to behave in the manner expected of an equality test.
- In particular, if `x.equals(y)` returns `true`, then `y.equals(x)` should also.
- It's not necessary to provide equality testing for every class:
  - Equality testing may not be meaningful for a class.
  - It may not be worthwhile to take the time to write the `equals` method.
The `toString` Method

- **Because `toString` is declared in the `Object` class, all classes must have a `toString` method.**
- **The `print` and `println` methods are overloaded, with one version of each method requiring an `Object` parameter.**
- **When supplied with an object as its argument, `print` (or `println`) calls the `toString` method that belongs to the object and prints the string returned by `toString`.**
The `toString` Method

- If an object belongs to a class that doesn’t have a `toString` method, `print` will use an inherited version of `toString`.
- The version of `toString` in the `Object` class returns a string containing the name of the object’s class, the `@` character, and a hexadecimal number (the object’s “hash code”).
The `toString` Method

- If the `Fraction` class didn’t contain a `toString` method, printing a fraction would cause `Object`’s version of `toString` to be invoked.
- The resulting output would have the form `Fraction@1cc78a`
- This information isn’t very useful, so it’s a good idea for most classes to provide their own `toString` method.
11.6 Abstract Classes

- Some classes are purely artificial, created solely so that subclasses can take advantage of inheritance.
- The Vehicle class was created for convenience—in the real world, there are no “generic” vehicles, only specific types of vehicles.
- The same reasoning applies to the Account class: banks offer only specific kinds of accounts.
- In Java, artificial classes like Vehicle and Account are called abstract classes.
Characteristics of Abstract Classes

- The declaration of an abstract class must include the word `abstract`, which is usually placed just before the word `class`.
- Some of the methods in an abstract class may be `abstract methods`.
- An abstract method is a “dummy” method that has no body:
  ```
  public abstract double doubleValue();
  ```
- It is illegal to create an instance of an abstract class.
Uses of Abstract Classes

- Abstract classes are useful only as starting points for defining subclasses.
- They often arise from “bottom-up” design, where the strategy is to first identify which classes are needed and then look for ways to “factor out” whatever those classes have in common.
- The result is an abstract class that can be extended to create the classes that are actually needed.
An Example of an Abstract Class

- During the design of classes that represent savings accounts and checking accounts, it will be apparent that both types of accounts have a great deal in common.
- To take advantage of this commonality, it makes sense to create an abstract `Account` class, then define `SavingsAccount` and `CheckingAccount` classes by extending `Account`.
- Account would define variables and methods that are common to both types of accounts.
Abstract Subclasses

- Most of the time, subclasses of an abstract class will provide bodies for all inherited abstract methods.
- These subclasses are normal classes and can be used to create objects.
- Occasionally, a subclass will fail to override all the abstract methods it inherits.
- In that case, the subclass is itself an abstract class, good only for defining further subclasses.
Example: A **Shape** Class

- Suppose that the need arises for a series of classes that represent specific geometric shapes, such as Circle and Rectangle.

- These classes have much in common, so work can be saved by first creating a generic **Shape** class.

- No **Shape** objects will ever be created. Instead, the **Shape** class will serve solely as a starting point for defining more-specific shape classes.
Example: A **Shape** Class

- **Shape** will have instance variables representing the properties that are common to all shapes:
  - Location (a pair of x and y coordinates)
  - Color (a **Color** object)

- Every shape will also have a width and a height.

- It might not be a good idea for the **Shape** class to have **width** and **height** variables, which would force all subclasses to inherit these variables.

- Some subclasses, such as **Circle**, won’t need these variables.
Example: A **Shape** Class

- **Shape** will have instance methods representing the behaviors that are common to all shapes:
  - `draw`
  - `move`
  - `getX, getY, getColor, getWidth, getHeight`
  - `setColor`

- **The** `draw, getHeight, and getWidth methods will have to be abstract**—they can’t be written without knowing what kind of shape is involved.**
Shape.java

// Represents a geometric shape that can be displayed in a graphics context

import java.awt.*;

public abstract class Shape {
    // Instance variables
    private int x;
    private int y;
    private Color color;

    // Constructor
    protected Shape(int x, int y, Color color) {
        this.x = x;
        this.y = y;
        this.color = color;
    }
}
// Abstract methods
public abstract void draw(Graphics g);
public abstract int getHeight();
public abstract int getWidth();

// Other instance methods
public Color getColor() {
    return color;
}

public int getX() {
    return x;
}

public int getY() {
    return y;
}
public void move(int dx, int dy) {
    x += dx;
    y += dy;
}

public void setColor(Color color) {
    this.color = color;
}
}
The Circle Class

- The Circle class will need a diameter instance variable.
- Circle will need to override the abstract methods that were inherited from the Shape class: draw, getHeight, and getWidth.
// Represents a circle that can be displayed in a graphics context

import java.awt.*;

public class Circle extends Shape {
    // Instance variables
    private int diameter;

    // Constructor
    public Circle(int x, int y, Color color, int diameter) {
        super(x, y, color);
        this.diameter = diameter;
    }
}
// Instance methods
    
    public void draw(Graphics g) {
        g.setColor(getColor());
        g.fillOval(getX(), getY(), diameter, diameter);
    }
    
    public int getHeight() {
        return diameter;
    }
    
    public int getWidth() {
        return diameter;
    }
}
The Rectangle Class

- The Rectangle class will need width and height instance variables.
- Like the Circle class, Rectangle will need to override the draw, getHeight, and getWidth methods.
// Represents a rectangle that can be displayed in a graphics context

import java.awt.*;

public class Rectangle extends Shape {
    // Instance variables
    private int width;
    private int height;

    // Constructor
    public Rectangle(int x, int y, Color color,
                     int width, int height) {
        super(x, y, color);
        this.width = width;
        this.height = height;
    }
}
// Instance methods
public void draw(Graphics g) {
    g.setColor(getColor());
    g.fillRect(getX(), getY(), width, height);
}

public int getHeight() {
    return height;
}

public int getWidth() {
    return width;
}
}
11.7 Final Classes and Methods

- The keyword `final` can be applied to a class or method.
- When `final` is present in a class declaration, it indicates that the class can’t be extended:

  ```java
  public final class SavingsAccount extends Account {
    ...
  }
  ``

  `SavingsAccount` is not allowed to have subclasses.
Final Classes

- Many API classes are final, including Math, String, and the “wrapper” classes, which include Integer, Double, and Character.
- String is declared final to guarantee that String variables always contain immutable objects.
Effect of `final` on Performance

- `final` can have an effect on program performance.
- Consider the following method call, where `x` is a variable of type `XClass`:
  
  ```
  x.f();
  ```

  Since `x` might store a reference to an instance of some subclass of `XClass`, the compiler inserts code to check `x` during program execution and call the appropriate version of `f`.

- If `XClass` were declared `final`, the compiler would know that `x` refers to an `XClass` object, so it would produce code that calls `f` without first checking `x`. 
Final Methods

- Individual methods can be declared `final` without declaring the entire class to be `final`.
- It’s illegal to override a final method, so declaring a method to be `final` provides a performance enhancement when that method is called.
- Methods that are declared either `static` or `private` are automatically `final`, so a class method or a private instance method can’t be overridden.
11.8 Inheritance in the Abstract Window Toolkit

- Java’s AWT (Abstract Window Toolkit) provides classes that are used to write programs with a graphical user interface (GUI).
- A GUI program provides the user with a window containing interactive graphical elements.
A GUI Window

- A window created by a typical GUI program:
Components

- The window contains several visual *components*:  
  - Lists  
  - Text fields  
  - Labels  
  - Buttons

- The components are shown with a Windows appearance. They may look somewhat different on other platforms.

- In the AWT, each type of component is represented by a class.
Buttons

- An instance of the `Button` class represents a labeled button:

- The user can “press” a button by pointing to it with the mouse and pressing a mouse button.
- When a button is pressed, it changes appearance:
Checkboxes

- An instance of the `Checkbox` class represents a box that can be clicked “on” or “off”:

  ![Checkbox](image)

  - Clicking on the box causes a check mark to appear:

  ![Checkbox](image)

- Clicking on the box again causes the check mark to disappear.

  ![Checkbox](image)
Choice Menus

- An instance of the `Choice` class represents a choice menu that displays one of several items:

- Clicking on the arrow button causes the full list to appear:
Choice Menus

- The user can now drag the mouse to move the highlight to any of the listed items.
- When the mouse button is released, the list disappears, and the selected item replaces the one originally displayed.
Labels

- An instance of the `Label` class represents a text string that can be positioned next to other components.
Lists

- An instance of the `List` class represents a list of items:

- The user can choose an item by clicking on it:
Lists

- If not all items are visible, a scrollbar appears to the right of the list:

![Scrollbar Example]
Scrollbars

- An instance of the `Scrollbar` class represents a sliding bar. Scrollbars can be either horizontal:

or vertical:
Text Areas

- An instance of the `TextArea` class represents a multiline area in which text can be displayed:

  ![Multiline text with scrollbars]

- Scrollbars at the bottom and right side make it possible for the user to view text that’s not otherwise visible. If desired, the user can edit the text.
Text Fields

- An instance of the `TextField` class represents an area in which a single line of text can be displayed:

  ![Your name here]

- Like a text area, a text field can be made editable if desired.
Relationships Among Component Classes

- **The Component class** represents the properties and behavior that all components have in common.

- Some of the properties shared by all components:
  - *Position*: The position of the component on the screen.
  - *Size*: The height and width of the component.
  - *Visibility*: Whether or not the component is currently visible.
  - *Colors*: The component’s foreground and background colors.

- The variables that keep track of these properties belong to **Component**, as do the access methods for testing and modifying these properties.
Relationships Among Component Classes

- Component is an abstract class that serves as the superclass for more-specific component classes.
- The existence of the Component class means that the individual component classes are relatively small and easy to write.
- The Component class can also be used as a starting point for writing custom components that aren’t provided by the AWT.
Relationships Among Component Classes

- Each component must belong to a window or some other kind of “container.”
- Container classes in the AWT:
  - Container—Serves as a superclass for all the other container classes.
  - Dialog—A dialog box.
  - FileDialog—A file dialog box.
  - Frame—A window with a title, a border, and possibly a menu.
  - Panel—A container with no border.
  - Window—A window with no title or border.
Relationships Among Component Classes

- Container is a subclass of Component, so any container can be used as a component, thanks to the Substitution Principle.
- As a result, containers can be nested inside other containers.
The component classes are related to each other through inheritance:
Relationships Among Component Classes

- `TextArea` and `TextField` have many properties and methods in common, which they inherit from `TextComponent`.
11.9 Case Study: “Nervous” Shapes

- The NervousShapes program will create a frame containing a random mixture of circles and rectangles with random colors, sizes, and positions.
- After a brief delay, the shapes will be moved to slightly different positions, with the direction of motion for each shape chosen randomly.
Program Behavior

- The new $x$ coordinate for each shape will either be the same as the old $x$ coordinate, one pixel smaller, or one pixel larger.
- The new $y$ coordinate will be computed in a similar manner.
- The size of the frame’s drawing area will be fixed at 200 pixels by 200 pixels.
- The number of shapes displayed will be 50.
- The “delay time”—the interval between animation cycles—will be 10 milliseconds.
Program Behavior
Design of the NervousShapes Program

- The program will consist of three steps:
  1. Create a frame labeled “Nervous Shapes.”
  2. Create 50 random Circle and Rectangle objects.
  3. Display the shapes within an animation loop.
- Each step will be done by a helper method.
- The helper methods will be named createWindow, createShapes, and animateShapes.
- createWindow is easy to write.
The `createShapes` Method

- `createShapes` will need a `shapes` array in which to store the 50 shapes that it creates.
- **Because `animateShapes` will also need access to the shapes array, it will be a class variable.**
- `shapes` will be declared as an array of `Shape` objects so that it can store both `Circle` objects and `Rectangle` objects.
The `createShapes` Method

- `createShapes` will execute the following steps 50 times:
  1. Select a color in which each component is a random number between 0 and 255.
  2. Decide whether the new shape will be a circle or a rectangle.
  3. If it’s a circle, choose a random diameter between 10 and 20, choose a random position, create a `Circle` object, and store it in the `shapes` array. If it’s a rectangle, choose a random width and height between 10 and 20, choose a random position, create a `Rectangle` object, and store it in the `shapes` array.
The **animateShapes** Method

- animateShapes will consist of an infinite loop that repeats the following steps:
  1. Clear the frame’s drawing area.
  2. Move each shape in the shapes array to a new position and display it. The new x coordinate for each shape will be the same as the old one, one pixel smaller, or one pixel larger; the new y coordinate is computed in a similar fashion. A move is not performed if it would cause any part of the shape to go outside the frame’s drawing area.
  3. Call `repaint` to update the screen.
  4. Pause briefly.
The **animateShapes** Method

- To display element \( i \) of the `shapes` array, `animateShapes` will call the `draw` method:
  ```java
  shapes[i].draw(g);
  ```
- Thanks to dynamic binding, there’s no need to test whether `shapes[i]` is a `Circle` object or a `Rectangle` object before calling `draw`. 
The `generateRandomInt` Method

- One more helper is needed: `generateRandomInt`, which returns an integer chosen randomly from a specified range of integers.
NervousShapes.java

// Program name: NervousShapes
// Author: K. N. King
// Written: 1999-08-12

// Displays a frame containing a random mixture of circles and rectangles with random colors, sizes, and positions. The shapes periodically change position, with the direction of motion chosen randomly for each shape. The new x coordinate for each shape will either be the same as the old x coordinate, one pixel smaller, or one pixel larger; the new y coordinate will be computed in a similar manner. Shapes will be constrained so that they do not move outside the drawing area.

import java.awt.*;
import jpb.*;
public class NervousShapes {
    // Constants
    private static final int DELAY = 10;
        // Animation delay (milliseconds)
    private static final int MAX_SIZE = 20;
        // Maximum width and height of a shape
    private static final int MIN_SIZE = 10;
        // Minimum width and height of a shape
    private static final int NUM_SHAPES = 50;
        // Number of shapes
    private static final int WINDOW_SIZE = 200;
        // Width and height of drawable portion of frame

    // Class variables
    private static DrawableFrame df;
        // Frame in which shapes are displayed
    private static Graphics g;
        // Graphics context for frame
    private static Shape shapes[] = new Shape[NUM_SHAPES];
        // Array of shapes
public static void main(String[] args) {
    createWindow();
    createShapes();
    animateShapes();
}
private static void createWindow() {
    // Create a frame labeled "Nervous Shapes" and set its size
    df = new DrawableFrame("Nervous Shapes");
    df.show();
    df.setSize(WINDOW_SIZE, WINDOW_SIZE);
    
    // Get the frame's graphics context
    g = df.getGraphicsContext();
}
// NAME:       createShapes
// BEHAVIOR:   Creates enough Circle and Rectangle objects
to fill the shapes array. Each shape has a random color, size, and position. The height and width of each shape must lie between MIN_SIZE and MAX_SIZE (inclusive). The position is chosen so that the shape is completely within the drawing area.
// PARAMETERS: None
// RETURNS:    Nothing

private static void createShapes() {
    for (int i = 0; i < shapes.length; i++) {
        // Select a random color
        int red = generateRandomInt(0, 255);
        int green = generateRandomInt(0, 255);
        int blue = generateRandomInt(0, 255);
        Color color = new Color(red, green, blue);
// Decide whether to create a circle or a rectangle
if (Math.random() < 0.5) {
    // Generate a circle with a random size and position
    int diameter = generateRandomInt(MIN_SIZE, MAX_SIZE);
    int x = generateRandomInt(0, WINDOW_SIZE - diameter);
    int y = generateRandomInt(0, WINDOW_SIZE - diameter);
    shapes[i] = new Circle(x, y, color, diameter);
} else {
    // Generate a rectangle with a random size and position
    int width = generateRandomInt(MIN_SIZE, MAX_SIZE);
    int height = generateRandomInt(MIN_SIZE, MAX_SIZE);
    int x = generateRandomInt(0, WINDOW_SIZE - width);
    int y = generateRandomInt(0, WINDOW_SIZE - height);
    shapes[i] = new Rectangle(x, y, color, width, height);
}
private static void animateShapes() {
    while (true) {
        // Clear drawing area
        g.setColor(Color.white);
        g.fillRect(0, 0, WINDOW_SIZE - 1, WINDOW_SIZE - 1);
    }
}
for (int i = 0; i < shapes.length; i++) {
    // Change the x coordinate for shape i
    int dx = generateRandomInt(-1, +1);
    int newX = shapes[i].getX() + dx;
    if (newX >= 0 &&
        newX + shapes[i].getWidth() < WINDOW_SIZE)
        shapes[i].move(dx, 0);

    // Change the y coordinate for shape i
    int dy = generateRandomInt(-1, +1);
    int newY = shapes[i].getY() + dy;
    if (newY >= 0 &&
        newY + shapes[i].getHeight() < WINDOW_SIZE)
        shapes[i].move(0, dy);

    // Draw shape i at its new position
    shapes[i].draw(g);
}

// Call repaint to update the screen
df.repaint();
// Pause briefly
try {
    Thread.sleep(DELAY);
} catch (InterruptedException e) {}