Chapter 7

Though These Be Methods, Yet There Is Madness in't

In This Chapter

▶ Matching Java types

▶ Calling methods effectively

▶ Understanding method parameters

In Chapter 5, I compare a method declaration to a recipe for scrambled eggs. In this chapter, I compute the tax and tip for a meal in a restaurant. And in Chapter 9 (spoiler alert!), I compare a Java class to the inventory in a cheese emporium. These comparisons aren't far-fetched. A method's declaration is a lot like a recipe, and a Java class bears some resemblance to a blank inventory sheet. But instead of thinking about methods, recipes, and Java classes, you might be reading between the lines. You might be wondering why this author uses so many food metaphors.

The truth is, my preoccupation with food is a recent development. Like most men my age, I've been told that I should shed my bad habits, lose a few pounds, exercise regularly, and find ways to reduce the stress in my life. (I've argued to my Wiley editors that submission deadlines are a source of stress, but so far the editors aren't buying a word of it. I guess I don't blame them.)

Above all, I've been told to adopt a healthy diet: Skip the chocolate, the cheeseburgers, the pizza, the fatty foods, the fried foods, the sugary snacks, and everything else that I normally eat. Instead, eat small portions of vegetables, carbs, and protein, and eat these things only at regularly scheduled meals. Sounds sensible, doesn't it?

I'm making a sincere effort. I've been eating right for about two weeks. My feelings of health and well-being are steadily improving. I'm only slightly hungry. (Actually, by "slightly hungry," I mean "extremely hungry." Yesterday I suffered a brief hallucination, believing that my computer keyboard was a giant Hershey's bar. And this morning I felt like gnawing on my office furniture. If I start trying to peeling my mouse, I'll stop writing and go out for a snack.)

One way or another, the gustatory arena provides many fine metaphors for object-oriented programming. A method's declaration is like a recipe. A declaration sits quietly, doing nothing, waiting to be executed. If you create a declaration but no one ever calls your declaration, then like a recipe for worm stew, your declaration goes unexecuted.

On the other hand, a *method call* is a call to action — a command to follow the declaration's recipe. When you call a method, the method's declaration wakes up and follows the instructions inside the body of the declaration.

In addition, a method call may contain parameters. You call

```
JOptionPane.showMessageDialog (null, ticketPrice)
```

with the parameters `null` and `ticketPrice`. The first parameter, `null`, tells the computer not to house the dialog box inside another window. The second parameter, `ticketPrice`, tells the computer what to display in the dialog box. In the world of food, you might call `meatLoaf(6)`, which means, "Follow the meat loaf recipe, and make enough to serve six people."

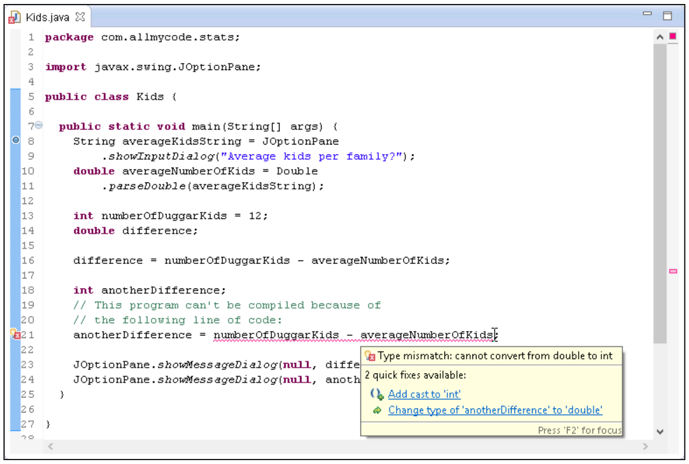
A method has two facets: the first is the method's declaration; the second consists of any statements making calls to the method.

Practice Safe Typing

"You can't fit a square peg into a round hole," or so the saying goes. In Java programming, the saying goes one step further: "Like all other developers, you sometimes make a mistake and try to fit a square peg into a round hole. Java's type system alerts you to the mistake and doesn't let you run the flawed code."

Here's an example illustrating pegs and holes: According to the U.S. census, the average number of children per family in the year 2000 was 0.9. But by mid-2000, the Duggar family (of *19 Kids & Counting* television fame) had 12 children. No matter when you take the census, the average number of children is a `double` value, and the number of children in a particular family is an `int` value.

In Figure 7-1, I try to calculate the Duggar family's divergence from the national average. I don't even show you a run of this program, because the program doesn't work. It's defective. It's damaged goods. As cousin Jeb would say, "This program is a dance party on a leaky raft in a muddy river."



```
package com.allmycode.stats;

import javax.swing.JOptionPane;

public class Kids {

  public static void main(String[] args) {
    String averageKidsString = JOptionPane
        .showInputDialog("Average kids per family?");
    double averageNumberOfKids = Double
        .parseDouble(averageKidsString);

    int numberOfDuggarKids = 12;
    double difference;

    difference = numberOfDuggarKids - averageNumberOfKids;

    int anotherDifference;
    // This program can't be compiled because of
    // the following line of code:
    anotherDifference = numberOfDuggarKids - averageNumberOfKids;

    JOptionPane.showMessageDialog(null, diffe
    JOptionPane.showMessageDialog(null, anoth
  }

}
```

```
Type mismatch: cannot convert from double to int
2 quick fixes available:
 Add cast to 'int'
 Change type of 'anotherDifference' to 'double'
                                    Press 'F2' for focus
```

Figure 7-1: Trying to fit a square peg into a round hole.

The code in Figure 7-1 deals with two types of values — `double` values (in the `averageNumberOfKids` variable) and `int` values (in the `numberOfDuggarKids` variable). You might plan to type `1` when the computer prompts you for `Average kids per family`. But the value stored in the `averageNumberOfKids` variable is of type `double`. An input like `1` or `1.0` doesn't scare the computer into storing anything but a `double` in the `averageNumberOfKids` variable.

The expression `numberOfDuggarKids - averageNumberOfKids` is an `int` minus a `double`, so (according to my sage advice in Chapter 6) the value of `numberOfDuggarKids - averageNumberOfKids` is of type `double`. Sure, if you type `1` when you're prompted for `Average kids per family`, then `numberOfDuggarKids - averageNumberOfKids` is 11.0, and 11.0 is sort of the same as the `int` value 11. But Java doesn't like things to be "sort of the same."

Java's *strong typing* rules say that you can't assign a `double` value (like 11.0) to an `int` variable (like `anotherDifference`). You don't lose any accuracy when you chop the *.0* off *11.0*. But with digits to the right of the decimal point (even with *0* to the right of the decimal point), Java doesn't trust you to stuff a `double` value into an `int` variable. After all, rather than type `1.0` when you're prompted for `Average kids per family`, you can type `0.9`. Then you'd definitely lose accuracy, from stuffing *11.1* into an `int` variable.

You can try to assure Java that things are okay by using a plain, old assignment statement, like this:

```
double averageNumberOfKids;
averageNumberOfKids = 1;
```

When you do, the only way for `numberOfDuggarKids - averageNumberOfKids` to have any value other than 11.0 is for you to make more changes to the Java code. Even so, Java doesn't like assigning 11.0 to the `int` variable `anotherDifference`. This statement is still illegal:

```
anotherDifference =
              numberOfDuggarKids - averageNumberOfKids;
```

When you put numbers in your Java code (like `1` in the previous paragraph or like the number `12` in Figure 7-1) you *hardcode* the values. In this book, my liberal use of hardcoding keeps the examples simple and (more importantly) concrete. But in real applications, hardcoding is generally a bad idea. When you hardcode a value, you make it difficult to change. In fact, the only way to change a hardcoded value is to tinker with the Java code, and all code (written in Java or not) can be brittle. It's much safer to input values in a dialog box (or to read the value from a hard drive or an SD card) than to change a value in a piece of code.

Remember to do as I say and not as I do. Avoid hardcoding values in your programs.

Widening is good; narrowing is bad

Java prevents you from making any assignment that potentially *narrows* a value, as shown in Figure 7-2. For example, if with the declarations

```
int numberOfDuggarKids = 12;
long lotsAndLotsOfKids;
```

the following attempt to narrow from a `long` value to an `int` value is illegal:

```
numberOfDuggarKids = lotsAndLotsOfKids; //Don't do this!
```

An attempt to *widen* from an `int` value to a `long` value, however, is fine:

```
lotsAndLotsOfKids = numberOfDuggarKids;
```

In fact, back in Figure 7-1, I assign an `int` value to a `double` value with no trouble at all:

```
double difference;
difference = numberOfDuggarKids - averageNumberOfKids;
```

Assigning an `int` value to a `double` value is legal because it's an example of widening.

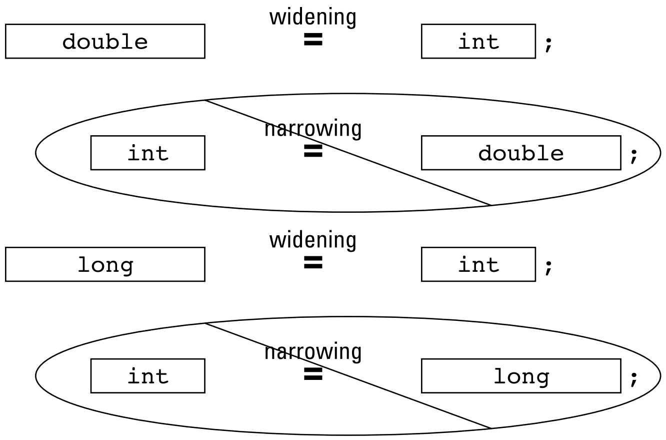


Figure 7-2: Widening and narrowing.

Incompatible types

Aside from the technical terms *narrowing* and *widening*, there's another possibility — plain, old incompatibility — trying to fit one element into another when the two have nothing in common and have no hope of ever being mistaken for one another. You can't assign an `int` value to a `boolean` value or assign a `boolean` value to an `int` value:

```
int numberOfDuggarKids;
boolean isLarge;
numberOfDuggarKids = isLarge; //Don't do this!
isLarge = numberOfDuggarKids; //Don't do this!
```

You can't do either assignment because `boolean` values aren't numeric. In other words, neither of these assignments makes sense.

Java is a *strongly typed* computer programming language. It doesn't let you make assignments that might result in a loss of accuracy or in outright nonsense.

Using a hammer to bang a peg into a hole

In some cases, you can circumvent Java's prohibition against narrowing by *casting* a value. For example, you can create the `long` variable `lotsAndLotsOfKids` and make the assignment `numberOfDuggarKids = (int) lotsAndLotsOfKids`, as shown in Listing 7-1.

Listing 7-1: Casting to the Rescue

```
package com.allmycode.stats;

import javax.swing.JOptionPane;

public class MoreKids {

  public static void main(String[] args) {
    long lotsAndLotsOfKids = 2147483647;
    int numberOfDuggarKids;

    numberOfDuggarKids = (int) lotsAndLotsOfKids;

    JOptionPane.showMessageDialog
                     (null, numberOfDuggarKids);
  }

}
```

The type name `(int)` in parentheses is a *cast operator*. It tells the computer that you're aware of the potential pitfalls of stuffing a `long` value into an `int` variable and that you're willing to take your chances.

When you run the code in Listing 7-1, the value of `lotsAndLotsOfKids` might be between –2147483648 and 2147483647. If so, the assignment `numberOfDuggarKids = (int) lotsAndLotsOfKids` is just fine. (***Remember:*** An `int` value can be between –2147483648 and 2147483647. Refer to Table 6-1.)

But if the value of `lotsAndLotsOfKids` isn't between –2147483648 and 2147483647, the assignment statement in Listing 7-1 goes awry. When I run the code in Listing 7-1 with the different initialization

```
long lotsAndLotsOfKids = 2098797070970970956L;
```

the value of `numberOfDuggarKids`. becomes –287644852 (a negative number!).

When you use a casting operator, you're telling the computer, "I'm aware that I'm doing something risky but (trust me) I know what I'm doing." And if you don't know what you're doing, you get a wrong answer. That's life!

Calling a Method

After all the fuss I make in the previous section over type safety for assignment statements, I should give equal time to type safety for method calls. After all, a method call involves values going both ways — from the call to the running method and from the running method back to the call. Here are the details:

- **In a method call, each parameter has a value. The computer sends that value to one of the declaration's parameters.**

  In a method call, each parameter has a type. The types of the parameters in the method's declaration must match the types of parameters in the method call.

- **A method declaration might contain a `return` statement, and the `return` statement might calculate a particular value. If so, the computer assigns that value back to the entire method call.**

  A method's *return type* is the type of value calculated by the `return` statement. So the return type is the type of the method call's value.

To make this concept more concrete, consider the code in Listing 7-2.

Listing 7-2: Parameter Types and Return Types

```
package com.allmycode.money;

import java.text.NumberFormat;

import javax.swing.JOptionPane;

public class Mortgage {

  public static void main(String[] args) {
    double principal = 100000.00, ratePercent = 5.25;
    double payment;
    int years = 30;
    String paymentString;

    payment =
        monthlyPayment(principal, ratePercent, years);

    NumberFormat currency =
        NumberFormat.getCurrencyInstance();
    paymentString = currency.format(payment);
    JOptionPane.showMessageDialog(null,
        paymentString, "Monthly payment",
        JOptionPane.INFORMATION_MESSAGE);

  }

  static double monthlyPayment
   (double pPrincipal, double pRatePercent, int pYears) {

    double rate, effectiveAnnualRate;
    int paymentsPerYear = 12, numberOfPayments;
    rate = pRatePercent / 100.00;
    numberOfPayments = paymentsPerYear * pYears;
    effectiveAnnualRate = rate / paymentsPerYear;
```

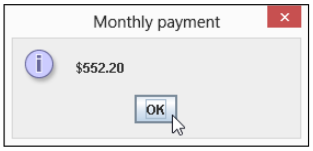
(continued)

Listing 7-2 *(continued)*

```
    return pPrincipal * (effectiveAnnualRate /
                (1 - Math.pow(1 + effectiveAnnualRate,
                  -numberOfPayments)));
  }

}
```

Again, to keep the example simple, I hardcode the values of the variables `principal`, `ratePercent`, and `years`, making Listing 7-2 useless for anything except one particular calculation. In a real app, you'd ask the user for the values of these variables.

Figure 7-3 shows the output of a run of the code in Listing 7-2.



```
$552.20
```

Figure 7-3: Pay it and weep.

In Listing 7-2, I choose the parameter names `principal` and `pPrincipal`, `ratePercent` and `pRatePercent`, and `years` and `pYears`. I use the letter `p` to distinguish a declaration's parameter from a call's parameter. I do this to drive home the point that the names in the call aren't automatically the same as the names in the declaration. In fact, there are many variations on this call/declaration naming theme, and they're all correct. For example, you can use the same names in the call as in the declaration:

```
  payment =
      monthlyPayment(principal, ratePercent, years);


static double monthlyPayment
 (double principal, double ratePercent, int years) {
```

You can use expressions in the call that aren't single variable names:

```
  payment =
      monthlyPayment(amount + fees, rate * 100, 30);


static double monthlyPayment
 (double pPrincipal, double pRatePercent, int pYears) {
```

When you call a method from Java's API, you don't even know the names of parameters used in the method's declaration. And you don't care. The only things that matter are the positions of parameters in the list and the compatibility of the parameters:

- **The value of the call's leftmost parameter becomes the value of the declaration's leftmost parameter, no matter what name the declaration's leftmost parameter has.**

  Of course, the types of the two leftmost parameters (the call's parameter and the declaration's parameter) must be compatible.

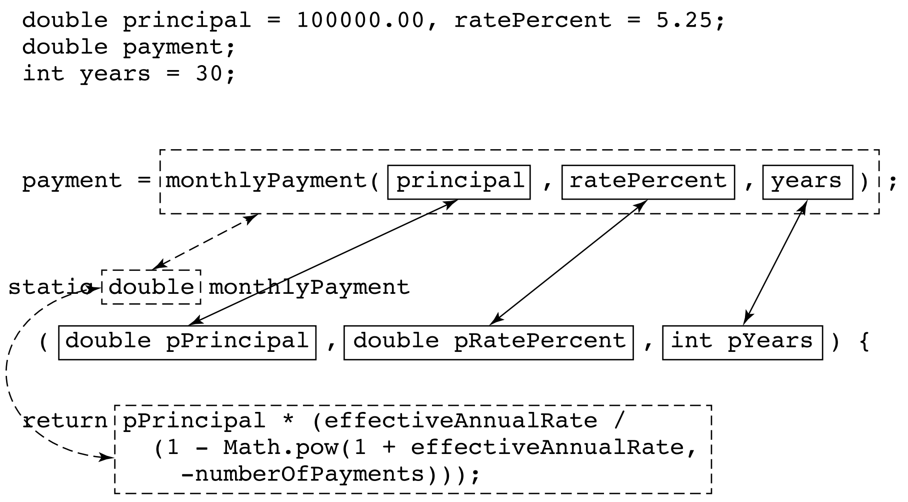
- **The value of the call's second parameter becomes the value of the declaration's second parameter, no matter what name the declaration's second parameter has.**

  And so on.

Real Java developers start the names of variables and methods with lowercase letters. You can ignore this convention and create a method named `MonthlyPayment` or `MONTHLY_PAYMENT`, for example. But if you ignore the convention, some developers will wince when they read your code.

Method parameters and Java types

Listing 7-2 contains both the declaration and a call for the `monthlyPayment` method. Figure 7-4 illustrates the type matches between these two parts of the program.



```
double principal = 100000.00, ratePercent = 5.25;
double payment;
int years = 30;

payment = monthlyPayment( principal , ratePercent , years ) ;

static double monthlyPayment
  ( double pPrincipal , double pRatePercent , int pYears ) {

return pPrincipal * (effectiveAnnualRate /
  (1 - Math.pow(1 + effectiveAnnualRate,
    -numberOfPayments)));
```

Figure 7-4: Each value fits like a glove.

In Figure 7-4, the `monthlyPayment` method call has three parameters, and the `monthlyPayment` declaration's header has three parameters. The call's three parameters have the types `double` and then `double` and then `int`. And sure enough, the declaration's three parameters have the types `double` and then `double` and then `int`.

As in the earlier section "Practice Safe Typing," you don't need an exact match between a method call's parameter and the declaration's parameter. You can take advantage of widening. For example, in Listing 7-2, adding the following call would be okay:

```
payment = monthlyPayment(100000, 5, years);
```

You can pass an `int` value (like `100000`) to the `pPrincipal` parameter, because the `pPrincipal` parameter is of type `double`. Java widens the values `100000` and `5` to the values `100000.0` and `5.0`. But, once again, Java doesn't narrow your values. The following call causes a big red blotch in the Eclipse editor:

```
payment = monthlyPayment(principal, ratePercent, 30.0);
```

You can't stuff a `double` value (like `30.0`) into the `pYears` parameter, because the `pYears` parameter is of type `int`.

In a method declaration, each parameter has the form

```
typeName variableName
```

For example, in the declaration that starts with `static double monthlyPayment(double pPrincipal`, the word `double` is a *`typeName`*, and the word `pPrincipal` is a *`variableName`*. But in a method call, each parameter is an expression with a certain value. In the `main` method in Listing 7-2, the call `monthlyPayment(principal, ratePercent, years)` contains three parameters: `principal`, `ratePercent`, and `years`. Each of these parameters has a value. So with the initializations in the `main` method, the call `monthlyPayment(principal, ratePercent, years)` is essentially the same as calling `monthlyPayment(100000.00, 5.25, 30)`. In fact, a call like `monthlyPayment(100000.00, 5.25, 30)` or `monthlyPayment(10 * 1000.00, 5 + 0.25, 30)` is legal in Java. A method call's parameters can be expressions of any kind. The only requirement is that the expressions in the call have types that are compatible with the corresponding parameters in the method's declaration.

Return types

A method declaration's header normally looks like this:

```
someWords returnType methodName(parameters) {
```

For example, Listing 7-2 contains a method declaration with the following header:

```
static double monthlyPayment
  (double pPrincipal, double pRatePercent, int pYears)
```

In this header, the *returnType* is double, the *methodName* is monthly Payment, and the *parameters* are double pPrincipal, double pRatePercent, int pYears.

A method declaration's parameter list differs from the method call's parameter list. The declaration's parameter list contains the name of each parameter's type. In contrast, the call's parameter list contains no type names.

An entire method call can have a value, and the declaration's *returnType* tells the computer what type that value has. In Listing 7-2, the *returnType* is double, so the call

```
monthlyPayment(principal, ratePercent, years)
```

has a value of type double. (Refer to Figure 7-4.)

I hardcoded the values of principal, ratePercent, and years in Listing 7-2. So when you run Listing 7-2, the value of the monthlyPayment method call is always 552.20. The call's value is whatever comes after the word return when the method is executed. And in Listing 7-2, the expression

```
pPrincipal * (effectiveAnnualRate /
  (1 - Math.pow(1 + effectiveAnnualRate,
    -numberOfPayments)))
```

always comes out to be 552.20. Also, in keeping with the theme of type safety, the expression after the word return is of type double.

In summary, a call to the monthlyPayment method has the *return value* 552.20 and has the *return type* double.

Only book authors and bad programmers hardcode values like `principal`, `ratePercent`, and `years`. I hardcoded these values to keep the example as simple as possible. But, normally, values like these should be part of the program's input so that the values can change from one run to another.

The great void

A method to compute a monthly mortgage payment naturally returns a value. But a Java program's `main` method, or Java's own `showMessageDialog` method (with no user input), has little reason to return a value.

When a method doesn't return a value, the method's body has no `return` statement. And, in place of a return type, the header in the method's declaration contains the word `void`. A program's `main` method doesn't return a value, so when you create a main method, you type

```
public static void main(String args[]) {
```

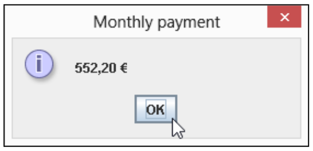
To be painfully precise, you can put a `return` statement in a method that doesn't return a value. When you do, the `return` statement has no expression. It's just one word, `return`, followed by a semicolon. When the computer executes this `return` statement, the computer ends the run of the method and returns to the code that called the method. This rarely used form of the `return` statement works well in a situation in which you want to end the execution of a method before you reach the last statement in the method's declaration.

Displaying numbers

Here are a few lines that are scattered about in Listing 7-2:

```
import java.text.NumberFormat;

NumberFormat currency =
        NumberFormat.getCurrencyInstance();
paymentString = currency.format(payment);
```

Taken together, these statements give you easy formatting of numbers into local currency amounts. On my computer, when I call `getCurrency Instance()` with no parameters, I get a number (like 552.2) formatted for United States currency. (Refer to Figure 7-3.) But if your computer is set to run in Germany, you see the message box shown in Figure 7-5.



```
552,20 €
```

Figure 7-5: Displaying the euro symbol.

A country, its native language, or a variant of the native language is a *locale*. And by adding a parameter to the `getCurrencyInstance` call, you can format for locales other than your own. For example, by calling

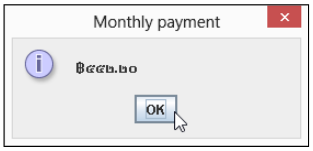
```
NumberFormat.getCurrencyInstance(Locale.GERMANY)
```

anyone in any country can get the message box shown in Figure 7-5.

In the choice of available locales, standard Oracle Java is a bit better than Android Java. For example, the `Locale.GERMANY` trick works in standard Java and in Android Java. But some variants of the Thai language use their own, special digit symbols. (See Figure 7-6.) To form a number with Thai digits, you need

```
NumberFormat.getCurrencyInstance(
                                  new Locale("th", "TH", "TH"))
```

And this locale works only in standard Java.



```
฿๕๕๒.๒๐
```

Figure 7-6: Thai digit symbols.

Method overload without software bloat

Chapter 5 introduces method overloading. But that chapter doesn't show you a complete example using method overloading. Listing 7-3 remedies this situation.

Listing 7-3: Filling but Not Fatty (Yes, I'm Still Hungry)

```
package com.allmycode.money;

import java.text.NumberFormat;

import javax.swing.JOptionPane;

public class Mortgage {

  public static void main(String[] args) {
    double principal = 100000.00, ratePercent = 5.25;
    double payment;
    int years = 30;
    String paymentString;
    NumberFormat currency =
        NumberFormat.getCurrencyInstance();

    payment =
        monthlyPayment(principal, ratePercent, years);
    paymentString = currency.format(payment);
    JOptionPane.showMessageDialog(null,
        paymentString, "Monthly payment",
        JOptionPane.INFORMATION_MESSAGE);

    ratePercent = 3.0;
    payment = monthlyPayment(principal, ratePercent);
    paymentString = currency.format(payment);
    JOptionPane.showMessageDialog(null,
        paymentString, "Monthly payment",
        JOptionPane.INFORMATION_MESSAGE);

    payment = monthlyPayment();
    paymentString = currency.format(payment);
    JOptionPane.showMessageDialog(null,
        paymentString, "Monthly payment",
        JOptionPane.INFORMATION_MESSAGE);

  }

  static double monthlyPayment
   (double pPrincipal, double pRatePercent, int pYears) {

    double rate, effectiveAnnualRate;
    int paymentsPerYear = 12, numberOfPayments;
    rate = pRatePercent / 100.00;
    numberOfPayments = paymentsPerYear * pYears;
    effectiveAnnualRate = rate / paymentsPerYear;
    return pPrincipal * (effectiveAnnualRate /
             (1 - Math.pow(1 + effectiveAnnualRate,
               -numberOfPayments)));
  }

  static double monthlyPayment
   (double pPrincipal, double pRatePercent) {

    return monthlyPayment(pPrincipal, pRatePercent, 30);
  }

  static double monthlyPayment() {
    return 0.0;
  }
}
```

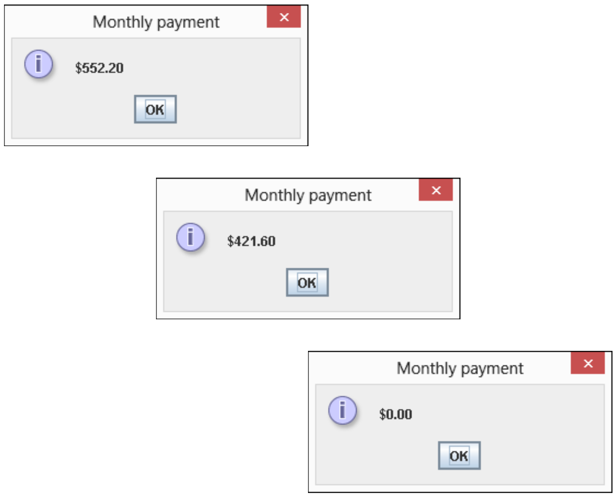
The three dialog boxes that you see when you run the code in Listing 7-3 are shown in Figure 7-7.

In Listing 7-3, the `monthlyPayment` method has three declarations, each with its own parameter list, and with each parameter list representing a different bunch of types. As a method name, the name `monthlyPayment` is *overloaded*.

- **The first `monthlyPayment` declaration is a copy of the declaration in Listing 7-2.**

  When you call the first declaration, you supply values for three parameters — two `double` values and one `int` value:

  ```
  monthlyPayment(principal, ratePercent, years)
  ```



```
$552.20
```

```
$421.60
```

```
$0.00
```

Figure 7-7: Running the code in Listing 7-3.

- **The second `monthlyPayment` declaration has only two parameters.**

  When you call the second declaration, you supply values for only two double parameters:

  ```
  monthlyPayment(principal, ratePercent)
  ```

  When the computer encounters this `monthlyPayment` call with two `double` parameters, the computer executes the `monthlyPayment` declaration that has two `double` parameters. (See Listing 7-3.) This automatic choice of method declaration is what makes overloading work.

  Notice the trick that I use in the body of the two-parameter `monthlyPayment` declaration. To create the two-parameter declaration, I could get away with simply duplicating the code from the three-parameter `monthlyPayment` declaration:

  ```
  // (Insert throat-clearing here.) This duplication
  // of code isn't a very good idea.
  static double monthlyPayment
   (double pPrincipal, double pRatePercent) {

    double rate, effectiveAnnualRate;
    int paymentsPerYear = 12, numberOfPayments;
    rate = pRatePercent / 100.00;
    numberOfPayments = paymentsPerYear * 30;
    effectiveAnnualRate = rate / paymentsPerYear;
    return pPrincipal * (effectiveAnnualRate /
             (1 - Math.pow(1 + effectiveAnnualRate,
               -numberOfPayments)));
  }
  ```

  But duplicating code is a bad idea. Copying and pasting code causes errors down the road. In Listing 7-3, I don't copy the three-parameter code. Instead, I call the three-parameter `monthlyPayment` method from the body of the two-parameter `monthlyPayment` method. I supply a default value of `30` for the third `pYears` parameter. In the program's documentation, I must state clearly that the two-parameter `monthlyPayment` method assumes a 30-year mortgage term.

- **The third `monthlyPayment` declaration has no parameters.**

  When you call the third declaration in Listing 7-3, you don't supply values for any parameters. Instead, you follow the method's name with an empty pair of parentheses:

  ```
  monthlyPayment()
  ```

> The parameterless `monthlyPayment` method might be useful in those don't-know-what-else-to-do situations. You have to display something about a borrower who hasn't yet decided on the principal, rate, or number of years. With little or no information about a mortgage loan, you display `$0.00` as a temporary value for the borrower's monthly payment.

For method overloading to work, the parameter types in a call must match the parameter types in a declaration. In Listing 7-3, no two `monthlyPayment` declarations have the same number of parameters, so parameter matching isn't too challenging.

But there's more to matching than having the same number of parameters. For example, you can add another two-parameter declaration to the code in Listing 7-3:

```
static double monthlyPayment
  (double pPrincipal, int pYears) {
```

With this addition, you have more than one two-parameter `monthly Payment` declaration — an old declaration with two `double` parameters and a new declaration with a `double` parameter and an `int` parameter. If you call `monthlyPayment(principal, 15)`, the computer calls the newly added method. It calls the new method because the new method, with its `double` and `int` parameters, is a better match for your call than the old `monthlyPayment(`**`double`**` pPrincipal, `**`double`**` pRatePercent)` declaration in Listing 7-3.

Primitive Types and Pass-by Value

Java has two kinds of types: primitive and reference. The eight primitive types are the atoms — the basic building blocks. In contrast, the reference types are the things you create by combining primitive types (and by combining other reference types).

My coverage of Java's reference types begins in Chapter 9.

Here are two concepts you should remember when you think about primitive types and method parameters:

- **When you assign a value to a variable with a primitive type, you're identifying that variable name with the value.**

The same is true when you initialize a primitive type variable to a particular value.

- **When you call a method, you're *making copies* of each of the call's parameter values and initializing the declaration's parameters with those copied values.**

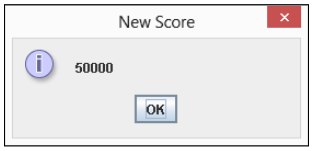
This scheme, in which you make copies of the call's values, is named *pass-by value*. Listing 7-4 shows you why you should care about any of this.

Listing 7-4: Rack Up Those Points!

```java
import javax.swing.JOptionPane;

public class Scorekeeper {

  public static void main(String[] args) {
    int score = 50000;
    int points = 1000;
    addPoints(score, points);
    JOptionPane.showMessageDialog(null, score,
        "New Score", JOptionPane.INFORMATION_MESSAGE);
  }

  static void addPoints(int score, int points) {
    score += points;
  }

}
```

In Listing 7-4, the `addPoints` method uses Java's compound assignment operator to add 1000 (the value of `points`) to the existing `score` (which is 50000). To make things as cozy as possible, I've used the same parameter names in the method call and the method declaration. (In both, I use the names `score` and `points`.)

So what happens when I run the code in Listing 7-4? I get the result shown in Figure 7-8.



```
50000
```

Figure 7-8: Getting 1000 more points?

But wait! When you add 1000 to 50000, you don't normally get 50000. What's wrong?

With Java's pass-by value feature, you *make a copy* of each parameter value in a call. You initialize the declaration's parameters with the copied values. So immediately after making the call, you have two pairs of variables: the original `score` and `points` variables in the `main` method and the new `score` and `points` variables in the `addPoints` method. The new `score` and `points` variables have copies of values from the `main` method. (See Figure 7-9.)

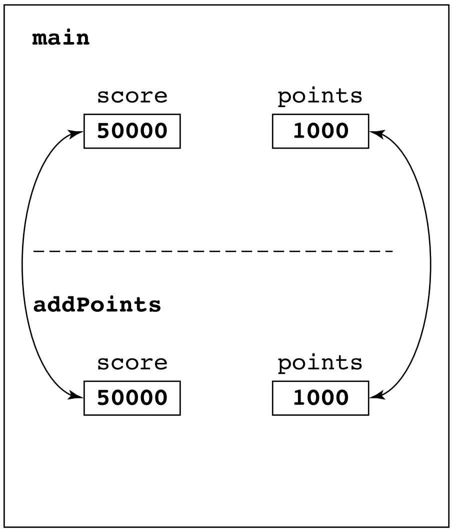


Figure 7-9: Java makes copies of the values of variables.

The statement in the body of the `addPoints` method adds 1000 to the value stored in its `score` variable. After adding 1000 points, the program's variables look like the stuff shown in Figure 7-10.

Notice how the value of the `main` method's `score` variable remains unchanged. After returning from the call to `addPoints`, the `addPoints` method's variables disappear. All that remains is the original `main` method and its variables. (See Figure 7-11.)

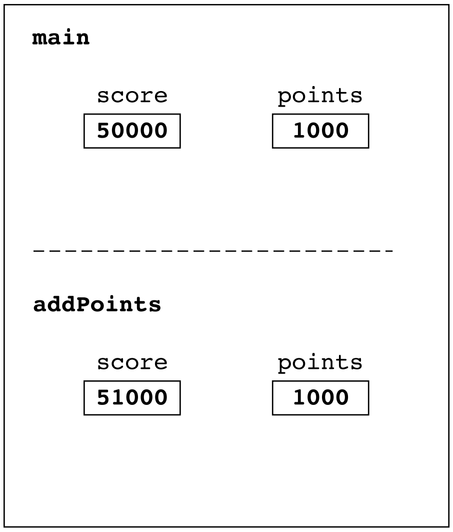


Figure 7-10: Java adds 1000 to only one of the two score variables.

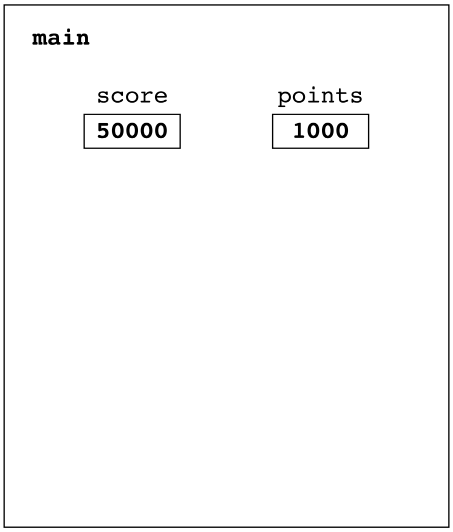


Figure 7-11: The variable with value 51000 no longer exists.

Finally, in Listing 7-4, the computer calls `showMessageDialog` to display the value of the main method's `score` variable. And (sadly, for the game player) the value of `score` is still 50000.

Perils and pitfalls of parameter passing

How would you like to change the value of 2 + 2? What would you like 2 + 2 to be? Six? Ten? Three hundred? In certain older versions of the FORTRAN programming language, you could make 2 + 2 be almost anything you wanted. For example, the following chunk of code (translated to look like Java code) would display `6` for the value of 2 + 2:

```
public void increment(int
   score) {
  score++;
}
...
increment(2);
JOptionPane.
   showMessageDialog(null, 2
   + 2);
```

When computer languages were first being developed, their creators didn't realize how complicated parameter passing can be. They weren't as careful about specifying the rules for copying parameters' values or for doing whatever else they wanted to do with parameters. As a result, some versions of FORTRAN indiscriminately passed memory addresses rather than values. Though address-passing alone isn't a terrible idea, things become ugly if the language designer isn't careful.

In some early FORTRAN implementations, the computer automatically (and without warning) turned the literal `2` into a variable named `two`. (In fact, the newly created variable probably wasn't named `two`. But in this story, the actual name of the variable doesn't matter.) FORTRAN would substitute the variable name `two` in any place where the programmer typed the literal value `2`. But then, while running this sidebar's code, the computer would send the address of the `two` variable to the `increment` method. The method would happily add 1 to whatever was stored in the `two` variable and then continue its work. Now the `two` variable stored the number 3. By the time you reached the `showMessageDialog` call, the computer would add to itself whatever was in `two`, getting 3 + 3, which is 6.

If you think parameter passing is a no-brainer, think again. Different languages use all different kinds of parameter passing. And in many situations, the minute details of the way parameters are passed makes a big difference.

What's a developer to do?

The program in Listing 7-4 has a big, fat bug. The program doesn't add 1000 to a player's score. That's bad.

You can squash the bug in Listing 7-4 in several different ways. For example, you can avoid calling the `addPoints` method by inserting score += points in the main method. But that's not a satisfactory solution. Methods such as addPoints are useful for dividing work into neat, understandable chunks. And avoiding problems by skirting around them is no fun at all.

A better way to get rid of the bug is to make the `addPoints` method return a value. Listing 7-5 has the code.

Listing 7-5: A New-and-Improved Scorekeeper Program

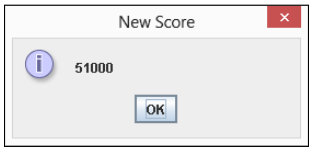
```
import javax.swing.JOptionPane;

public class Scorekeeper {

  public static void main(String[] args) {
    int score = 50000;
    int points = 1000;
    score = addPoints(score, points);
    JOptionPane.showMessageDialog(null, score,
        "New Score", JOptionPane.INFORMATION_MESSAGE);
  }

  static int addPoints(int score, int points) {
    return score + points;
  }

}
```

In Listing 7-5, the new-and-improved `addPoints` method returns an `int` value; namely, the value of `score + points`. So the value of the `addPoints(score, points)` call is 51000. Finally, I change the value of `score` by assigning the method call's value, 51000, to the `score` variable.

TECHNICAL STUFF

Java's nitpicky rules insure that the juggling of the `score` variable's values is reliable and predictable. In the statement `score = addPoints(score, points)`, there's no conflict between the old value of `score` (50000 in the `addPoints` parameter list) and the new value of `score` (51000 on the left side of the assignment statement).

A run of the code in Listing 7-5 is shown in Figure 7-12. You probably already know what the run looks like. (After all, 50000 + 1000 is 51000.) But I can't bear to finish this example without showing the correct answer.


New Score

51000

Figure 7-12: At last, a higher score!

Making `addPoints` return a value isn't the only way to correct the problem in Listing 7-4. At least two other ways (using fields and passing objects) are among the subjects of discussion in Chapter 9.

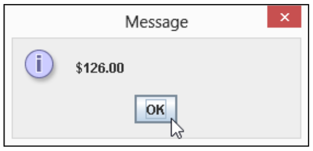
A final word

The program in Listing 7-6 displays the total cost of a $100 meal.

Listing 7-6: Yet Another Food Example

```
package org.allyourcode.food;

import java.text.NumberFormat;

import javax.swing.JOptionPane;

public class CheckCalculator {

  public static void main(String[] args) {
    NumberFormat currency =
        NumberFormat.getCurrencyInstance();
    JOptionPane.showMessageDialog(null,
        currency.format(addAll(100.00, 0.05, 0.20)));
  }

  static double addAll
        (double bill, double taxRate, double tipRate) {
    bill *= 1 + taxRate;
    bill *= 1 + tipRate;
    return bill;
  }

}
```

A run of the program in Listing 7-6 is shown in Figure 7-13.



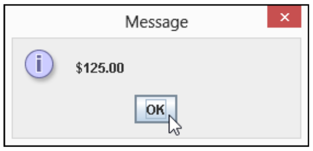
```
$126.00
```

Figure 7-13: Support your local eating establishment.

Listing 7-6 is nice, but this code computes the tip after the tax has been added to the original bill. Some of my less generous friends believe that the tip should be based on only the amount of the original bill. (Guys, you know who you are!) They believe that the code should compute the tax but that it should remember and reuse the original $100.00 amount when calculating the tip. Here's my friends' version of the `addAll` method:

```
static double addAll
      (double bill, double taxRate, double tipRate) {
  double originalBill = bill;
  bill *= 1 + taxRate;
  bill += originalBill * tipRate;
  return bill;
}
```

The new (stingier) total is shown in Figure 7-14.



```
$125.00
```

Figure 7-14: A dollar saved is a dollar earned.

The revised `addAll` method is overly complicated. (In fact, in creating this example, I got this little method wrong two or three times before getting it right.) Wouldn't it be simpler to insist that the `bill` parameter's value never changes? Rather than mess with the `bill` amount, you make up new variables named `tax` and `tip` and total everything in the `return` statement:

```
static double addAll
      (double bill, double taxRate, double tipRate) {
  double tax = bill * taxRate;
  double tip = bill * tipRate;
  return bill + tax + tip;
}
```

When you have these new `tax` and `tip` variables, the `bill` parameter always stores its original value — the value of the untaxed, untipped meal.

After developing this improved code, you make a mental note that the `bill` variable's value shouldn't change. Months later, when your users are paying big bucks for your app and demanding many more features, you might turn the program into a complicated, all-purpose meal calculator with localized currencies and tipping etiquette from around the world. Whatever you do, you always want easy access to that original `bill` value.

After your app has gone viral, you're distracted by the need to count your earnings, pay your servants, and maintain the fresh smell of your private jet's leather seats. With all these pressing issues, you accidentally forget your old promise not to change the `bill` variable. You change the variable's value somewhere in the middle of your 1000-line program. Now you've messed everything up.

But wait! You can have Java remind you that the `bill` parameter's value doesn't change. To do this, you add the keyword `final` (one of Java's modifiers) to the method declaration's parameter list. And while you're at it, you can add `final` to the other parameters (`taxRate` and `tipRate`) in the `addAll` method's parameter list:

```
static double addAll (final double bill,
                      final double taxRate,
                      final double tipRate) {
  double tax = bill * taxRate;
  double tip = bill * tipRate;
  return bill + tax + tip;
}
```

With this use of the word `final`, you're telling the computer not to let you change a parameter's value. If you plug the newest version of `addAll` into the code in Listing 7-6, `bill` becomes 100.00 and `bill` stays 100.00 throughout the execution of the `addAll` method. If you accidentally add the statement

```
bill += valetParkingFee;
```

to your code, Eclipse flags that line as an error because a `final` parameter's value cannot be changed. Isn't it nice to know that, with servants to manage and your private jet to maintain, you can still rely on Java to help you write a good computer program?