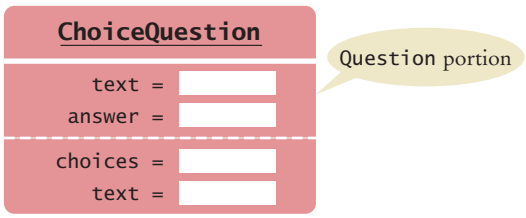
```
   private ArrayList<String> choices;
   private String text; // Don't!
   . . .
}
```

Sure, now the constructor compiles, but it doesn't set the correct text! Such a `ChoiceQuestion` object has two instance variables, both named `text`. The constructor sets one of them, and the `display` method displays the other.



Common Error 9.2

Confusing Super- and Subclasses

If you compare an object of type `ChoiceQuestion` with an object of type `Question`, you find that

- The reserved word `extends` suggests that the `ChoiceQuestion` object is an extended version of a `Question`.
- The `ChoiceQuestion` object is larger; it has an added instance variable, `choices`.
- The `ChoiceQuestion` object is more capable; it has an `addChoice` method.

It seems a superior object in every way. So why is `ChoiceQuestion` called the *subclass* and `Question` the *superclass*?

The *super/sub* terminology comes from set theory. Look at the set of all questions. Not all of them are `ChoiceQuestion` objects; some of them are other kinds of questions. Therefore, the set of `ChoiceQuestion` objects is a *subset* of the set of all `Question` objects, and the set of `Question` objects is a *superset* of the set of `ChoiceQuestion` objects. The more specialized objects in the subset have a richer state and more capabilities.

9.3 Overriding Methods

An overriding method can extend or replace the functionality of the superclass method.

The subclass inherits the methods from the superclass. If you are not satisfied with the behavior of an inherited method, you *override* it by specifying a new implementation in the subclass.

Consider the `display` method of the `ChoiceQuestion` class. It overrides the superclass `display` method in order to show the choices for the answer. This method *extends* the functionality of the superclass version. This means that the subclass method carries out the action of the superclass method (in our case, displaying the question text), and it also does some additional work (in our case, displaying the choices). In other cases, a subclass method *replaces* the functionality of a superclass method, implementing an entirely different behavior.

Let us turn to the implementation of the `display` method of the `ChoiceQuestion` class. The method needs to

- **Display the question text.**
- **Display the answer choices.**

The second part is easy because the answer choices are an instance variable of the subclass.

```
public class ChoiceQuestion
{
   . . .
   public void display()
   {
      // Display the question text
      . . .
      // Display the answer choices
      for (int i = 0; i < choices.size(); i++)
      {
         int choiceNumber = i + 1;
         System.out.println(choiceNumber + ": " + choices.get(i));
      }
   }
}
```

But how do you get the question text? You can't access the `text` variable of the superclass directly because it is private.

Use the reserved word `super` to call a superclass method.

Instead, you can call the `display` method of the superclass, by using the reserved word `super`:

```
public void display()
{
   // Display the question text
   super.display(); // OK
   // Display the answer choices
   . . .
}
```

If you omit the reserved word `super`, then the method will not work as intended.

```
public void display()
{
   // Display the question text
   display(); // Error—invokes this.display()
   . . .
}
```

Because the implicit parameter `this` is of type `ChoiceQuestion`, and there is a method named `display` in the `ChoiceQuestion` class, that method will be called—but that is just the method you are currently writing! The method would call itself over and over.

Here is the complete program that lets you take a quiz consisting of two `ChoiceQuestion` objects. We construct both objects and pass them to a method `presentQuestion`. That method displays the question to the user and checks whether the user response is correct.

section_3/QuestionDemo2.java

```
1  import java.util.Scanner;
2
3  /**
4     This program shows a simple quiz with two choice questions.
5  */
6  public class QuestionDemo2
7  {
8     public static void main(String[] args)
9     {
```

```
10       ChoiceQuestion first = new ChoiceQuestion();
11       first.setText("What was the original name of the Java language?");
12       first.addChoice("*7", false);
13       first.addChoice("Duke", false);
14       first.addChoice("Oak", true);
15       first.addChoice("Gosling", false);
16
17       ChoiceQuestion second = new ChoiceQuestion();
18       second.setText("In which country was the inventor of Java born?");
19       second.addChoice("Australia", false);
20       second.addChoice("Canada", true);
21       second.addChoice("Denmark", false);
22       second.addChoice("United States", false);
23
24       presentQuestion(first);
25       presentQuestion(second);
26    }
27
28    /**
29       Presents a question to the user and checks the response.
30       @param q the question
31    */
32    public static void presentQuestion(ChoiceQuestion q)
33    {
34       q.display();
35       System.out.print("Your answer: ");
36       Scanner in = new Scanner(System.in);
37       String response = in.nextLine();
38       System.out.println(q.checkAnswer(response));
39    }
40 }
```

section_3/ChoiceQuestion.java

```
1  import java.util.ArrayList;
2
3  /**
4     A question with multiple choices.
5  */
6  public class ChoiceQuestion extends Question
7  {
8     private ArrayList<String> choices;
9
10    /**
11       Constructs a choice question with no choices.
12    */
13    public ChoiceQuestion()
14    {
15       choices = new ArrayList<String>();
16    }
17
18    /**
19       Adds an answer choice to this question.
20       @param choice the choice to add
21       @param correct true if this is the correct choice, false otherwise
22    */
23    public void addChoice(String choice, boolean correct)
24    {
```

```
25          choices.add(choice);
26          if (correct)
27          {
28             // Convert choices.size() to string
29             String choiceString = "" + choices.size();
30             setAnswer(choiceString);
31          }
32       }
33
34       public void display()
35       {
36          // Display the question text
37          super.display();
38          // Display the answer choices
39          for (int i = 0; i < choices.size(); i++)
40          {
41             int choiceNumber = i + 1;
42             System.out.println(choiceNumber + ": " + choices.get(i));
43          }
44       }
45    }
```

Program Run

```
What was the original name of the Java language?
1: *7
2: Duke
3: Oak
4: Gosling
Your answer: *7
false
In which country was the inventor of Java born?
1: Australia
2: Canada
3: Denmark
4: United States
Your answer: 2
true
```

11. What is wrong with the following implementation of the `display` method?

```
public class ChoiceQuestion
{
   . . .
   public void display()
   {
      System.out.println(text);
      for (int i = 0; i < choices.size(); i++)
      {
         int choiceNumber = i + 1;
         System.out.println(choiceNumber + ": " + choices.get(i));
      }
   }
}
```

12. What is wrong with the following implementation of the `display` method?

```
public class ChoiceQuestion
{
```

```
   . . .
   public void display()
   {
      this.display();
      for (int i = 0; i < choices.size(); i++)
      {
         int choiceNumber = i + 1;
         System.out.println(choiceNumber + ": " + choices.get(i));
      }
   }
}
```

13. Look again at the implementation of the `addChoice` method that calls the `setAnswer` method of the superclass. Why don't you need to call `super.setAnswer`?
14. In the `Manager` class of Self Check 7, override the `getName` method so that managers have a `*` before their name (such as `*Lin, Sally`).
15. In the `Manager` class of Self Check 9, override the `getSalary` method so that it returns the sum of the salary and the bonus.

Practice It Now you can try these exercises at the end of the chapter: P9.1, P9.2, P9.11.

Common Error 9.3

Accidental Overloading

In Java, two methods can have the same name, provided they differ in their parameter types. For example, the `PrintStream` class has methods called `println` with headers

```
void println(int x)
```

and

```
void println(String x)
```

These are different methods, each with its own implementation. The Java compiler considers them to be completely unrelated. We say that the `println` name is **overloaded**. This is different from overriding, where a subclass method provides an implementation of a method whose parameter variables have the *same* types.

If you mean to override a method but use a parameter variable with a different type, then you accidentally introduce an overloaded method. For example,

```
public class ChoiceQuestion extends Question
{
   . . .
   public void display(PrintStream out)
   // Does not override void display()
   {
      . . .
   }
}
```

The compiler will not complain. It thinks that you want to provide a method just for `PrintStream` arguments, while inheriting another method `void display()`.

When overriding a method, be sure to check that the types of the parameter variables match exactly.

Common Error 9.4

Forgetting to Use super When Invoking a Superclass Method

A common error in extending the functionality of a superclass method is to forget the reserved word `super`. For example, to compute the salary of a manager, get the salary of the underlying `Employee` object and add a bonus:

```
public class Manager
{
   . . .
   public double getSalary()
   {
      double baseSalary = getSalary();
         // Error: should be super.getSalary()
      return baseSalary + bonus;
   }
}
```

Here `getSalary()` refers to the `getSalary` method applied to the implicit parameter of the method. The implicit parameter is of type `Manager`, and there is a `getSalary` method in the `Manager` class. Calling that method is a recursive call, which will never stop. Instead, you must tell the compiler to invoke the superclass method.

Whenever you call a superclass method from a subclass method with the same name, be sure to use the reserved word `super`.

Special Topic 9.1

Calling the Superclass Constructor

Consider the process of constructing a subclass object. A subclass constructor can only initialize the instance variables of the subclass. But the superclass instance variables also need to be initialized. Unless you specify otherwise, the superclass instance variables are initialized with the constructor of the superclass that has no arguments.

Unless specified otherwise, the subclass constructor calls the superclass constructor with no arguments.

In order to specify another constructor, you use the `super` reserved word, together with the arguments of the superclass constructor, as the *first statement* of the subclass constructor.

For example, suppose the `Question` superclass had a constructor for setting the question text. Here is how a subclass constructor could call that superclass constructor:

To call a superclass constructor, use the `super` reserved word in the first statement of the subclass constructor.

```
public ChoiceQuestion(String questionText)
{
   super(questionText);
   choices = new ArrayList<String>();
}
```

In our example program, we used the superclass constructor with no arguments. However, if all superclass constructors have arguments, you must use the `super` syntax and provide the arguments for a superclass constructor.

The constructor of a subclass can pass arguments to a superclass constructor, using the reserved word `super`.

When the reserved word `super` is followed by a parenthesis, it indicates a call to the superclass constructor. When used in this way, the constructor call must be *the first statement of the subclass constructor*. If `super` is followed by a period and a method name, on the other hand, it indicates a call to a superclass method, as you saw in the preceding section. Such a call can be made anywhere in any subclass method.