Chapter 14

Extending Classes

Two important concepts in object-oriented programming are *composition* and *inheritance*. In *Think Java, 2nd Edition*, you have seen composition in Chapter 13 and inheritance in Chapter 14, but mostly by example. We're going to go into more detail on both of these concepts here.

14.1 Composition

Composition happens when an object is made up of other objects. We saw this in Chapter 13, where a `Deck` was made up of an array of `Card` objects:

```
public class Deck {
    private Card[] cards;
    ...
}
```

Composition is also known as a *Has-A* relationship. (A `Deck` *has an* array of `Cards`). Put formally, the `Deck` is the *aggregating class*, and the `Card` class is is the *aggregated class*.

When we draw a UML diagram involving composition, as in Figure 14.1, we use an arrow with a diamond head to indicate aggregation, and we can also add notation here that shows that one `Deck` contains many `Card` objects.

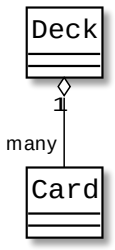

Figure 14.1: UML Diagram Showing Aggregation - a Deck has Card(s)

Let's use composition to build a Java simulation of a toaster. What things is a toaster built from?

- A chassis with a number of slots (at least one, at most four)
- A lever to push bread down or pop it out
- A power supply to turn the toaster on and off
- A dial to control the darkness (1=light, 10=dark)

The first two of these are built by the toaster company. But the company that makes toasters doesn't build the power supply or the dial (which has circuitry to control the current flow). Instead, those are parts they order from some other companies and put into the chassis.

That means that our `Toaster` *has-a* `PowerSupply` object and *has-a* `Dial` object. Figure 14.2 shows the UML diagram. You can see the full code in the file *ch14/ToasterTest.java* in the repository.

Before we see how composition affects the way we construct and manipulate objects, let's give a few more instances of composition (*has-a*) relationships. Using composition reflects the way objects are built out of other objects (parts). Each of the sub-parts has its own attributes and things that it can do (its "methods"). Notice that we sometimes need multiple instances of a sub-part when constructing the larger object.

- A printer has a power supply, printer drum, and toner cartridge.
- A bicycle has a gear assembly, handbrakes (2), and tires (2).

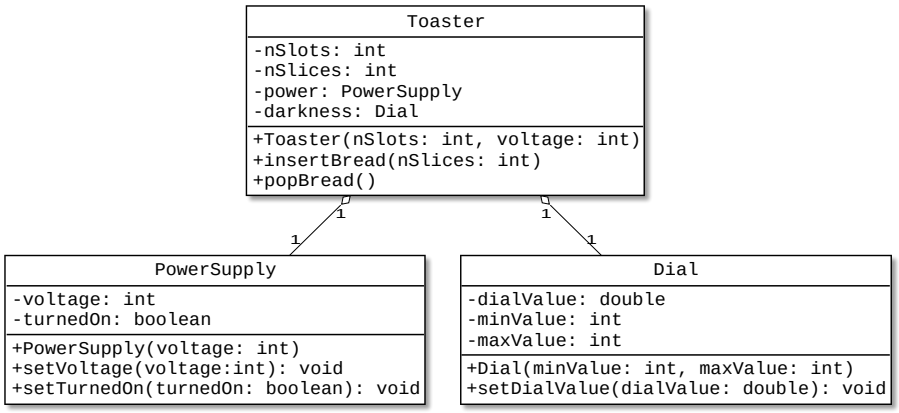

Figure 14.2: UML Diagram Showing a Toaster composed of a PowerSupply and Dial

- A refrigerator has a power supply, an icemaker, and a compressor.
- A window in a word processor has a text area, a ribbon (icons for manipulating text), and two scroll bars (horizontal and vertical).

Back to the toaster. Here are constructors for the `PowerSupply` and `Dial` classes:

```
public PowerSupply(int voltage) {
    if (voltage == 220) {
        this.voltage = voltage;
    } else {
        this.voltage = 110;
    }

    // new power supplies are always turned off
    this.turnedOn = false;
}

public Dial(int minValue, int maxValue) {
    this.minValue = minValue;
    this.maxValue = maxValue;
    // new dials are always set to lowest value
    this.dialValue = minValue;
}
```

Now, look at how the `Toaster` class starts, with line numbers for reference:

```
 1 class Toaster {
 2     private int nSlots;
 3     private int nSlices;
 4     private PowerSupply power;
 5     private Dial darkness;
 6
 7     public Toaster(int nSlots, int voltage) {
 8         this.nSlots = Math.max(1, Math.min(4, nSlots));
 9         this.nSlices = 0;
10         this.power = new PowerSupply(voltage);
11         this.darkness = new Dial(1, 10);
12     }
13     // ...
14 }
```

The constructor in line 7 has two parameters: the number of slots in the toaster and what voltage it should have. The number of slots and number of slices of bread currently in the toaster are attributes belonging to the `Toaster` class, and those get set directly.

The power supply is an object, which is why line 10 has to call the `PowerSupply` constructor to build a power supply with the desired voltage. Think of this as the toaster company calling up the power supply company and telling them "send me a 110-volt power supply" and putting that power supply into the finished toaster.

Similarly, line 11 has to call the `Dial` constructor to build a dial with a range of 1-10.

Although a real-life toaster is composed of parts, all of the controls are on the toaster's exterior. We don't expect—or want—the customer to have to directly access the power supply to turn the toaster on! Similarly, we want to provide methods in the `Toaster` class that will give programs that use the class access to the private methods and attributes of `power` and `darkness`:

```
public boolean isTurnedOn() {
    return this.power.isTurnedOn();
}

public void setTurnedOn(boolean turnedOn) {
    this.power.setTurnedOn(turnedOn);
}

public double getDialValue() {
    return this.darkness.getDialValue();
}

public void setDialValue(double dialValue) {
    this.darkness.setDialValue(dialValue);
}
```

Now we can write a program that creates a `Toaster` object and makes it do things:

```
public class ToasterTest {
    public static void main(String[] args) {
        Toaster euroFour = new Toaster(4, 220);

        euroFour.setTurnedOn(true);
        euroFour.setDialValue(4.5);
        euroFour.insertBread(1);

        System.out.println(euroFour);
    }
}
```

To summarize this discussion of composition:

- Use composition when you have objects that are built up from other objects.
- Provide methods in an object to give users access to `private` attributes and methods of the sub-objects.

14.2 Inheritance

An *IsA* relationship is known as *inheritance*. For example, a `Hand` class (subclass) also *is-a* `CardCollection` class (superclass). In Java, we use `extends` to indicate inheritance:

```
public class Hand extends CardCollection {
    ...
}
```

Figure 14.3 shows the UML for inheritance, using an open arrowhead:

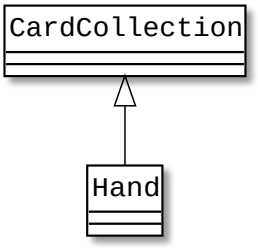

Figure 14.3: UML Diagram Showing Inheritance - a Hand is a CardCollection

Other examples of this sort of relationship:

- An electric bicycle *is a* bicycle (with extra attributes and capabilities)
- A trumpet *is a* bugle with valves (keys) that give the ability to play more notes
- An alarm clock *is a* desk clock with extra attributes and capabilities
- In a computer application, a "combo box" *is a* drop-down menu with extra capabilities (you can type the value as well as select it from the list)

Without inheritance, we would have to represent the `Bicycle` and `ElectricBicycle` classes as shown in Figure 14.4

That's a lot of duplicated code when we translate it to Java. If we say that `ElectricBicycle extends Bicycle`, that means that `ElectricBicycle` *inherits* all of `Bicycle`'s methods and attributes. All we have to include now in

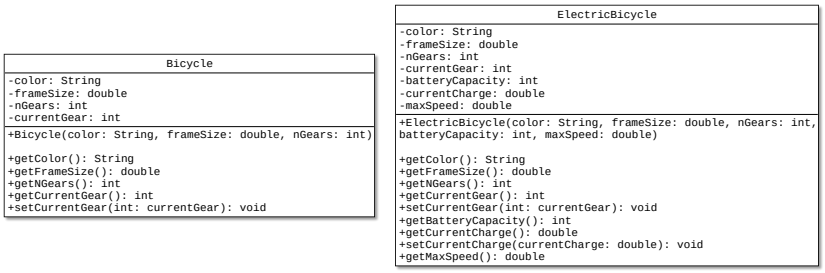

Figure 14.4: UML Diagram of Two Separate Classes for Bicycle and ElectricBicycle

`ElectricBicycle` are the attributes and methods that add extra capabilities to an electric bicycle, as shown in Figure 14.5

In this example, `Bicycle` is called the *superclass* or *parent class*, and `ElectricBicycle` is called the *subclass* or *child class*. As in the real world, the child inherits things from the parent.[1]

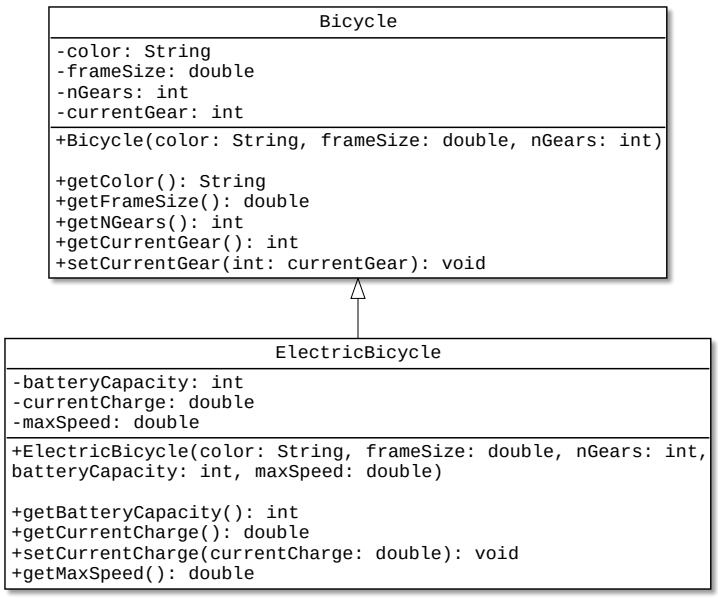

Figure 14.5: UML Diagram of ElectricBicycle Extending the Bicycle Class

Before we go into details about how to use inheritance in Java, let's take a

[1] With one exception: as the joke says, "Parents inherit their gray hair from their children."

break for a short exercise:

14.3 Exercise: Composition vs. Inheritance

Exercise 14.1 In this exercise, we will explore *has-a* and *is-a* relationships. First, state whether the relationship of the following classes is composition or inheritance, and draw the UML diagram showing that relationship.

- `Address` and `Student`
- `Car` and `Vehicle`
- `Account` and `SavingsAccount`
- `State`, `Capital`, and `Country`
- `Instructor`, `Course`, and `Textbook`
- `Dog`, `Cat`, and `Animal`
- `Rectangle`, `Circle`, `Square`, and `Shape`

For classes that exhibit the inheritance relationship, could you name a few data fields/attributes for the superclass? Could you name a few for the subclass only?

For example, `Teacher` (subclass) is also a `Person` (superclass). Data fields for `Person` are: name, age, address, etc. Data fields for `Teacher` are: school, hire date etc.

14.4 Inheritance (continued)

Let's use a smaller example for our continued discussion of inheritance. We'll have an `Item` class, which represents an item in a store. An `Item` instance has a name, a SKU (stock keeping unit), and a price. This is the parent class. We also have a `SaleItem` class, which is a child class. In addition to a name, SKU, and price (which it inherits from the parent class), a `SaleItem` has a discount

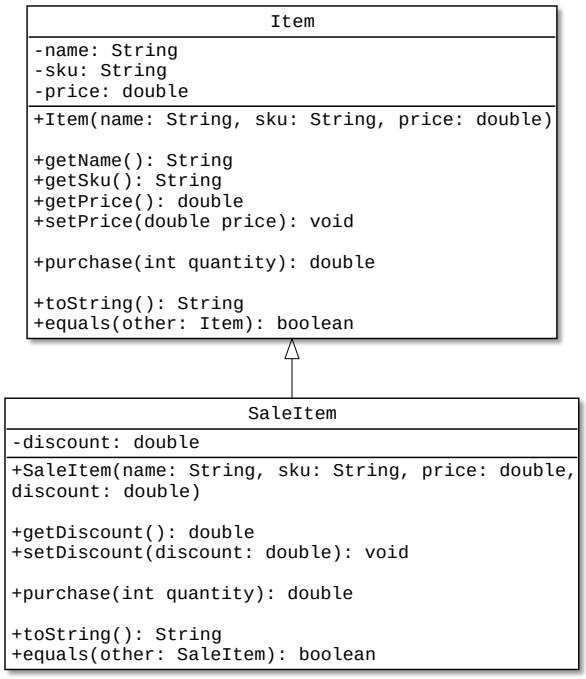

Figure 14.6: UML Diagram of Item and SaleItem classes

percentage (expressed as a decimal). Figure 14.6 shows the UML diagram for these two classes:

There are no setters for the `name` and `sku` attributes; once an item is created, those attributes should never change.

Here is the code for `Item`, which you will find in the repository in file *ch14/Item.java*:

```
public class Item {
    private String name;
    private String sku;
    private double price;

    public Item(String name, String sku, double price)  {
        this.name = name;
        this.sku = sku;
        this.price = Math.abs(price);
    }

    public String getName() {
        return this.name;
    }

    public String getSku() {
        return this.sku;
    }

    public double getPrice() {
        return this.price;
    }

    public void setPrice(double price) {
        this.price = Math.abs(price);
    }

    public double purchase(int quantity) {
        return this.price * quantity;
    }

    public String toString() {
        return String.format("%s (%s): $%.2f", this.name,
            this.sku, this.price);
    }

    public boolean equals(Item other) {
        return (this.name.equals(other.name) &&
            this.sku.equals(other.sku) &&
            this.price == other.price);
    }
}
```

Let's start on the code for `SaleItem`, in file *ch14/SaleItem.java*. You might be tempted to write this:

```
public class SaleItem extends Item {
    private double discount; // as a decimal

    public SaleItem(String name, String sku, double price,
       double discount)  {
       this.name = name;
       this.sku = sku;
       this.price = Math.abs(price);
       this.discount = Math.max(0,
           Math.min(discount, 1.00));
    }
}
```

But that won't work, because `name`, `price`, and `sku` are `private` to the `Item` class. We can't call the setter methods for `name` and `sku` because there aren't any. What we need to do is call the superclass constructor, which *can* set those attributes. To call a superclass constructor, you use the keyword `super`:

```
public class SaleItem extends Item {
    private double discount; // as a decimal

    public SaleItem(String name, String sku, double price,
       double discount)  {
       super(name, sku, price);
       this.discount = Math.max(0,
           Math.min(discount, 1.00));
    }
}
```

Important: When calling a superclass constructor, the call to `super` ***must*** be the first non-comment line in the subclass's constructor.

Here's the rest of the `SaleItem` code:

```
    public double getDiscount() {
        return this.discount;
    }

    public void setDiscount(double discount) {
        this.discount =
            Math.max(0, Math.min(discount, 1.00));
    }

    public double purchase(int quantity) {
        return (this.getPrice() * quantity) *
            (1 - discount);
    }

    public String toString() {
        return String.format("%s (%s): $%.2f - %.1f%% discount",
            this.getName(), this.getSku(), this.getPrice(),
            this.discount * 100.0);
    }

    public boolean equals(SaleItem other) {
        return (this.getName().equals(other.getName()) &&
            this.getSku().equals(other.getSku()) &&
            this.getPrice() == other.getPrice() &&
            this.discount == other.discount);
    }
}
```

Again, because `name`, `sku`, and `price` are `private` to the `Item` class, the `purchase`, `toString` and `equals` methods must use the getter methods to access those private values.

Here's a program in file *ch14/ItemTest.java* that creates an `Item` and a `SaleItem`, prints them, and then purchases ten of each:

```
public class ItemTest {

    public static void main(String[] args) {
        Item envelopes = new Item(
            "Letter Size Envelopes - 100 count",
            "LSE-0100", 5.75);
        SaleItem marker = new SaleItem(
            "Erasable marker - black", "EMB-913",
            2.15, 0.10);

        System.out.println(envelopes);
        double envelopeTotal = envelopes.purchase(10);
        System.out.printf("Ten boxes of envelopes cost $%.2f\n",
            envelopeTotal);

        System.out.println(marker);
        double markerTotal = marker.purchase(10);
        System.out.printf("Ten markers cost $%.2f\n",
            markerTotal);
    }
}
```

When we run the program, we get the correct output:

```
Letter Size Envelopes - 100 count (LSE-0100): $5.75
Ten boxes of envelopes cost $57.50
Erasable marker - black (EMB-913): $2.15 - 10.0% discount
Ten markers cost $19.35
```

Congratulations! We've used inheritance, and our program works. Now it's time to do what Joe Armstrong[2]said in his book *Erlang and OTP in Action*: "Make it work, then make it beautiful, then if you really, really have to, make it fast. 90 percent of the time, if you make it beautiful, it will already be fast. So really, just make it beautiful!"

Part of making a program beautiful is getting rid of unnecessary duplication. In that spirit, let's take a closer look at the `SaleItem` class's `purchase`,

[2]Joe Armstrong was the co-inventor of the Erlang programming language. It's totally different from Java, and it's an incredibly interesting language.

`toString` and `equals` methods. The calculation of the base price in `purchase` is the same as in `Item`. The part of `toString`'s format string for the name, sku, and price is the same as in `Item`. Similarly, the first three lines of `equals` are the same as in `Item`.

We can once again use the `super` keyword to call methods in a parent class. Here's a rewrite that eliminates unnecessary duplication:

```
public double purchase(int quantity) {
    return super.purchase(quantity) *
        (1 - discount / 100.0);
}

public String toString() {
    return String.format("%s - %.1f%% discount",
        super.toString(),
        this.discount * 100.0);
}

public boolean equals(SaleItem other) {
    return (super.equals(other) &&
        this.discount == other.discount);
}
```

14.5 Exercises (continued)

Exercise 14.2 Design a class named `Person` with two subclasses: `Employee` and `Customer`. The attributes for these classes are described in italics. A `Person` has a *name*, *address*, *phone number*, and *email address*.

An `Employee` has an *employee number*, *hire date*, and *salary*.

The `Employee` class, in turn, has three subclasses: `Programmer`, `Tester`, and `Manager`.

- A `Programmer` and a `Tester` have a *cubicle number*. Both will receive a *fixed bonus* at the end of the year.

- A `Manager` has an *office number* and has a variable bonus based on the performance of their team. This means that a `Manager` should have attributes for the *target bonus amount* and the *performance percentage*.

Finally, the `Customer` should have a *customer number* and *company* they work for.

Draw the UML diagram showing the relationship of these classes, then code all these classes showing the data fields and attributes. Make meaningful names for the attributes and give them an appropriate data type. (You do not need to create constructors or other methods for the classes.)

Exercise 14.3 The XYZZY Corporation wants to retain their most loyal customers. They launch a customer retention program and offer discount to customers who have been purchasing from the company for at least one year.

Write a subclass `PrefCust` that extends the `Customer` class from the preceding exercise. The `PrefCust` class should have two data fields: *purchase amount* and *customer history* (number of years they have been a customer). These are both private variables.

Customers get a discount percentage based on their history and purchase amount. There are three levels of Preferred Customers: bronze, silver and gold.

- Bronze: history $\geq$ 1 year and average purchase amount $\geq$ \$5000 per year. 5
- Silver: history $\geq$ 2 years and average purchase amount $\geq$ \$10000 per year. 7.5
- Gold: history $\geq$ 3 years and average purchase amount $\geq$ \$15000 per year. 10

The discount percentage is a *derived attribute*—it is never set directly, but instead is computed based on other attributes.

Write the `PrefCust` class with all its data fields. Please write all the getter and setter methods. Write a method named `getDiscount` that uses the purchase amount and customer history to return the discount percent (as a percentage).

Exercise 14.4 In this exercise, you will implement an `Account` class which represents a bank checking account. You will then create two classes that inherit from `Account`: `SavingsAccount` and `CreditCardAccount`.

You will then use composition to create a `Customer` class which includes instances of each of these account classes.

Finally, you will write a program with a `main` method that works with these classes.

Part 1: Create a class named `Account`, which has the following private properties:

- `number: long`
- `balance: double`

1. Create a two-parameter constructor that takes an account number and balance. Make sure that the balance is always greater than zero (*Hint*: `Math.abs`)
2. Implement getters and setters: `getNumber()`, `getBalance()`, `setBalance(double newBalanc` There is no `setNumber` method—once an account is created, its account number cannot change.
3. Implement these methods: `void deposit(double amount)` and `void withdraw(double amo` For both these methods, if the amount is less than zero, the account balance remains untouched. For the `withdraw` method, if the amount is greater than the balance, it remains untouched. *These methods do not print anything.*
4. Implement a `toString` method that returns a string with the account number and balance, properly labeled.

Part 2: Next, implement the `SavingsAccount` class. It inherits from `Account` and adds a private `apr` property, which is the annual percentage rate (APR) for interest.

1. Write a three-argument constructor that takes an account number, balance, and interest rate as a decimal (thus, a 3.5% interest rate is given as 0.035). Make sure that the interest rate is never less than zero.

2. add a getter and setter: `getApr()` and `setApr(double apr)`. The setter must ensure that the APR is never less than zero.

3. Write a `calculateInterest` instance method that returns the annual interest, calculated as the current balance times the annual interest rate.

4. Modify `toString` to include the interest rate. IMPORTANT: The value returned by the `toString` method must ***not*** include the calculated annual interest.

Part 3: Next, implement the `CreditCardAccount` class, which inherits from `Account` and adds these `private` properties:

- `apr`, a `double` representing the annual interest rate charged on the balance.
- `creditLimit`, a `double` which gives the credit limit for the card.

Then, implement:

1. A four-argument constructor that takes an account number, balance, interest rate as a decimal (thus, a 3.5% interest rate is given as 0.035), and credit limit. Make sure that neither the interest rate nor credit limit can be negative.

2. Write getters and setters for the `apr` and `creditLimit`. The apr setter should leave the APR untouched if given a negative value. The `creditLimit` setter should leave the credit limit untouched if given a negative value.

3. Modify `toString` to include the interest rate and credit limit. IMPORTANT: the value returned by the `toString` method must ***not*** include the monthly payment.

4. Override the `withdraw` method so that you can have a negative balance. If a withdrawal would push you over the credit limit, leave the balance untouched. Examples:
   - If your balance is $300 with a credit limit of $700, you can withdraw $900 (leaving a balance of $-600).

- If your balance is $-300 with a credit limit of $700, you can withdraw $350 (leaving a balance of $-650).
- If your balance is $-300 with a credit limit of $700, you can not withdraw $500, because that would then owe $800, which is more than your $700 limit.

In short, the maximum amount you can withdraw is your current balance plus the credit limit.

5. Implement a `calculatePayment` method that works as follows: If the balance is positive, the minimum amount you have to pay on your card per month is zero. Otherwise, your monthly payment is the minimum of 20 and $(apr/12) \cdot (-balance)$

Part 4: Now, write a `Customer` class that will use composition to include the preceding classes.

1. The `Customer` class has the following private attributes:
   - `name: String`
   - `acct: Account`
   - `savings: SavingsAccount`
   - `credit: CreditAccount`
2. Implement a four-argument constructor for this class.
3. Write getters and setters for all the fields.

Figure 14.7 shows the details of the `Account` class and its subclasses. Figure 14.8 shows the relationships of all the classes in this exercise.

Part 5: Finally, write a program named *TestCustomer.java* that creates a `Customer` named "Joe Doakes" with this data:

- Regular account number 1037825 with a balance of $3,723.00
- Savings account number 9016632 with a balance of $4,810.25 and an annual interest rate of 2.05%
- Checking account number 85332162 with a balance of -$2500.00, an interest rate of 7.125%, and a credit limit of $3000.00.

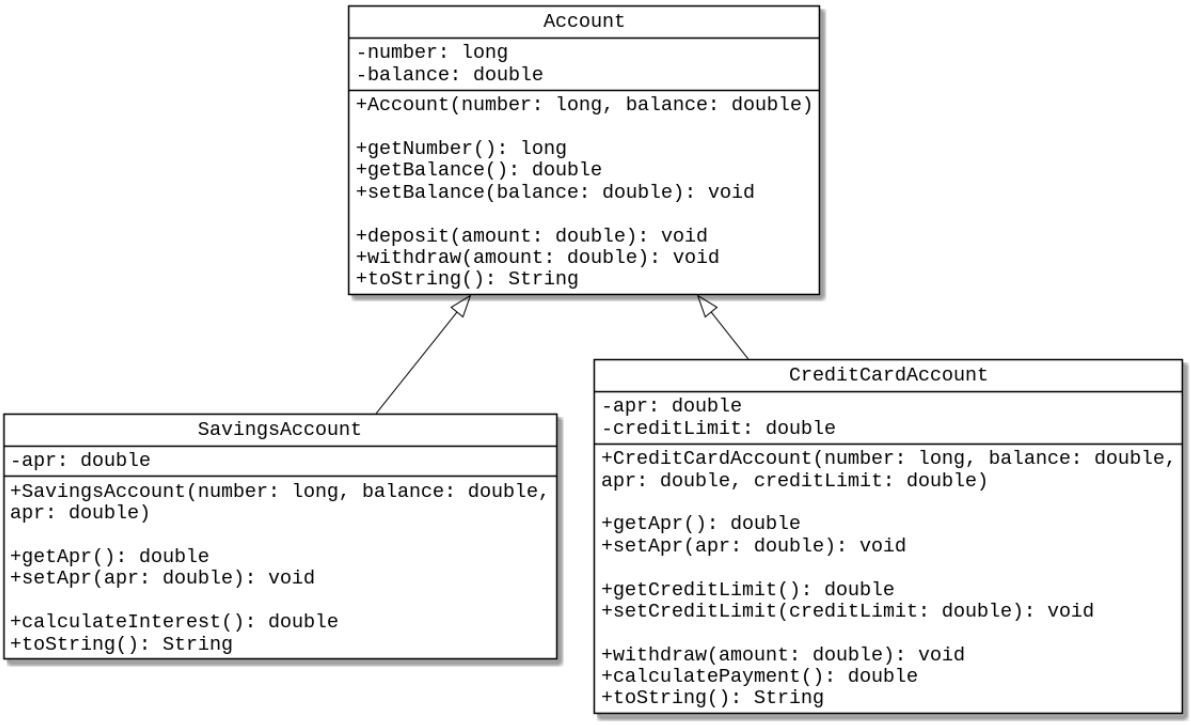

Figure 14.7: `Account`, `SavingsAccount`, and `CreditCardAccount` classes

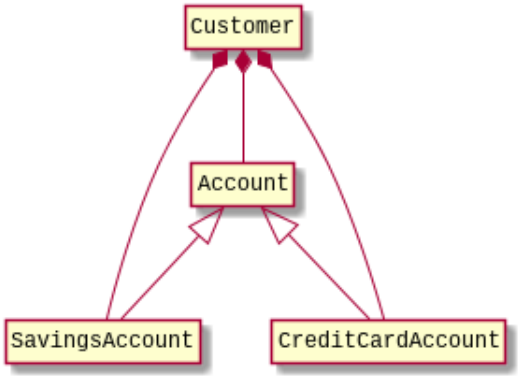

Figure 14.8: Composition and Inheritance Among All Classes

Then, do the following transactions:

- Deposit $257.18 into the regular account, then withdraw $587.23.
- Deposit $2,466.12 into the savings account, then withdraw $8,000.00.

- Withdraw $480.00 from the credit card account.
- Display the status of the regular account (number and balance).
- Display the status of the savings account (number, balance, and annual interest amount).
- Display the status of the credit card account (number, balance, interest rate, and monthly payment due).

14.6 Polymorphism and Dynamic Binding

Let's return to the `Item` and `SaleItem` classes. Try the following program (in the repository in file *ch14/Polymorphism.java*) and see what happens:

```
public class Polymorphism {
    public static void main(String[] args) {
        Item item1 = new Item("Cat Food", "CF-909", 7.85);
        Item item2 = new SaleItem("Lima Beans", "LB-104",
            5.95, 7.5);

        System.out.println("item 1: " + item1.toString());
        System.out.println("item 2: " + item2.toString());
    }
}
```

The first assignment statement for `item1` is no surprise; it assigns an `Item` object to an `Item` variable. But that second assignment statement—assigning a `SaleItem` object to an `Item` variable? How can that possibly work? What mad sorcery is this?

Remember, we're talking about an *is-a* relationship. A `SaleItem` *is an* `Item`. That's why it's legal to make the assignment. That is called *polyorphism*—the ability to assign a child class object to a parent class variable.

Note that you can't assign a parent object to a child variable. Every `SaleItem` is an `Item`, but not every `Item` is a `SaleItem`. If you try this:

```
SaleItem badNews = new Item("Oops", "X-000", 6.66);
```

The compiler will (correctly) complain:

```
Polymorphism.java:7: error: incompatible types:
  Item cannot be converted to SaleItem
        SaleItem badNews = new Item("Oops", "X-001", 6.66);
```

If you compile and run this program (without the bad line in it), you get this output:

```
item 1: Cat Food (CF-909): $7.85
item 2: Lima Beans (LB-104): $5.95 - 7.5% discount
```

Take a look at the `System.out.println` statements that produced that output. The first one uses `Item`'s `toString` method to print `item1` with its name, SKU, and price. The second one uses `SaleItem`'s `toString` method to print `item2`'s name, SKU, price *and* discount rate.

But both `item1` and `item2` are `Item` objects, so how did the second `println` know to call the `toString` method from `SaleItem`?

The answer is *dynamic binding*. At compile time, both `item1` and `item2` have the `Item` data type. But at run time, the JVM looks at the actual object that `item2` refers to and finds that it has the `SaleItem` data type, and it uses `SaleItem`'s `toString` method.

This difference between what the compiler sees and what the runtime looks at has very important consequences.

Consider this code, which sets up an array of `Item`. Polymorphism allows some of them to be `Item` instances and others to be `SaleItem` instances:

```
public class PolyArray {
    public static void main(String[] args) {
        Item [] foods = {
            new Item("Rye Bread", "RB-010", 3.95),
            new SaleItem("Tomato Soup", "TS-882", 1.29, 0.05),
            new SaleItem("Canned Lima Beans", "CLB-104",
                2.98, 0.155),
            new SaleItem("Frozen Pizza", "FP-326",
                5.90, 0.12),
            new Item("Organic Salsa", "OS-245", 3.79)
        };
// ...
    }
}
```

We want the program to go through the array and print each item's name and price, and, if it's a sale item, say how much the customer saves:

```
Rye Bread: $3.95
Tomato Soup: $1.23 - you save $0.06
Canned Lima Beans: $2.52 - you save $0.46
Frozen Pizza: $5.19 - you save $0.71
Organic Salsa: $3.79
```

Here's the pseudo-code for what we want to do:

```
for (Item food: foods) {
    System.out.printf("%s: ", food.getName());
    if (food is a sale item) {
        discounted price is getPrice() * (1 - getDiscount());
        amount saved is getPrice() * getDiscount();
        display discounted price and amount saved;
    } else {
        display getPrice();
    }
}
```

Note that the `for` loop variable is an `Item`, because we have told the compiler that `foods` is an array of `Item`. Here's the big question: how do we determine—at run time—if an array element is a `SaleItem` or an `Item`? We use the `instanceof` operator, which takes the form:

```
variable instanceof Class
```

and returns `true` if the given `variable` is an instance of `Class`; `false` otherwise.

A first try at the code looks like this:

```
for (Item food: foods) {
    System.out.printf("%s: ", food.getName());
    if (food instanceof SaleItem) {
        double amountSaved = food.getPrice() *
            food.getDiscount();
        System.out.printf("$%.2f - you saved $%.2f\n",
            food.getPrice() - amountSaved, amountSaved);
    } else {
        System.out.printf("$%.2f\n",
            food.getPrice());
    }
}
```

But when we compile, the compiler tells us:

```
PolyArray.java:17: error: cannot find symbol
            food.getDiscount();
                ^
  symbol:   method getDiscount()
  location: variable food of type Item
```

Why can't the compiler find `getDiscount`? The answer is in variable food **of type Item**. As far as the compiler is concerned, `food` is an `Item`, and that class does not have a `getDiscount` method in it.

What we have to do is use a *cast* to tell the compiler, "yes, we declared it as an `Item`, but please treat it as a `SaleItem`":

```
for (Item food: foods) {
    System.out.printf("%s: ", food.getName());
    if (food instanceof SaleItem) {
        // tell the compiler to treat food as a SaleItem
        SaleItem saleFood = (SaleItem) food;
        double amountSaved = saleFood.getPrice() *
            saleFood.getDiscount();
        System.out.printf("$%.2f - you saved $%.2f\n",
            saleFood.getPrice() - amountSaved, amountSaved);
    } else {
        System.out.printf("$%.2f\n",
            food.getPrice());
    }
}
```

Now the compiler is happy—`saleFood` is a `SaleItem`, and that class has a `getDiscount` method.

14.7 Summary

Here's a quick summary of inheritance, polymorphism, and dynamic binding.

- A subclass (the child class) extends a superclass (the parent class).
- A subclass constructor can call the superclass constructor by invoking the `super` method.
- If you use a `super` constructor, it *must* be the first non-comment line.
- Subclass methods can invoke the superclass methods by using `super.method` anywhere in the subclass method body.
- Polymorphism allows you to assign a subclass object to a superclass variable. For example,

  ```
  Item myItem = new SaleItem("Tomato Soup", "TS-882",
     1.29, 0.05);
  ```

- At compile time, the compiler sees the superclass variable as having the superclass data type. In the preceding code, the compiler says that `myItem` has data type `Item`.

- At run time, the compiler uses the actual data type of the object. When you say:

  ```
  System.out.println(myItem.toString());
  ```

  the JVM will see that `myItem` contains a reference to a `SaleItem` object and will invoke `SaleItem`'s `toString` method.

- The compiler won't let you call a method that exists only in the subclass on a superclass variable. This won't work:

  ```
  double saving = myItem.getPrice() * myItem.getDiscount();
  ```

  `getPrice` is fine; that method belongs to `Item`, but `getDiscount` belongs only to the subclass.

- You can determine if a variable belongs to a class at run time by using the `instanceof` operator:

  ```
  if (myItem instanceof SaleItem) {
      ...
  }
  ```

- Once you have established that you have a variable of the subclass data type, you can convince the compiler to treat it as a subclass by using a cast:

  ```
  if (myItem instanceof SaleItem) {
      double saving = myItem.getPrice() *
          ((SaleItem) myItem).getDiscount();
  }
  ```

  The extra parentheses around the cast are required to get everything evaluated in the correct order.

14.8 Exercises (concluded)

Exercise 14.5 This exercise will let you practice polymorphism and dynamic binding. Implement the `Bicycle`, `ElectricBicycle`, and `CargoBicycle` classes. (These are not defined in the same way as in the preceding text.) The parent `Bicycle` class has these attributes and methods:

- `frameSize`, (in centimeters) a double, with a getter (but not a setter).

- `nGears`, an integer, with a getter (but not a setter).
- `currentGear`, an integer, with both a getter and setter.
- A constructor with two parameters for the frame size and number of gears.
- A `toString` method that includes frame size, number of gears, and current gear, properly labeled.

The `ElectricBicycle` class is a child of `Bicycle` and adds these attributes and methods:

- `batteryCapacity` (in watt-hours), an integer, with a getter (but not a setter).
- `currentCharge` (in watt-hours), a double, with a getter and setter.
- A constructor with three parameters for frame size, number of gears, and battery capacity.
- `chargePercent`, a method that returns the percentage charge in the battery as a decimal, by dividing current charge by battery capacity.
- A `toString` method that includes frame size, number of gears, current gear, battery capacity, and current charge, properly labeled.

The `CargoBicycle` class is also a child of `Bicycle` and adds these attributes and methods:

- `maxLoad` (in kilograms), a double, with a getter (but not a setter). This is the maximum load that the bicycle can carry.
- `currentLoad` (in kilograms), a double, with a getter and setter.
- A constructor with three parameters for frame size, number of gears, and maximum cargo load.
- `loadFactor`, a method that returns the percentage of load on the bicycle as a decimal, by dividing the current load by the maximum load.
- A `toString` method that includes frame size, number of gears, current gear, maximum load, and current load, properly labeled.

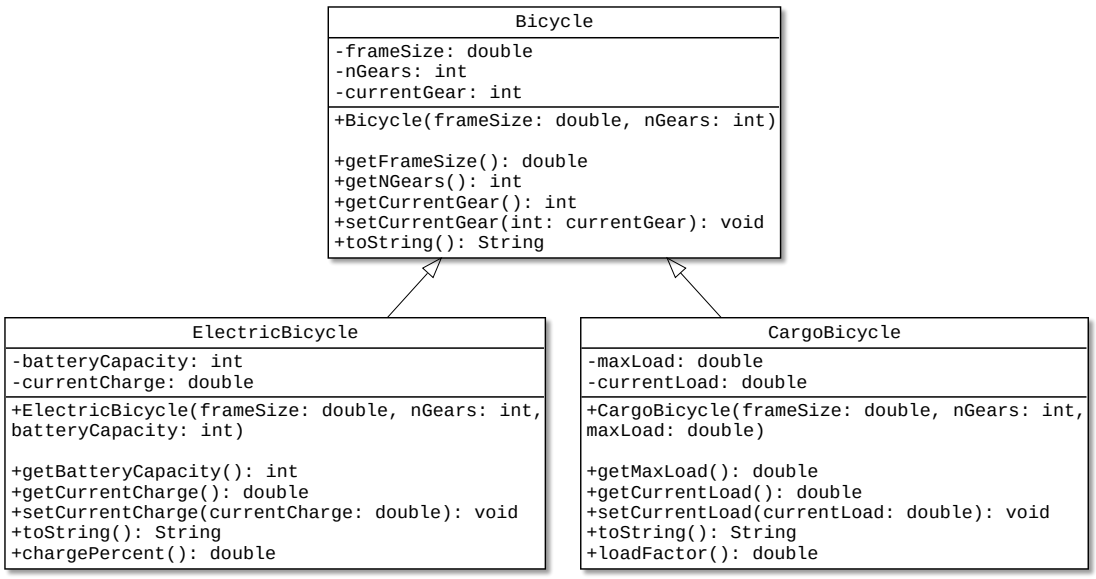

Figure 14.9: UML Diagram of Bicycle, ElectricBicycle, and CargoBicycle

Figure 14.9 is the UML diagram for the three classes.

Once you have implemented these classes, write a class named `BicycleTest` with a `main` method that does the following:

- Create an array of `Bicycle` with these bicycles:
  - A bicycle with a 55cm frame, one gear.
  - An electric bicycle with a 57cm frame, five gears, and a battery capacity of 500 wH (watt-hours).
  - A cargo bicycle with a 60cm frame, ten gears, and a maximum load of 35kg.
- Set the electric bicycle's current charge to 312.5 wH.
- Set the cargo bicycle's current load to 27.5 kg. - note: you may need to use a cast to do this step and the preceding step!
- Iterate through the array. For each bicycle, call the `toString` method and print the information it returns.

  - If the entry is an electric bicycle, also print out what percentage charge it has by calling the `chargePercent` method and printing its result.
  - If the entry is a cargo bicycle, also print out what percentage of the maximum load it is carrying by calling the `loadFactor` method and printing its result.

Exercise 14.6 This exercise will let you practice polymorphism and dynamic binding. It uses the `Account`, `SavingsAccount`, and `CreditCardAccount` classes you developed in Exercise 14.4. Instead of creating a customer with multiple accounts, create an array of these accounts:

- An `Account` number 1066 with a balance of $7,500.
- A `SavingsAccount` number 30507 with a balance of $4,500 and an APR of 1.5%
- A `CreditCardAccount` number 51782737 with a balance of $7,000.00, APR of 8%, and credit limit of $1000.00
- A `CreditCardAccount` number 629553328 with a balance of $1,500.00, an APR of 7.5%, and a credit limit of $5,000
- A `CreditCardAccount` number 4977201043L with a balance of -$5,000.00, an APR of 7%, and a credit limit of $10,000 (The `L` after the account number lets the compiler know that the account number is a `long` integer.)

Your program will use a loop to do the following for each account:

- Deposit $2,134.00
- Withdraw $4,782.00
- Print the account status using `toString()`.
  - For savings accounts, also display the annual interest
  - For credit card accounts, also display the monthly payment

Here's what the output might look like:

```
Account: 1066
Balance: $4852.00

Account: 30507
Balance: $1852.00
Interest Rate: 1.50%
Annual Interest: $27.78

Account: 51782737
Balance: $4352.00
Interest Rate: 8.00%
Credit Limit: $1000.00
Monthly Payment: $0.00

Account: 629553328
Balance: $-1148.00
Interest Rate: 7.50%
Credit Limit: $5000.00
Monthly Payment: $7.18

Account: 4977201043
Balance: $-7648.00
Interest Rate: 7.00%
Credit Limit: $10000.00
Monthly Payment: $20.00
```