IN THIS CHAPTER

- Defining constructors
- Inheriting constructors from a parent class
- Using Java's default constructor features
- Constructing a simple GUI from scratch

Chapter 9
Constructing New Objects

Ms. Jennie Burd

121 Schoolhouse Lane

Anywhere, Kansas

Dear Ms. Burd,

In response to your letter of June 21, I believe I can say with complete assurance that objects are not created spontaneously from nothing. Although I've never actually seen an object being created (and no one else in this office can claim to have seen an object in its moment of creation), I have every confidence that some process or another is responsible for the building of these interesting and useful thingamajigs. We here at ObjectsAndClasses.com support the unanimous opinions of both the scientific community and the private sector in matters of this nature. Furthermore, we agree with the recent finding of a blue ribbon panel that concludes, beyond any doubt, that spontaneous object creation would impede the present economic outlook.

Please be assured that I have taken all steps necessary to ensure the safety and well-being of you, our loyal customer. If you have any further questions, please do not hesitate to contact our complaint department. The department's manager is Mr. Blake Wholl. You can contact him by visiting our company's website.

Once again, let me thank you for your concern, and I hope you continue to patronize ObjectsAndClasses.com.

Yours truly,

Mr. Scott Brickenchicker

The one who couldn't get on the elevator in Chapter 4

Defining Constructors (What It Means to Be a Temperature)

Here's a statement that creates an object:

```
Account myAccount = new Account();
```

I know it works — I got it from one of my own examples in Chapter 7. Anyway, in Chapter 7 I say, "[W]hen Java executes `new Account()`, you're creating an object by calling the `Account` class's constructor." What does this pithy sentence mean?

Well, the keyword `new` tells Java to create an object — an instance of a class. Java responds by performing certain actions. For starters, Java finds a place in its memory to store information about the new object. If the object has fields, the fields should eventually have meaningful values.

CROSS REFERENCE

To find out about fields, see Chapter 7.

When you ask Java to create a new object, you may want to specify what's placed in the object's fields. And what if you're interested in doing more than filling fields? Perhaps, when Java creates a new object, you have a whole list of jobs for Java to carry out. For instance, when Java creates a new window object, you want Java to realign the sizes of all buttons in that window.

Creating a new object can involve all kinds of tasks, so in this chapter you create constructors. A *constructor* tells Java to perform a new object's start-up tasks.

What is a temperature?

"Good morning, and welcome to Object News. The local temperature in your area is a pleasant 73 degrees Fahrenheit."

Each temperature consists of two parts: a number and a temperature scale. A number is just a `double` value, such as 32.0 or 70.52. But what's a temperature scale? Is it a string of characters, like `"Fahrenheit"` or `"Celsius"`? Not really, because some strings aren't temperature scales. There's no `"Quelploof"` temperature scale, and a program that can display the temperature `"73 degrees Quelploof"` is a bad program. So how can you limit the temperature scales to the small number of scales that people use? One way to do it is with Java's `enum` type.

What is a temperature scale? (Java's enum type)

Java provides lots of ways for you to group things together. In Chapter 11, you group things to form an array. And in Chapter 12, you group things to form a collection. In this chapter, you group things into an `enum` type. (Of course, you can't group anything unless you can pronounce `enum`. The word *enum* is pronounced "ee-NOOM," like the first two syllables of the word *enumeration.*)

Creating a complicated `enum` type isn't easy, but to create a simple `enum` type, just write a bunch of words inside a pair of curly braces. Listing 9-1 defines an `enum` type. The name of the `enum` type is `TempScale`.

LISTING 9-1: The TempScale Type (an enum Type)

```
package com.example.weather;

public enum TempScale {
    CELSIUS, FAHRENHEIT, KELVIN, RANKINE,
    NEWTON, DELISLE, RÉAUMUR, RØMER, LEIDEN
}
```

WARNING

In Listing 9-1, I'm showing off my physics prowess by naming not two, not four, but *nine* different temperature scales. Some readers' computers have trouble with the special characters in the words `RÉAUMUR` and `RØMER`. If you're one of those readers, simply delete the words `RÉAUMUR` and `RØMER` from the code. I promise: It won't mess up the example.

When you define an `enum` type, two important things happen:

- **You create values.**

  Just as `13` and `151` are `int` values, `CELSIUS` and `FAHRENHEIT` are `TempScale` values.

- **You can create variables to refer to those values.**

  In Listing 9-2, I declare the fields `number` and `scale`. Just as

  ```
  double number;
  ```

  declares that a `number` variable is of type `double`,

  ```
  TempScale scale;
  ```

  declares variable `scale` to be of type `TempScale`.

  "To be of type `TempScale`" means that you can have values `CELSIUS`, `FAHRENHEIT`, `KELVIN`, and so on. So, in Listing 9-2, I can give the `scale` variable the value `FAHRENHEIT` (or `TempScale.FAHRENHEIT`, to be more precise).


TECHNICAL STUFF

An `enum` type is a Java class in disguise. That's why Listing 9-1 contains an entire file devoted to one thing — namely, the declaration of an `enum` type (the `TempScale` type). Like the declaration of a class, an `enum` type declaration belongs in a file all its own. The code in Listing 9-1 belongs in a file named `TempScale.java`.

Okay, so then what is a temperature?

Each temperature consists of two things: a number and a temperature scale. The code in Listing 9-2 makes this fact abundantly clear.

LISTING 9-2: The Temperature Class

```
package com.example.weather;

public class Temperature {
    private double number;

    private TempScale scale;

    public Temperature() {
        number = 0.0;
        scale = TempScale.FAHRENHEIT;
    }
```

```
    public Temperature(double number) {
        this.number = number;
        scale = TempScale.FAHRENHEIT;
    }

    public Temperature(TempScale scale) {
        number = 0.0;
        this.scale = scale;
    }

    public Temperature(double number, TempScale scale) {
        this.number = number;
        this.scale = scale;
    }

    public void setNumber(double number) {
        this.number = number;
    }

    public double getNumber() {
        return number;
    }

    public void setScale(TempScale scale) {
        this.scale = scale;
    }

    public TempScale getScale() {
        return scale;
    }
}
```

The code in Listing 9-2 has the usual setter and getter methods (accessor methods for the `number` and `scale` fields).

CROSS REFERENCE

For some good reading on setter and getter methods (also known as accessor methods), see Chapter 7.

On top of all of that, Listing 9-2 has four other method-like-looking things. Each of these method-like things has the name `Temperature`, which happens to be the same as the name of the class. None of these `Temperature` method-like things has a `return` type of any kind — not even `void`, which is the cop-out `return` type.

Each of these method-like things is called a constructor. A *constructor* is like a method, except that a constructor has a special purpose: to create new objects.

REMEMBER

Whenever the computer creates a new object, the computer executes the statements inside a constructor.

You can omit the word `public` in the first lines of Listings 9-1 and 9-2. If you omit `public`, other Java programs might not be able to use the features defined in the `TempScale` type and in the `Temperature` class. (Don't worry about the programs in this chapter: With or without the word `public`, all programs in this chapter can use the code in Listings 9-1 and 9-2. To find out which Java programs can use classes that aren't public, see Chapter 14.) If you *do* use the word `public` in the first line of Listing 9-1, Listing 9-1 *must* be in a file named `TempScale.java`, starting with a capital letter `T`. And if you *do* use the word `public` in the first line of Listing 9-2, Listing 9-2 *must* be in a file named `Temperature.java`, starting with a capital letter `T`. (For an introduction to public classes, see Chapter 7.)

What you can do with a temperature

Listing 9-3 gives form to some of the ideas that I describe in the preceding section. In Listing 9-3, you call the constructors that are declared earlier, in Listing 9-2. Figure 9-1 shows what happens when you run all this code.

```
70.00 degrees FAHRENHEIT
32.00 degrees FAHRENHEIT
 0.00 degrees CELSIUS
 2.73 degrees KELVIN
```

FIGURE 9-1: Running the code from Listing 9-3.

LISTING 9-3: Using the Temperature Class

```
package com.example.weather;

import static java.lang.System.out;

public class UseTemperature {

    public static void main(String[] args) {
        final String format = "%5.2f degrees %s\n";

        var temp = new Temperature();
        temp.setNumber(70.0);
        temp.setScale(TempScale.FAHRENHEIT);
        out.printf(format, temp.getNumber(), temp.getScale());

        temp = new Temperature(32.0);
        out.printf(format, temp.getNumber(), temp.getScale());
```

```
        temp = new Temperature(TempScale.CELSIUS);
        out.printf(format, temp.getNumber(), temp.getScale());

        temp = new Temperature(2.73, TempScale.KELVIN);
        out.printf(format, temp.getNumber(), temp.getScale());
    }
}
```

In Listing 9-3, each statement of the kind

```
temp = new Temperature(blah,blah,blah);
```

calls one of the constructors from Listing 9-2. So, by the time the code in Listing 9-3 finishes running, it creates four instances of the `Temperature` class. Each instance is created by calling a different constructor from Listing 9-2.

In Listing 9-3, the last of the four constructor calls has two parameters: `2.73` and `TempScale.KELVIN`. This isn't particular to constructor calls. A method call or a constructor call can have a bunch of parameters. You separate one parameter from another with a comma. Another name for "a bunch of parameters" is a *parameter list*.

The only rule you must follow is to match the parameters in the call with the parameters in the declaration. For example, in Listing 9-3, the fourth and last constructor call

```
new Temperature(2.73, TempScale.KELVIN)
```

has two parameters: the first of type `double` and the second of type `TempScale`. Java approves of this constructor call because Listing 9-2 contains a matching declaration. That is, the header

```
public Temperature(double number, TempScale scale)
```

has two parameters: the first of type `double` and the second of type `TempScale`. If a `Temperature` constructor call in Listing 9-3 had no matching declaration in Listing 9-2, Listing 9-3 would crash and burn. (To state things more politely, Java would display errors when you tried to compile the code in Listing 9-3.)

By the way, this business about multiple parameters isn't new. Over in Chapter 6, I write `keyboard.findWithinHorizon(".",0).charAt(0)`. In that line, the method call `findWithinHorizon(".",0)` has two parameters: a string and an `int` value. Luckily for me, the Java API has a method declaration for `findWithinHorizon` — a declaration whose first parameter is a string and whose second parameter is an `int` value.

HOW TO CHEAT: ENUM TYPES AND SWITCHES

Listings 9-2 and 9-3 contain long-winded names such as `TempScale.FAHRENHEIT` and `TempScale.CELSIUS`. Names such as `FAHRENHEIT` and `CELSIUS` belong to my `TempScale` type (the type defined in Listing 9-1). These names have no meaning outside of my `TempScale` context. (If you think I'm being egotistical with this "no meaning outside of my context" remark, try deleting the `TempScale.` part of `TempScale.FAHRENHEIT` in Listing 9-2. Suddenly, Java tells you that your code contains an error.)

Java is normally fussy about type names and dots. But when they created `enum` types, the makers of Java decided that `enum` types in `switch` statements and expressions deserved special treatment. You can use an `enum` value to decide which case to execute in a `switch` statement or switch expression. When you do this, you don't use the `enum` type name in the case expressions. For example, the following Java code is correct:

```
TempScale scale = TempScale.RANKINE;
char letter =
        switch (scale) {
            case CELSIUS -> 'C';
            case KELVIN -> 'K';
            case RANKINE, RÉAUMUR, RØMER -> 'R';
            default -> 'X';
        };
```

In the first line of code, I write `TempScale.RANKINE` because this first line isn't inside a `switch`. But in the next several lines of code, I write `case CELSIUS`, `case KELVIN`, and `case RANKINE` without the word `TempScale`. In fact, if I create a `case` clause by writing `case TempScale.RANKINE`, Java complains with a loud, obnoxious error message.

Constructing a temperature; a slow-motion replay

When the computer executes one of the `new Temperature` statements in Listing 9-3, the computer has to decide which of the constructors in Listing 9-2 to use. The computer decides by looking at the parameter list — the stuff in parentheses after the words `new Temperature`. For instance, when the computer executes

```
temp = new Temperature(32.0);
```

from Listing 9-3, the computer says to itself, "The number 32.0 in parentheses is a `double` value. One of the `Temperature` constructors in Listing 9-2 has just one parameter with type `double`. The constructor's header looks like this:

```
public Temperature(double number)
```

"So I guess I'll execute the statements inside that particular constructor." The computer goes on to execute the following statements:

```
this.number = number;
scale = TempScale.FAHRENHEIT;
```

As a result, you get a brand-new object whose `number` field has the value `32.0` and whose `scale` field has the value `TempScale.FAHRENHEIT`.

In the two lines of code, you have two statements that set values for the fields `number` and `scale`. Take a look at the second of these statements, which is a bit easier to understand. The second statement sets the new object's `scale` field to `TempScale.FAHRENHEIT`. You see, the constructor's parameter list is `(double number)`, and that list doesn't include a `scale` value. So whoever programmed this code had to make a decision about what value to use for the `scale` field. The programmer could have chosen `FAHRENHEIT` or `CELSIUS`, but they could also have chosen `KELVIN`, `RANKINE`, or any of the other obscure scales named in Listing 9-1. (This programmer happens to live in New Jersey, in the United States, where people commonly use the old Fahrenheit temperature scale.)

Marching back to the first of the two statements, this first statement assigns a value to the new object's `number` field. The statement uses a cute trick that you can see in many constructors (and in other methods that assign values to objects' fields). To understand the trick, take a look at Listing 9-4. The listing shows you two ways that I could have written the same constructor code.

LISTING 9-4: Two Ways to Accomplish the Same Thing

```
//Use this constructor...

    public Temperature(double whatever) {
        number = whatever;
        scale = TempScale.FAHRENHEIT;
    }

//... or use this constructor...

    public Temperature(double number) {
        this.number = number;
        scale = TempScale.FAHRENHEIT;
    }

//... but don't put both constructors in your code.
```

Listing 9-4 has two constructors in it. In the first constructor, I use two different names: `number` and `whatever`. In the second constructor, I don't need two names. Rather than make up a new name for the constructor's parameter, I reuse an existing name by writing `this.number`.

Here's what's going on in Listing 9-2:

- In the statement `this.number = number`, the name *this.number* refers to the new object's `number` field — the field that's declared near the top of Listing 9-2. (See Figure 9-2.)

  In the statement `this.number = number`, *number* (on its own, without `this`) refers to the constructor's parameter. (Again, see Figure 9-2.)

In general, `this.`*`someName`* refers to a field belonging to the object that contains the code. In contrast, plain old *`someName`* refers to the closest place where *`someName`* happens to be declared. In the statement `this.number = number` (refer to Listing 9-2), that closest place happens to be the `Temperature` constructor's parameter list.

WHAT'S THIS ALL ABOUT?

Suppose that your code contains a constructor — the first of the two constructors in Listing 9-4. The `whatever` parameter is passed a number like 32.0, for instance. Then the first statement in the constructor's body assigns that value, `32.0`, to the new object's `number` field. The code works. But in writing this code, you had to make up a new name for a parameter — the name *whatever.* And the only purpose for this new name is to hand a value to the object's `number` field. What a waste! To distinguish between the parameter and the `number` field, you gave a name to something that was just momentary storage for the `number` value.

Making up names is an art, not a science. I've gone through plenty of naming phases. Years ago, whenever I needed a new name for a parameter, I picked a confusing misspelling of the original variable name. (I'd name the parameter something like `numbr` or `nuhmber`.) I've also tried changing a variable name's capitalization to come up with a parameter name. (I'd use parameter names like `Number` or `nUMBER`.) In Chapter 8, I name all my parameters by adding the suffix *In* to their corresponding variable names. (The `jobTitle` variable matched up with the `jobTitleIn` parameter.) None of these naming schemes works well — I can never remember the quirky new names I've created. The good news is that this parameter-naming effort isn't necessary. You can give the parameter the same name as the variable. To distinguish between the two, you use the Java keyword `this`.

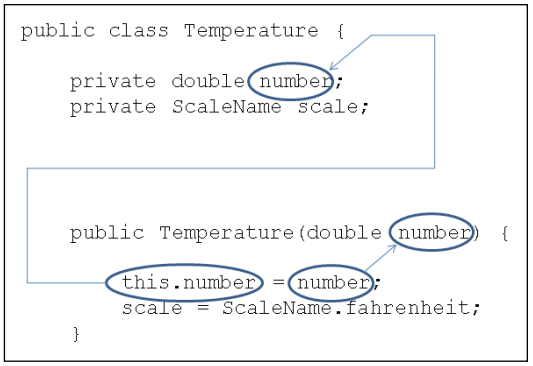

FIGURE 9-2: What `this.number` and `number` mean.

Some things never change

Chapter 7 introduces the `printf` method and explains that each `printf` call starts with a format string. The format string describes the way the other parameters are to be displayed.

In previous examples, this format string is always a quoted literal. For instance, the first `printf` call in Listing 7-7 (see Chapter 7) is

```
out.printf("$%4.2f\n", myInterest);
```

In Listing 9-3, I break with tradition and begin the `printf` call with a variable that I name *format:*

```
out.printf(format, temp.getNumber(), temp.getScale());
```

That's okay as long as my `format` variable is of type `String`. And indeed, in Listing 9-3, the first variable declaration is

```
final String format = "%5.2f degrees %s\n";
```

In this declaration of the `format` variable, take special note of the word `final`. This Java keyword indicates that the value of `format` can't be changed. If I add another assignment statement to Listing 9-3:

```
format = "%6.2f (%s)\n";
```

the compiler barks back at me with the message `cannot assign a value to final variable`.

When I write the code in Listing 9-3, the use of the `final` keyword isn't absolutely necessary. But the `final` keyword provides some extra protection. When I initialize `format` to `"%5.2f degrees %s\n"`, I intend to use this same format just as it is, over and over again. I know darn well that I don't intend to change the `format` variable's value. Of course, in a 10,000-line program, I can become confused and try to assign a new value to `format` somewhere deep down in the code. To prevent me from accidentally changing the `format` string, I declare the `format` variable to be final. It's just good, safe programming practice.

TRY IT OUT

There's always more stuff for you to try.

SCHOOL DAYS

Create a `Student` class with a name, an ID number, a grade point average (GPA), and a major area of study. The student's name is a `String`. The student's ID number is an `int` value. The GPA is a `double` value between 0.0 and 4.0. The `Major` is an `enum` type, with values such as `COMPUTER_SCIENCE`, `MATHEMATICS`, `LITERATURE`, `PHYSICS`, and `HISTORY`.

Every student has a name and an ID number, but a brand-new student might not have a GPA or a major. Create constructors with and without GPA and `Major` parameters.

As usual, create a separate class that makes use of your new `Student` class.

FLIGHT OF FANCY

Create an `AirplaneFlight` class with a flight number, a departure airport, the time of departure, an arrival airport, and a time of arrival. The flight number is an `int` value. The departure and arrival airport fields belong to an `Airport` enum type, with values corresponding to some of the official IATA airport codes. (For example, London Heathrow Airport's code is LHR; Los Angeles International Airport's code is LAX; check out `www.iata.org/publications/Pages/code-search.aspx` for a searchable database of airline codes.)

For the times of arrival and departure, use Java's `LocalTime` class. (For more on `LocalTime`, check out the `LocalTime` documents page at `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/time/LocalTime.html`.) To create a `LocalTime` object that's set to 2:15 P.M. (also known as 14:15), execute

```
LocalTime twoFifteen = LocalTime.of(14, 15);
```

To create a `LocalTime` object that's set to the current time (according to the computer's system clock), execute

```
LocalTime currentTime = LocalTime.now();
```

Every flight has a number, a departure airport, and an arrival airport. But some flights might not have departure and arrival times. Create constructors with and without departure and arrival time parameters.

Create a separate class that makes use of your new `AirplaneFlight` class.

MAKE A HIT RECORD

Newer versions of Java (from Java 16 onward) have a fancy feature called record classes. For an introduction to these beauties, name two files `TemperatureRecord.java` and `UseTemperatureRecord.java`. Put the following code in these files and then give the code a spin:

```
//TemperatureRecord.java

package com.example.weather;

public record TemperatureRecord(double number, TempScale scale) {
}

//UseTemperatureRecord.java

package com.example.weather;

import static java.lang.System.out;

public class UseTemperatureRecord {

    public static void main(String[] args) {
        final String format = "%5.2f degrees %s\n";

        TemperatureRecord temp = new TemperatureRecord(2.73, TempScale.KELVIN);
        out.printf(format, temp.number(), temp.scale());
    }
}
```

Doing Something about the Weather

In Chapter 8, I make a big fuss over the notion of subclasses. That's the right thing to do. Subclasses make code reusable, and reusable code is good code. With that in mind, it's time to create a subclass of the `Temperature` class (which I develop in this chapter's first section).

Building better temperatures

After perusing the code in Listing 9-3, you decide that the responsibility for displaying temperatures has been seriously misplaced. Listing 9-3 has several tedious repetitions of the lines to print temperature values. A 1970s programmer would tell you to collect those lines into one place and turn them into a method. (The 1970s programmer wouldn't have used the word *method,* but that's not important right now.) Collecting lines into methods is fine, but with today's object-oriented programming methodology, you think in broader terms. Why not get each `temperature` object to take responsibility for displaying itself? After all, if you develop a `display` method, you probably want to share the method with other people who use temperatures. So put the method right inside the declaration of a `temperature` object. That way, anyone who uses the code for temperatures has easy access to your `display` method.

Now replay the tape from Chapter 8. "Blah, blah, blah . . . don't want to modify existing code . . . blah, blah, blah . . . too costly to start again from scratch . . . blah, blah, blah . . . extend existing functionality." It all adds up to one thing:

> Don't abuse it. Instead, reuse it.

So you decide to create a subclass of the `Temperature` class — the class defined in Listing 9-2. Your new subclass complements the `Temperature` class's functionality by having methods to display values in a nice, uniform fashion. The new class, `TemperatureNice`, is shown in Listing 9-5.

LISTING 9-5: The TemperatureNice Class

```
package com.example.weather;

import static java.lang.System.out;

public class TemperatureNice extends Temperature {

    public TemperatureNice() {
        super();
    }
```

```
    public TemperatureNice(double number) {
        super(number);
    }

    public TemperatureNice(TempScale scale) {
        super(scale);
    }

    public TemperatureNice(double number, TempScale scale) {
        super(number, scale);
    }

    public void display() {
        out.printf("%5.2f degrees %s\n", getNumber(), getScale());
    }
}
```

In the `display` method of Listing 9-5, notice the calls to the `Temperature` class's `getNumber` and `getScale` methods. Why do I do this? Well, inside the `TemperatureNice` class's code, any direct references to the `number` and `scale` fields would generate error messages. It's true that every `TemperatureNice` object has its own `number` and `scale` fields. (After all, `TemperatureNice` is a subclass of the `Temperature` class, and the code for the `Temperature` class defines the `number` and `scale` fields.) But because number and scale are declared to be private inside the `Temperature` class, only code that's right inside the `Temperature` class can directly use these fields.

WARNING

Don't put additional declarations of the `number` and `scale` fields inside the `TemperatureNice` class's code. If you do, you inadvertently create four different variables (two called `number` and another two called `scale`). You'll assign values to one pair of variables. Then you'll be shocked that when you display the other pair of variables, those values seem to have disappeared.

REMEMBER

When an object's code contains a call to one of the object's own methods, you don't need to preface the call with a dot. For instance, in the last statement of Listing 9-5, the object calls its own methods with `getNumber()` and `getScale()`, not with *`someObject`*`.getNumber()` and `somethingOrOther.getScale()`. If going dotless makes you queasy, you can compensate by taking advantage of yet another use for the `this` keyword: Just write `this.getNumber()` and `this.getScale()` in the last line of Listing 9-5.

Constructors for subclasses

By far, the biggest news in Listing 9-5 is the way the code declares constructors. The `TemperatureNice` class has four of its own constructors. If you've gotten in gear thinking about subclass inheritance, you may wonder why these constructor declarations are necessary. Doesn't `TemperatureNice` inherit the parent `Temperature` class's constructors? No, subclasses don't inherit constructors.

REMEMBER

Subclasses don't inherit constructors.

That's right. Subclasses don't inherit constructors. In one oddball case, a constructor may look like it's being inherited, but that oddball situation is a fluke, not the norm. In general, when you define a subclass, you declare new constructors to go with the subclass.

I describe the oddball case (in which a constructor looks like it's being inherited) later in this chapter, in the section "The default constructor."

So the code in Listing 9-5 has four constructors. Each constructor has the name `TemperatureNice`, and each constructor has its own uniquely identifiable parameter list. That's the boring part. The interesting part is that each constructor makes a call to something named `super`, which is a Java keyword.

In Listing 9-5, `super` stands for a constructor in the parent class:

- The statement `super()` in Listing 9-5 calls the parameterless `Temperature()` constructor that's in Listing 9-2. That parameterless constructor assigns `0.0` to the `number` field and `TempScale.FAHRENHEIT` to the `scale` field.
- The statement `super(number, scale)` in Listing 9-5 calls the constructor `Temperature(double number, TempScale scale)` that's in Listing 9-2. In turn, the constructor assigns values to the `number` and `scale` fields.
- In a similar way, the statements `super(number)` and `super(scale)` in Listing 9-5 call constructors from Listing 9-2.

The computer decides which of the `Temperature` class's constructors is being called by looking at the parameter list after the word `super`. For instance, when the computer executes

```
super(number, scale);
```

from Listing 9-5, the computer says to itself, "The `number` and `scale` fields in parentheses have types `double` and `TempScale`. But only one of the `Temperature`

constructors in Listing 9-2 has two parameters with types `double` and `TempScale`. The constructor's header looks like this:

```
public Temperature(double number, TempScale scale)
```

"So I guess I'll execute the statements inside that particular constructor."

Using all this stuff

In Listing 9-5, I define what it means to be in the `TemperatureNice` class. Now it's time to put this `TemperatureNice` class to good use. Listing 9-6 has code that uses `TemperatureNice`.

LISTING 9-6: **Using the TemperatureNice Class**

```
package com.example.weather;

public class UseTemperatureNice {

    public static void main(String[] args) {

        var temp = new TemperatureNice();
        temp.setNumber(70.0);
        temp.setScale(TempScale.FAHRENHEIT);
        temp.display();

        temp = new TemperatureNice(32.0);
        temp.display();

        temp = new TemperatureNice(TempScale.CELSIUS);
        temp.display();

        temp = new TemperatureNice(2.73, TempScale.KELVIN);
        temp.display();
    }
}
```

The code in Listing 9-6 is much like its cousin code in Listing 9-3. The big differences are described here:

- Listing 9-6 creates instances of the `TemperatureNice` class. That is, Listing 9-6 calls constructors from the `TemperatureNice` class, not the `Temperature` class.

» Listing 9-6 takes advantage of the `display` method in the `TemperatureNice` class. So the code in Listing 9-6 is much tidier than its counterpart in Listing 9-3.

A run of Listing 9-6 looks exactly like a run of the code in Listing 9-3 — it just reaches the finish line in a far more elegant fashion. (The run is shown previously, in Figure 9-1.)

The default constructor

The main message in the previous section is that subclasses don't inherit constructors. So, what gives with all the listings over in Chapter 8? In Listing 8-6, a statement says

```
FullTimeEmployee ftEmployee = new FullTimeEmployee();
```

But here's the problem: The code defining `FullTimeEmployee` (refer to Listing 8-3) doesn't seem to have any constructors declared inside it. So, in Listing 8-6, how can you possibly call the `FullTimeEmployee` constructor?

Here's what's going on. When you create a subclass and don't put any explicit constructor declarations in your code, Java creates one constructor for you. It's called a *default constructor.* If you're creating the `public FullTimeEmployee` subclass, the default constructor looks like the one in Listing 9-7.

LISTING 9-7: A Default Constructor

```
public FullTimeEmployee() {
    super();
}
```

The constructor in Listing 9-7 takes no parameters, and its single statement calls the constructor of whatever class you're extending. (Woe be to you if the class you're extending has no parameterless constructor.)

You've just read about default constructors, but watch out! Notice one thing that this talk about default constructors *doesn't* say: It doesn't say that you always get a default constructor. In particular, if you create a subclass and define any constructors yourself, Java doesn't add a default constructor for the subclass (and the subclass doesn't inherit any constructors, either).

How can this trip you up? Listing 9-8 has a copy of the code from Listing 8-3, but with one constructor added to it. Take a look at this modified version of the `FullTimeEmployee` code.

LISTING 9-8: Look, I Have a Constructor!

```
package com.example.payroll;

public class FullTimeEmployee extends Employee {
    private double weeklySalary;
    private double benefitDeduction;

    public FullTimeEmployee(double weeklySalary) {
        this.weeklySalary = weeklySalary;
    }

    public void setWeeklySalary(double weeklySalaryIn) {
        weeklySalary = weeklySalaryIn;
    }

    public double getWeeklySalary() {
        return weeklySalary;
    }

    public void setBenefitDeduction(double benefitDedIn) {
        benefitDeduction = benefitDedIn;
    }

    public double getBenefitDeduction() {
        return benefitDeduction;
    }

    public double findPaymentAmount() {
        return weeklySalary - benefitDeduction;
    }
}
```

If you use the `FullTimeEmployee` code in Listing 9-8, a line like the following doesn't work:

```
FullTimeEmployee ftEmployee = new FullTimeEmployee();
```

It doesn't work because, having declared a `FullTimeEmployee` constructor that takes one `double` parameter, you no longer get a default parameterless constructor for free.

What do you do about this? If you declare any constructors, declare all constructors that you'll possibly need. Take the constructor in Listing 9-7 and add it to the code in Listing 9-8. Then the call `new FullTimeEmployee()` starts working again.

Under certain circumstances, Java automatically adds an invisible call to a parent class's constructor at the top of a constructor body. This automatic addition of a `super` call is a tricky bit of business that doesn't appear often, so when it does appear, it may seem quite mysterious. For more information, see this book's website (`http://javafordummies.allmycode.com`).

TRY IT OUT

In this section, I have four (count 'em — *four*) experiments for you to try:

STUDENT SHOWCASE

In a previous section, you create your own `Student` class. Create a subclass that has a method named `getString`.

Like the `display` method in this chapter's `TemperatureNice` class, the `getString` method creates a nice-looking `String` representation of its object. But unlike the `TemperatureNice` class's `display` method, the `getString` method doesn't print that `String` representation on the screen. Instead, the `getString` method simply returns that `String` representation as its result.

In a way, a `getString` method is much more versatile than a `display` method. With a `display` method, all you can do is show a `String` representation on the screen. But with a `getString` method, you can create a `String` representation and then do whatever you want with it.

Create a separate class that creates some instances of your new subclass and puts their `getString` methods to good use.

THE WAITING GAME

In a previous section, you create your own `AirplaneFlight` class. Create a subclass that has a method named `duration`. The `duration` method, which has no parameters, returns the amount of time between the flight's departure time and arrival time.

To find the number of hours between two `LocalTime` objects (such as `twoFifteen` and `currentTime`), execute

```
long hours = ChronoUnit.HOURS.between(twoFifteen, currentTime);
```

To find the number of minutes between two `LocalTime` objects (such as `twoFifteen` and `currentTime`), execute

```
long minutes = ChronoUnit.MINUTES.between(twoFifteen, currentTime);
```

A CONVERSION DIVERSION

Create a new `TemperatureEvenNicer` class — a subclass of this section's `TemperatureNice` class. The `TemperatureEvenNicer` class has a `convertTo` method. If the variable `temp` refers to a Fahrenheit temperature and Java executes

```
temp.convertTo(TempScale.CELSIUS);
```

then the `temp` object changes to a Celsius temperature, with the number converted appropriately. The same kind of thing happens if Java executes

```
temp.convertTo(TempScale.FAHRENHEIT);
```

with `temp` already referring to a Celsius temperature.

SET A NEW RECORD

Follow up on the "Make a Hit Record" experiment from earlier in this chapter. Name two files `TemperatureNiceRecord.java` and `UseTemperatureNiceRecord.java`. Put the following code in these files and see how they run.

```
//TemperatureNiceRecord.java

package com.example.weather;

import static java.lang.System.out;

public record TemperatureNiceRecord(double number, TempScale scale) {

    public TemperatureNiceRecord() {
        this(0, TempScale.CELSIUS);
    }

    public void display() {
        out.printf("%5.2f degrees %s\n", number, scale);
    }
}

//UseTemperatureNiceRecord.java
```

```
package com.example.weather;

public class UseTemperatureNiceRecord {

    public static void main(String[] args) {
        var temp = new TemperatureNiceRecord();
        temp.display();

        temp = new TemperatureNiceRecord(2.73, TempScale.KELVIN);
        temp.display();
    }
}
```

A Constructor That Does More

Here's a quote from somewhere near the start of this chapter: "And what if you're interested in doing more than filling fields? Perhaps, when the computer creates a new object, you have a whole list of jobs for the computer to carry out." Okay, what-if?

This section's example has a constructor that does more than just assign values to fields. The example is in Listings 9-9 and 9-10. The result of running the example's code is shown in Figure 9-3.

FIGURE 9-3: Don't panic.

LISTING 9-9: **Defining a Frame**

```
package com.example.graphical;

import javax.swing.JButton;
import javax.swing.JFrame;
import java.awt.FlowLayout;

public class SimpleFrame extends JFrame {
```

```
    public SimpleFrame() {
        setTitle("Don't click the button!");
        setLayout(new FlowLayout());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        add(new JButton("Panic"));
        setSize(300, 100);
        setVisible(true);
    }
}
```

LISTING 9-10: Displaying a Frame

```
package com.example.graphical;

public class ShowAFrame {

    public static void main(String[] args) {
        new SimpleFrame();
    }
}
```

Like my `DummiesFrame` examples, the code in Listings 9-9 and 9-10 displays a window on the computer screen. But unlike my `DummiesFrame` examples, all the method calls in Listings 9-9 and 9-10 refer to methods in Java's standard API (application programming interface).

CROSS REFERENCE

To find my `DummiesFrame` examples, refer to Chapter 7.

The code in Listing 9-9 contains lots of names that are probably unfamiliar to you — names from Java's API. When I was first becoming acquainted with Java, I foolishly believed that knowing Java meant remembering all these names. Quite the contrary: These names are just carry-on baggage. The real Java is the way the language implements object-oriented concepts.

Anyway, Listing 9-10's main method has only one statement: a call to the constructor in the `SimpleFrame` class. Notice how the object that this call creates isn't even assigned to a variable. That's okay because the code doesn't need to refer to the object anywhere else.

Up in the `SimpleFrame` class, there's only one constructor declaration. Far from just setting variables' values, this constructor calls method after method from the Java API.

All the methods called in the `SimpleFrame` class's constructor come from the parent class, `JFrame`. The `JFrame` class lives in the `javax.swing` package. This package and another package, `java.awt`, have classes that help you put windows, images, drawings, and other gizmos on a computer screen. (In the `java.awt` package, the letters *awt* stand for *a*bstract *w*indowing *t*oolkit.)

For a little gossip about the notion of a Java package, see Chapters 7 and 14.

REMEMBER

In the Java API, what people normally call a *window* is an instance of the `javax.swing.JFrame` class.

Classes and methods from the Java API

Looking at Figure 9-3, you can probably tell that an instance of the `SimpleFrame` class doesn't do much. The frame has only one button, and when you click the button, nothing happens. I made the frame this way to keep the example from becoming too complicated. Even so, the code in Listing 9-9 uses several API classes and methods. The `setTitle`, `setLayout`, `setDefaultCloseOperation`, `add`, `setSize`, and `setVisible` methods all belong to the `javax.swing.JFrame` class. Here's a list of names used in the code:

- `setTitle`: Calling `setTitle` puts words on the frame's title bar. (The new `SimpleFrame` object is calling its own `setTitle` method.)
- `FlowLayout`: An instance of the `FlowLayout` class positions objects on the frame in a centered, typewriter fashion. Because the frame in Figure 9-3 has only one button on it, that button is centered near the top of the frame. If the frame had eight buttons, five of them may be lined up in a row across the top of the frame and the remaining three would be centered along a second row.
- `setLayout`: Calling `setLayout` puts the new `FlowLayout` object in charge of arranging components, such as buttons, on the frame. (The new `SimpleFrame` object is calling its own `setLayout` method.)
- `setDefaultCloseOperation`: Calling `setDefaultCloseOperation` tells Java what to do when you click the little × in the frame's upper right corner. (On a Mac, you click the little red circle in the frame's upper left corner.) Without this method call, the frame itself disappears, but the Java virtual machine (JVM) keeps running. To stop your program's run, you have to perform one more step. (You may have to look for a Terminate option in Eclipse, IntelliJ IDEA, or NetBeans.)

  Calling `setDefaultCloseOperation(EXIT_ON_CLOSE)` tells Java to shut itself down when you click the × in the frame's upper right corner. The alternatives to `EXIT_ON_CLOSE` are `HIDE_ON_CLOSE`, `DISPOSE_ON_CLOSE`, and, my

personal favorite, `DO_NOTHING_ON_CLOSE`. Use one of these alternatives when your program has more work to do after the user closes your frame.

- `JButton`: The `JButton` class lives in the `javax.swing` package. One of the class's constructors takes a `String` instance (such as `"Panic"`) for its parameter. Calling this constructor makes that `String` instance into the label on the face of the new button.
- `add`: The new `SimpleFrame` object calls its `add` method. Calling the `add` method places the button on the object's surface (in this case, the surface of the frame).
- `setSize`: The frame becomes 300 pixels wide and 100 pixels tall. (In the `javax.swing` package, whenever you specify two dimension numbers, the width number always comes before the height number.)
- `setVisible`: When it's first created, a new frame is invisible. But when the new frame calls `setVisible(true)`, the frame appears on your computer screen.

Live dangerously

Your IDE may warn you that the `SimpleFrame` in Listing 9-9 has no `serialVersionUID` field. "And what," you ask, "is a `serialVersionUID` field?" It's something having to do with storing a `JFrame` object — something you don't care about. Not having a `serialVersionUID` field generates a warning, not an error. So throw caution to the wind and ignore the warning.

If, for some reason, you can't ignore the warning, suppress the warning by adding the line `@SuppressWarnings("serial")` with no semicolon immediately above the `public class SimpleFrame` line. (Like `@Override` from Chapter 8, `@SuppressWarnings` is a Java annotation.)

If, for some other reason, you don't want to suppress the warning, add the statement `private static final long serialVersionUID = 1L;` immediately below the `public class SimpleFrame` line.

THE NULL HYPOTHESIS

TRY IT OUT

In JShell, type the following sequence of declarations and statements. What happens? Why?

```
jshell> import javax.swing.JFrame

jshell> JFrame frame
```

```
jshell> frame.setSize(100, 100)

jshell> frame = new JFrame()

jshell> frame.setSize(100, 100)

jshell> frame.setVisible(true)
```

WIDESPREAD PANIC

In Listing 9-9, change the statement

```
setLayout(new FlowLayout());
```

to

```
setLayout(new BorderLayout));
```

What difference does this change make when you run the program?