
4 Writing Classes

Chapter Objectives

- Discuss the structure and content of a class definition.
- Establish the concept of object state using instance data.
- Describe the effect of visibility modifiers on methods and data.
- Explore the structure of a method definition, including parameters and return values.
- Discuss the structure and purpose of a constructor.
- Define and display arc shapes.
- Load and display images.
- Introduce the concepts needed to create an interactive graphical user interface.
- Explore some basic GUI controls and event processing.

*In **Chapter 3**, we used classes and objects for the various services they provide. That is, we used the predefined classes in the Java API that are provided to us to make the process of writing programs easier. In this chapter, we address the heart of object-oriented programming: writing our own classes to define our own objects. This chapter explores the basics of class definitions, including the structure of methods and the scope and encapsulation of data. The Graphics Track sections of this chapter discuss how to represent arcs and images, and introduce the*

issues necessary to create a truly interactive graphical user interface.

4.1 Classes and Objects Revisited

In [Chapter 1](#), we introduced basic object-oriented concepts, including a brief overview of objects and classes. In [Chapter 3](#), we used several predefined classes from the Java standard class library to create objects and use them for the particular functionality they provided.

In this chapter, we turn our attention to writing our own classes. Although existing class libraries provide many useful classes, the essence of object-oriented program development is the process of designing and implementing our own classes to suit our specific needs.

Recall the basic relationship between an object and a class: a class is a blueprint of an object. The class represents the concept of an object, and any object created from that class is a realization of that concept.


For example, from [Chapter 3](#) we know that the `String` class represents a concept of a character string, and that each `String` object represents a particular string that contains specific characters.

Let's consider another example. Suppose a class called `Student` represents a student at a university. An object created from the `Student` class would represent a particular student. The `Student` class represents the general concept of a student, and every object

created from that class represents an actual student attending the school. In a system that helps manage the business of a university, we would have one `Student` class and thousands of `Student` objects.

Recall that an object has a *state*, which is defined by the values of the *attributes* associated with that object. The attributes of a student may include the student's name, address, major, and grade point average. The `Student` class establishes that each student has these attributes. Each `Student` object stores the values of these attributes for a particular student. In Java, an object's attributes are defined by variables declared within a class.

An object also has *behaviors*, which are defined by the *operations* associated with that object. The operations of a student would include the ability to update that student's address and compute that student's current grade point average. The `Student` class defines the operations, such as the details of how a grade point average is computed. These operations can then be executed on (or by) a particular `Student` object. Note that the behaviors of an object may modify the state of that object. In Java, an object's operations are defined by methods declared within a class.

Figure 4.1  lists some examples of classes, with some attributes and operations that might be defined for objects of those classes. It's up to the program designer to determine what attributes and operations are needed, which depends on the purpose of the program and the role a particular object plays in that purpose. Consider other attributes and operations you might include for these examples.

Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
Flight	Airline Flight number Origin city Destination city Current status	Set airline Set flight number Determine status
Employee	Name Department Title Salary	Set department Set title Set salary Compute wages Compute bonus Compute taxes

Figure 4.1 Examples of classes and some possible attributes and operations

Self-Review Questions

(see answers in [Appendix L](#) )

SR 4.1 What is an attribute?

SR 4.2 What is an operation?


SR 4.3 List some attributes and operations that might be defined for a class called `Book` that represents a book in a library.

SR 4.4 True or False? Explain.

- a. We should use only classes from the Java standard class library when writing our programs—there is no need to define or use other classes.
- b. An operation on an object can change the state of an object.
- c. The current state of an object can affect the result of an operation on that object.
- d. In Java, the state of an object is represented by its methods.

4.2 Anatomy of a Class

In all of our previous examples, we've written a single class containing a single `main` method. These classes represent small but complete programs. These programs often instantiated objects using predefined classes from the Java class library and used those objects for the services they provide. Those predefined classes are part of the program too, but we never really concern ourselves with them other than to know how to interact with them. We simply trust them to provide the services they promise.

The `RollingDice` class shown in [Listing 4.1](#)  contains a `main` method that instantiates two `Die` objects (as in the singular of dice). It then rolls the dice and prints the results. It also calls several other methods provided by the `Die` class, such as the ability to explicitly set and get the current face value of a die.

Listing 4.1

```
//*****  
  
//  RollingDice.java      Author: Lewis/Loftus  
//  
//  Demonstrates the creation and use of a user-defined class.  
//*****
```



```
public class RollingDice
{
    //-----
    //  Creates two Die objects and rolls them several times.
    //-----

    public static void main(String[] args)
    {
        Die die1, die2;
        int sum;

        die1 = new Die();
        die2 = new Die();

        die1.roll();
        die2.roll();

        System.out.println("Die One: " + die1 + ", Die Two: " +
die2);

        die1.roll();
        die2.setFaceValue(4);

        System.out.println("Die One: " + die1 + ", Die Two: " +
die2);

        sum = die1.getFaceValue() + die2.getFaceValue();
    }
}
```

```
        System.out.println("Sum: " + sum);

        sum = die1.roll() + die2.roll();

        System.out.println("Die One: " + die1 + ", Die Two: " +
die2);

        System.out.println("New sum: " + sum);
    }
}
```

Output

```
Die One: 5, Die Two: 2
Die One: 1, Die Two: 4
Sum: 5
Die One: 4, Die Two: 2
New sum: 6
```

The primary difference between this example and previous examples is that the `Die` class is not a predefined part of the Java class library. We have to write the `Die` class ourselves, defining the services we want `Die` objects to perform, if this program is to compile and run.

Every class can contain data declarations and method declarations, as depicted in [Figure 4.2](#). The data declarations represent the data that will be stored in each object of the class. The method declarations

define the services that those objects will provide. Collectively, the data and methods of a class are called the **members** ⓘ of a class.

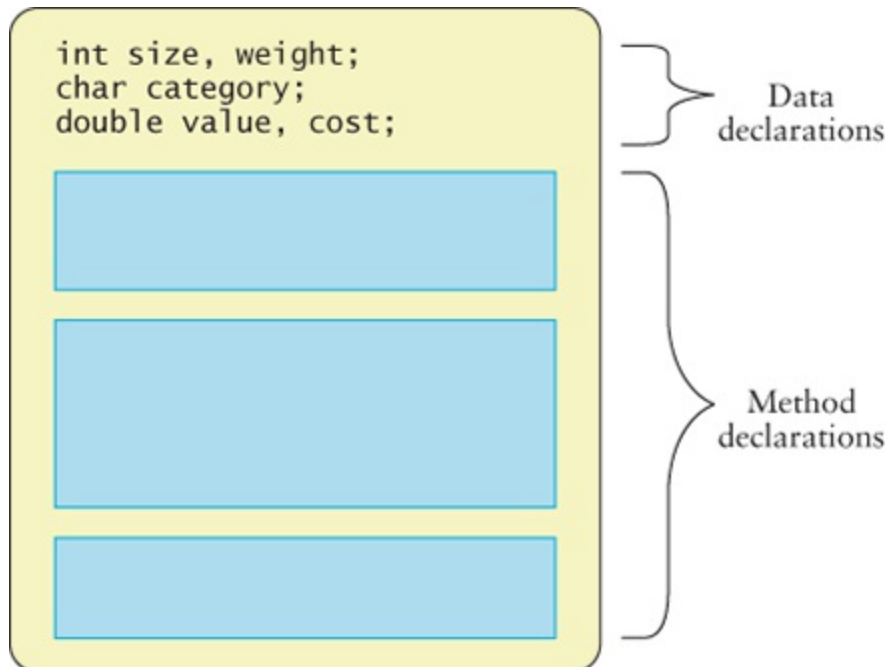


Figure 4.2 The members of a class: data and method declarations

The classes we've written in previous examples follow this model as well, but contain no data at the class level and contain only one method (the `main` method). We'll continue to define classes like this, such as the `RollingDice` class, to define the starting point of a program.

True object-oriented programming, however, comes from defining classes that represent objects with well-defined state and behavior. For example, at any given moment a `Die` object is showing a particular face value, which we could refer to as the state of the die. A `Die` object also has various methods we can invoke on it, such as the

ability to roll the die or get its face value. These methods represent the behavior of a die.

Key Concept

The heart of object-oriented programming is defining classes that represent objects with well-defined state and behavior.

The `Die` class is shown in [Listing 4.2](#). It contains two data values: an integer constant (`MAX`) that represents the maximum face value of the die and an integer variable (`faceValue`) that represents the current face value of the die. It also contains a constructor called `Die` and four regular methods: `roll`, `setFaceValue`, `getFaceValue`, and `toString`.

Listing 4.2

```
//*****  
  
//  Die.java      Author: Lewis/Loftus  
//  
//  Represents one die (singular of dice) with faces showing  
values
```

```

//  between 1 and 6.

//*****

public class Die
{
    private final int MAX = 6;    // maximum face value

    private int faceValue;    // current value showing on the
die

    //-----

    // Constructor: Sets the initial face value.

    //-----

    public Die()
    {
        faceValue = 1;
    }

    //-----

    // Rolls the die and returns the result.

    //-----

    public int roll()
    {

```

```
        faceValue = (int) (Math.random() * MAX) + 1;
```

```
    return faceValue;
```

```
}
```

```
//-----
```

```
// Face value mutator.
```

```
//-----
```

```
public void setFaceValue(int value)
```

```
{
```

```
    faceValue = value;
```

```
}
```

```
//-----
```

```
// Face value accessor.
```

```
//-----
```

```
public int getFaceValue()
```

```
{
```

```
    return faceValue;
```

```
}
```

```
//-----
```

```
// Returns a string representation of this die.
```


```
//-----  
-----  
public String toString()  
{  
    String result = Integer.toString(faceValue);  
  
    return result;  
}  
}
```

You will recall from [Chapters 2](#) and [3](#) that constructors are special methods that have the same name as the class. The `Die` constructor gets called when the `new` operator is used to create a new instance of the `Die` class. The rest of the methods in the `Die` class define the various services provided by `Die` objects.

We use a header block of documentation to explain the purpose of each method in the class. This practice is not only crucial for anyone trying to understand the software, it also separates the code visually so that it's easy for the eye to jump from one method to the next while reading the code.



Dissecting the `Die` class.

Figure 4.3  lists the methods of the `Die` class. From this point of view, it looks no different from any other class that we've used in previous examples. The only important difference is that the `Die` class was not provided for us by the Java API. We wrote it ourselves.

```
Die()  
    Constructor: Sets the initial face value of the die to 1.  
  
int roll()  
    Rolls the die by setting the face value to a random number in the appropriate range.  
  
void setFaceValue(int value)  
    Sets the face value of the die to the specified value.  
  
int getFaceValue()  
    Returns the current face value of the die.  
  
String toString()  
    Returns a string representation of the die indicating its current face value.
```

Figure 4.3 Some methods of the `Die` class

The methods of the `Die` class include the ability to roll the die, producing a new random face value. The `roll` method returns the new face value to the calling method, but you can also get the current face value at any time using the `getFaceValue` method. The `setFaceValue` method sets the face value explicitly, as if you had reached over and turned the die to whatever face you wanted. The `toString` method of any object gets called automatically whenever you pass the object to a `print` or `println` method to obtain a string description of the object to print. Therefore, it's usually a good idea to define a `toString` method for most classes. The definitions of these

methods have various parts, and we'll dissect them as we proceed through this chapter.

For the examples in this book, we usually store each class in its own file. Java allows multiple classes to be stored in one file. If a file contains multiple classes, only one of those classes can be declared using the reserved word `public`. Furthermore, the name of the public class must correspond to the name of the file. For instance, class `Die` is stored in a file called `Die.java`.

Instance Data

Note that in the `Die` class, the constant `MAX` and the variable `faceValue` are declared inside the class but not inside any method.

The location at which a variable is declared defines its *scope*, which is the area within a program in which that variable can be referenced. By being declared at the class level (not within a method), these variables and constants can be referenced in any method of the class.

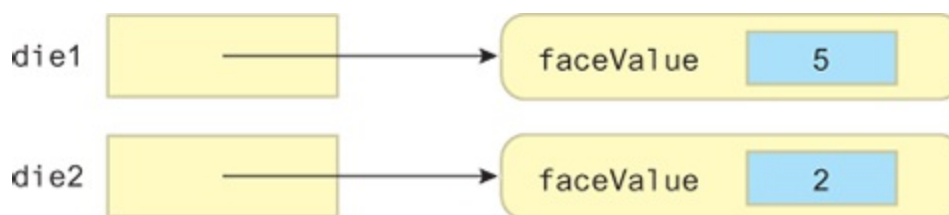
Attributes such as the variable `faceValue` are called *instance data* because new memory space is reserved for that variable every time an instance of the class is created. Each `Die` object has its own `faceValue` variable with its own data space. That's how each `Die` object can have its own state. We see that in the output of the `RollingDice` program: one die has a face value of 5 and the other

has a face value of 2. That's possible only because the memory space for the `faceValue` variable is created for each `Die` object.

Key Concept

The scope of a variable, which determines where it can be referenced, depends on where it is declared.

We can depict this situation as follows:



The `die1` and `die2` reference variables point to (that is, contain the address of) their respective `Die` objects. Each object contains a `faceValue` variable with its own memory space. Thus each object can store different values for its instance data.


Java automatically initializes any variables declared at the class level. For example, all variables of numeric types such as `int` and `double` are initialized to zero. However, despite the fact that the language performs this automatic initialization, it is good practice to initialize

variables explicitly (usually in a constructor) so that anyone reading the code will clearly understand the intent.

UML Class Diagrams

Throughout this book, we use *UML diagrams* to visualize relationships among classes and objects. UML stands for the *Unified Modeling Language*, which has become the most popular notation for representing the design of an object-oriented program.

Several types of UML diagrams exist, each designed to show specific aspects of object-oriented programs. We focus primarily on UML *class diagrams* in this book to show the contents of classes and the relationships among them.

In a UML diagram, each class is represented as a rectangle, possibly containing three sections to show the class name, its attributes (data), and its operations (methods). **Figure 4.4**  shows a class diagram containing the classes of the `RollingDice` program.

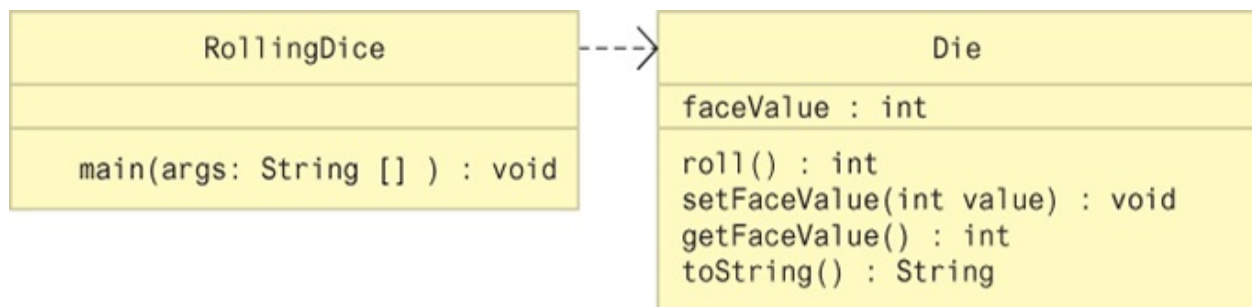



Figure 4.4 A UML class diagram showing the classes involved in

the `RollingDice` program

Key Concept

A UML class diagram helps us visualize the contents of and relationships among the classes of a program.

The arrow connecting the `RollingDice` and `Die` classes in [Figure 4.4](#)  indicates that a relationship exists between the classes. A dotted arrow indicates that one class *uses* the methods of the other class. Other types of object-oriented relationships between classes are shown with different types of connecting lines and arrows. We'll discuss these other relationships as we explore the appropriate topics in the book.

Keep in mind that UML is not designed specifically for Java programmers. It is intended to be language independent. Therefore, the syntax used in a UML diagram is not necessarily the same as Java. For example, the type of a variable is shown after the variable name, separated by a colon. Return types of methods are shown the same way.

UML diagrams are versatile. We can include whatever appropriate information is desired, depending on the goal of a particular diagram.

We might leave out the data and method sections of a class, for instance, if those details aren't relevant for a particular diagram.

UML diagrams allow you to visualize a program's design. As our programs get larger, made up of more and more classes, these visualizations become increasingly helpful. We will explore new aspects of UML diagrams as the situation dictates.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 4.5 What is the difference between an object and a class?

SR 4.6 Describe the instance data of the `Die` class.



SR 4.7 Which of the methods defined for the `Die` class can change the state of a `Die` object—that is, which of the methods assign values to the instance data?

SR 4.8 What happens when you pass an object to a `print` or `println` method?

SR 4.9 What is the scope of a variable?

SR 4.10 What are UML diagrams designed to do?

4.3 Encapsulation

We mentioned in our overview of object-oriented concepts in [Chapter 1](#)  that an object should be *self-governing*. That is, the instance data of an object should be modified only by that object. For example, the methods of the `Die` class should be solely responsible for changing the value of the `faceValue` variable. We should make it difficult, if not impossible, for code outside of a class to “reach in” and change the value of a variable that is declared inside that class. This characteristic is called **encapsulation**. 

An object should be encapsulated from the rest of the system. It should interact with other parts of a program only through the specific set of methods that define the services that that object provides. These methods define the *interface* between that object and the program that uses it.

Key Concept

An object should be encapsulated, guarding its data from inappropriate access.

Encapsulation is depicted graphically in [Figure 4.5](#). The code that uses an object, sometimes called the *client* of an object, should not be allowed to access variables directly. The client should call an object's methods, and those methods then interact with the data encapsulated within the object. For example, the `main` method in the `RollingDice` program calls the `roll` method of the `Die` objects. The `main` method should not (and in fact cannot) access the `faceValue` variable directly.

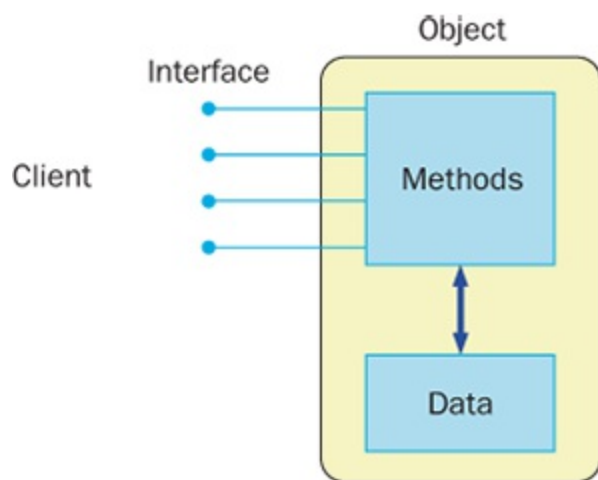


Figure 4.5 A client interacting with the methods of an object

In Java, we accomplish object encapsulation using *modifiers*. A **modifier** ⓘ is a Java reserved word that is used to specify particular characteristics of a programming language construct. In [Chapter 2](#), we discussed the `final` modifier, which is used to declare a constant. Java has several modifiers that can be used in various ways. Some modifiers can be used together, but some combinations are invalid. We discuss various Java modifiers at appropriate points throughout this book, and all of them are summarized in [Appendix E](#).



Visibility Modifiers


Some of the Java modifiers are called **visibility modifiers** ⓘ because they control access to the members of a class. The reserved words `public` and `private` are visibility modifiers that can be applied to the variables and methods of a class. If a member of a class has *public visibility*, it can be directly referenced from outside of the object. If a member of a class has *private visibility*, it can be used anywhere inside the class definition but cannot be referenced externally. A third visibility modifier, `protected`, is relevant only in the context of inheritance. We discuss it in **Chapter 9** 📖.

Key Concept

Instance variables should be declared with private visibility to promote encapsulation.

Public variables violate encapsulation. They allow code external to the class in which the data is defined to reach in and access or modify the value of the data. Therefore, instance data should be defined with private visibility. Data that is declared as `private` can be accessed only by the methods of the class.

The visibility we apply to a method depends on the purpose of that method. Methods that provide services to the client must be declared with public visibility so that they can be invoked by the client. These methods are sometimes referred to as **service methods**.  A `private` method cannot be invoked from outside the class. The only purpose of a `private` method is to help the other methods of the class do their job. Therefore, they are sometimes referred to as **support methods**. 

The table in **Figure 4.6**  summarizes the effects of public and private visibility on both variables and methods.

	public	private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Figure 4.6 The effects of public and private visibility

Giving constants public visibility is generally considered acceptable because, although their values can be accessed directly, they cannot be changed because they were declared using the `final` modifier. Keep in mind that encapsulation means that data values should not be able to be *changed* directly by another part of the code. Because

constants, by definition, cannot be changed, the encapsulation issue is largely moot.

UML class diagrams can show the visibility of a class member by preceding it with a particular character. A member with public visibility is preceded by a plus sign (+), and a member with private visibility is preceded by a minus sign (-).

Accessors and Mutators

Because instance data is generally declared with private visibility, a class usually provides services to access and modify data values. A method such as `getFaceValue` is called an *accessor method* because it provides read-only access to a particular value. Likewise, a method such as `setFaceValue` is called a *mutator method* because it changes a particular value.

Key Concept

Most objects contain accessor and mutator methods to allow the client to manage data in a controlled manner.

Often, accessor method names have the form `getX`, where `x` is the value to which it provides access. Likewise, mutator method names have the form `setX`, where `x` is the value they are setting. Therefore, these types of methods are sometimes referred to as “getters” and “setters.”

For example, if a class contains the instance variable `height`, it might also contain the methods `getHeight` and `setHeight`. Note that this naming convention capitalizes the first letter of the variable when used in the method names, which is consistent with how method names are written in general.

Some methods may provide accessor and/or mutator capabilities as a side effect of their primary purpose. For example, the `roll` method of the `Die` class changes the `faceValue` of the die and returns that new value as well. Therefore, `roll` is also a mutator method.

Note that the code of the `roll` method is careful to keep the face value of the die in the valid range (1 to `MAX`). Service methods must be carefully designed to permit only appropriate access and valid changes. This points out a flaw in the design of the `Die` class: there is no restriction on the `setFaceValue` method—a client could use it to set the die value to a number such as 20, which is outside the valid range. The code of the `setFaceValue` method should allow only valid modifications to the face value of a die. We explore how that kind of control can be accomplished in the next chapter.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 4.11 Objects should be self-governing. Explain.

SR 4.12 What is the interface to an object?

SR 4.13 What is a modifier?


SR 4.14 Why might a constant be given public visibility?

SR 4.15 Describe each of the following:


- a. public method
- b. private method
- c. public variable
- d. private variable

4.4 Anatomy of a Method

We've seen that a class is composed of data declarations and method declarations. Let's examine method declarations in more detail.

As we stated in [Chapter 1](#) , a method is a group of programming language statements that is given a name. A *method declaration* specifies the code that is executed when the method is invoked. Every method in a Java program is part of a particular class.

When a method is called, the flow of control transfers to that method. One by one, the statements of that method are executed. When that method is done, control returns to the location where the call was made and execution continues.

The *called method* (the one that is invoked) might be part of the same class as the *calling method* that invoked it. If the called method is part of the same class, only the method name is needed to invoke it. If it is part of a different class, it is invoked through the name of an object of that other class, as we've seen many times. [Figure 4.7](#)  shows the flow of execution as methods are called.

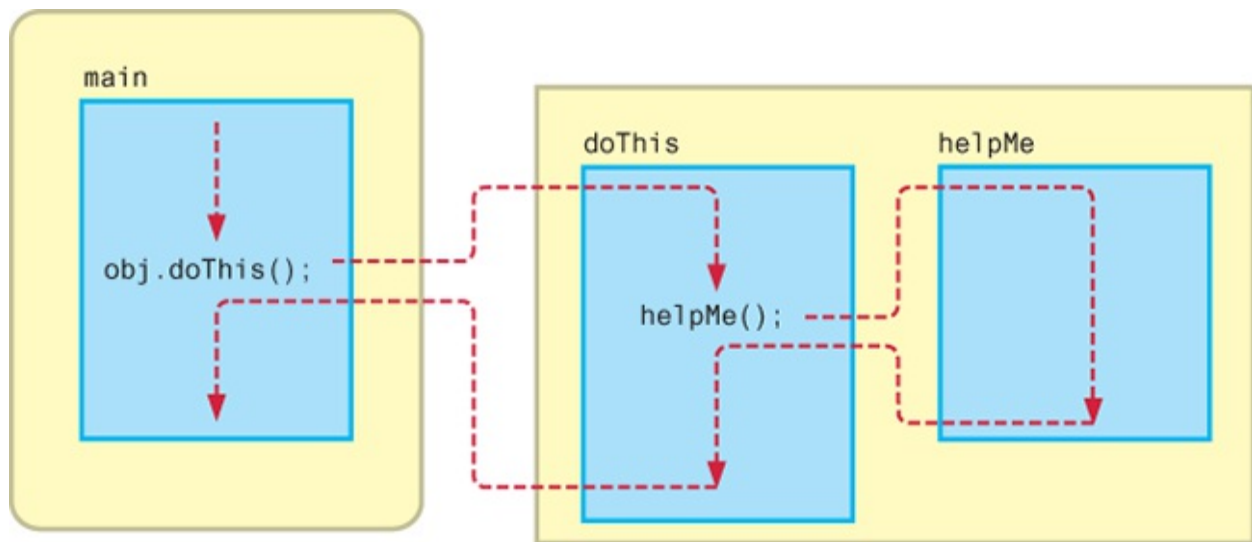
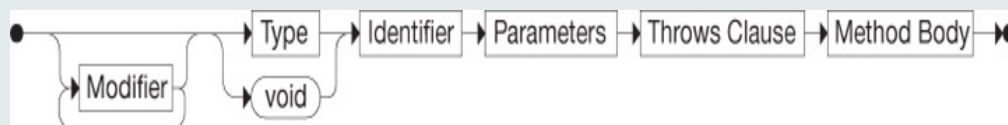


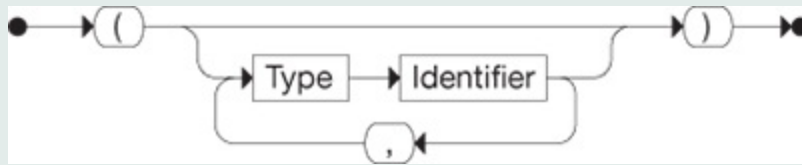
Figure 4.7 The flow of control following method invocations

We've defined the `main` method of a program many times in previous examples. Its definition follows the same syntax as all methods. The header of a method includes the type of the return value, the method name, and a list of parameters that the method accepts. The statements that make up the body of the method are defined in a block delimited by braces. The rest of this section discusses issues related to method declarations in more detail.

Method Declaration



Parameters



A method is defined by optional modifiers, followed by a return Type, followed by an Identifier that determines the method name, followed by a list of Parameters, followed by the Method Body. The return Type indicates the type of value that will be returned by the method, which may be `void`. The Method Body is a block of statements that executes when the method is invoked. The Throws Clause is optional and indicates the exceptions that may be thrown by this method.

Example:

```
public void instructions(int count)
{
    System.out.println("Follow all instructions.");
    System.out.println("Use no more than " + count +
        " turns.");
}
```

The `return` Statement

The return type specified in the method header can be a primitive type, class name, or the reserved word `void`. When a method does not return any value, `void` is used as the return type, as is always done with the `main` method. The `setFaceValue` method of the `Die` class also has a return type of `void`.

A method that returns a value must have a *return statement*. When a `return` statement is executed, control is immediately returned to the statement in the calling method, and processing continues there. A `return` statement consists of the reserved word `return` followed by an expression that dictates the value to be returned. The expression must be consistent with the return type in the method header.

The `getFaceValue` method of the `Die` class returns an `int` value that represents the current value of the die. The `roll` method does the same, returning the new value to which `faceValue` was just randomly set. The `toString` method returns a `String` object.

Key Concept

The value returned from a method must be consistent with the return type specified in the method header.

A method that does not return a value does not usually contain a `return` statement. The method automatically returns to the calling method when the end of the method is reached. Such methods may contain a `return` statement without an expression.

It is usually not good practice to use more than one `return` statement in a method, even though it is possible to do so. In general, a method should have one `return` statement as the last line of the method body, unless that makes the method overly complex.

The value that is returned from a method can be ignored in the calling method. For example, in the `main` method of the `RollingDice` class, the value that is returned from the `roll` method is ignored in several calls, while in others the return value is used in a calculation.

Return Statement



A `return` statement consists of the `return` reserved word followed by an optional Expression. When

executed, control is immediately returned to the calling method, returning the value defined by Expression.

Examples:

```
return;  
return distance * 4;
```

Constructors do not have a return type (not even `void`) and therefore cannot return a value. We discuss constructors in more detail later in this chapter.

Parameters

As we defined in [Chapter 2](#), a parameter is a value that is passed into a method when it is invoked. The **parameter list** in the header of a method specifies the types of the values that are passed and the names by which the called method will refer to those values.

The names of the parameters in the header of the method declaration are called **formal parameters**. In an invocation, the values passed into a method are called **actual parameters**. The actual parameters are also called the *arguments* to the method.

A method invocation and definition always give the parameter list in parentheses after the method name. If there are no parameters, an empty set of parentheses is used, as is the case in the `roll` and `getFaceValue` methods. The `Die` constructor also takes no parameters, although constructors often do.

The formal parameters are identifiers that serve as variables inside the method and whose initial values come from the actual parameters in the invocation. When a method is called, the value in each actual parameter is copied and stored in the corresponding formal parameter. Actual parameters can be literals, variables, or full expressions. If an expression is used as an actual parameter, it is fully evaluated before the method call and the result is passed as the parameter.

Key Concept

When a method is called, the actual parameters are copied into the formal parameters.

The only method in the `Die` class that accepts any parameters is the `setFaceValue` method, which accepts a single `int` parameter. The formal parameter name is `value`. In the `main` method, the value of 4 is passed into it as the actual parameter.

The parameter lists in the invocation and the method declaration must match up. That is, the value of the first actual parameter is copied into the first formal parameter, the second actual parameter into the second formal parameter, and so on, as shown in [Figure 4.8](#). The types of the actual parameters must be consistent with the specified types of the formal parameters.

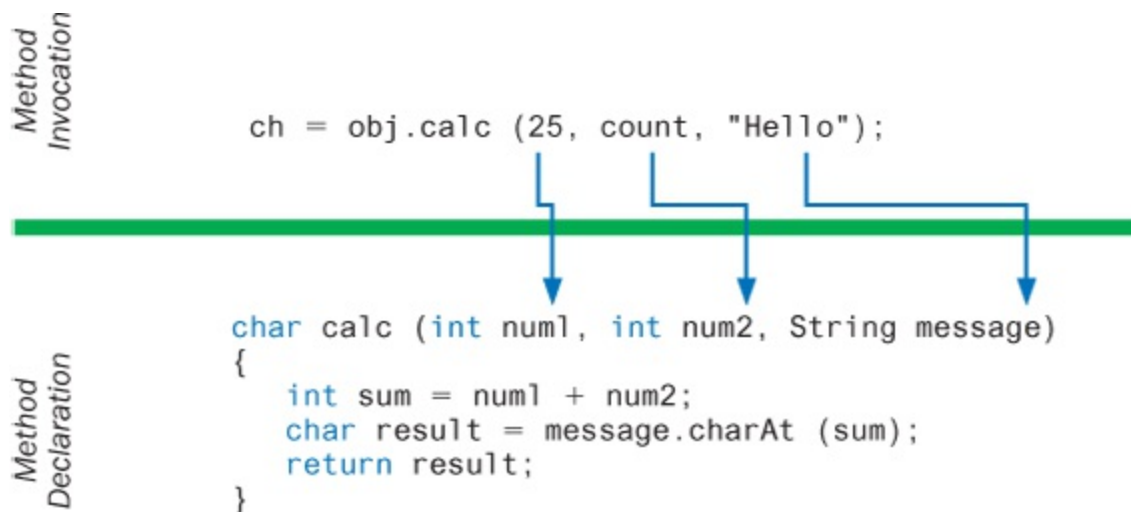


Figure 4.8 Passing parameters from the method invocation to the declaration

Other details regarding parameter passing are discussed in [Chapter 7](#).

Local Data

As we described earlier in this chapter, the scope of a variable or constant is the part of a program in which a valid reference to that

variable can be made. A variable can be declared inside a method, making it *local data* as opposed to instance data. Recall that instance data is declared in a class but not inside any particular method.

Key Concept

A variable declared in a method is local to that method and cannot be used outside of it.

Local data has scope limited to only the method in which it is declared. The variable `result` declared in the `toString` method of the `Die` class is local data. Any reference to `result` in any other method of the `Die` class would have caused the compiler to issue an error message. A local variable simply does not exist outside of the method in which it is declared. On the other hand, instance data, declared at the class level, has a scope of the entire class; any method of the class can refer to it.

Because local data and instance data operate at different levels of scope, it's possible to declare a local variable inside a method with the same name as an instance variable declared at the class level. Referring to that name in the method will reference the local version of the variable. This naming practice obviously has the potential to confuse anyone reading the code, so it should be avoided.

The formal parameter names in a method header serve as local data for that method. They don't exist until the method is called, and they cease to exist when the method is exited. For example, the formal parameter `value` in the `setFaceValue` method comes into existence when the method is called and goes out of existence when the method finishes executing.

Bank Account Example

Let's look at another example of a class and its use. The `Transactions` class shown in [Listing 4.3](#) contains a `main` method that creates a few `Account` objects and invokes their services.

Listing 4.3

```
//*****

// Transactions.java      Author: Lewis/Loftus
//
// Demonstrates the creation and use of multiple Account
// objects.
//*****


public class Transactions
{
```

```
//-----  
-----  
  
//  Creates some bank accounts and requests various  
services.  
  
//-----  
-----  
  
public static void main(String[] args)  
{  
  
    Account acct1 = new Account("Ted Murphy", 72354,  
102.56);  
  
    Account acct2 = new Account("Jane Smith", 69713, 40.00);  
    Account acct3 = new Account("Edward Demsey", 93757,  
759.32);  
  
  
    acct1.deposit(25.85);  
  
  
    double smithBalance = acct2.deposit(500.00);  
    System.out.println("Smith balance after deposit: " +  
        smithBalance);  
  
  
    System.out.println("Smith balance after withdrawal: " +  
        acct2.withdraw (430.75, 1.50));  
  
  
    acct1.addInterest();  
    acct2.addInterest();  
    acct3.addInterest();  
  
  
    System.out.println();  
}
```

```
        System.out.println(acct1);  
        System.out.println(acct2);  
        System.out.println(acct3);  
    }  
}
```

Output

```
Smith balance after deposit: 540.0  
Smith balance after withdrawal: 107.75  
  
72354    Ted Murphy    $132.90  
69713    Jane Smith    $111.52  
93757    Edward Demsey  $785.90
```

The `Account` class, shown in [Listing 4.4](#) , represents a basic bank account. It contains instance data representing the account number, the account's current balance, and the name of the account's owner. Note that instance data can be an object reference variable (not just a primitive type), such as the account owner's name, which is a reference to a `String` object. The interest rate for the account is stored as a constant.

Listing 4.4


```

//*****

// Account.java      Author: Lewis/Loftus
//
// Represents a bank account with basic services such as
deposit
// and withdraw.

//*****

import java.text.NumberFormat;

public class Account
{
    private final double RATE = 0.035;    // interest rate of
3.5%

    private long acctNumber;
    private double balance;
    private String name;

    //-----
    -----
    // Sets up the account by defining its owner, account
number,
    // and initial balance.

    //-----

```

```
-----  
  
    public Account(String owner, long account, double  
initial)  
    {  
        name = owner;  
        acctNumber = account;  
        balance = initial;  
    }
```

```
    //-----  
    -----  
    // Deposits the specified amount into the account. Returns  
the  
    // new balance.
```

```
    //-----  
    -----  
    public double deposit(double amount)  
    {  
        balance = balance + amount;  
        return balance;  
    }
```

```
    //-----  
    -----  
    // Withdraws the specified amount from the account and  
applies  
    // the fee. Returns the new balance.  
    //-----
```

```
-----  
  
    public double withdraw(double amount, double fee)  
    {  
        balance = balance - amount - fee;  
  
        return balance;  
    }
```

```
//-----  
-----  
    // Adds interest to the account and returns the new  
balance.
```

```
//-----  
-----  
    public double addInterest()  
    {  
        balance += (balance * RATE);  
        return balance;  
    }
```

```
//-----  
-----  
    // Returns the current balance of the account.  
//-----
```

```
-----  
    public double getBalance()  
    {  
        return balance;  
    }
```

```

    }

    //-----

    // Returns a one-line description of the account as a
    string.

    //-----

    public String toString()
    {
        NumberFormat fmt = NumberFormat.getCurrencyInstance();

        return acctNumber + "\t" + name + "\t" +
        fmt.format(balance);
    }
}

```

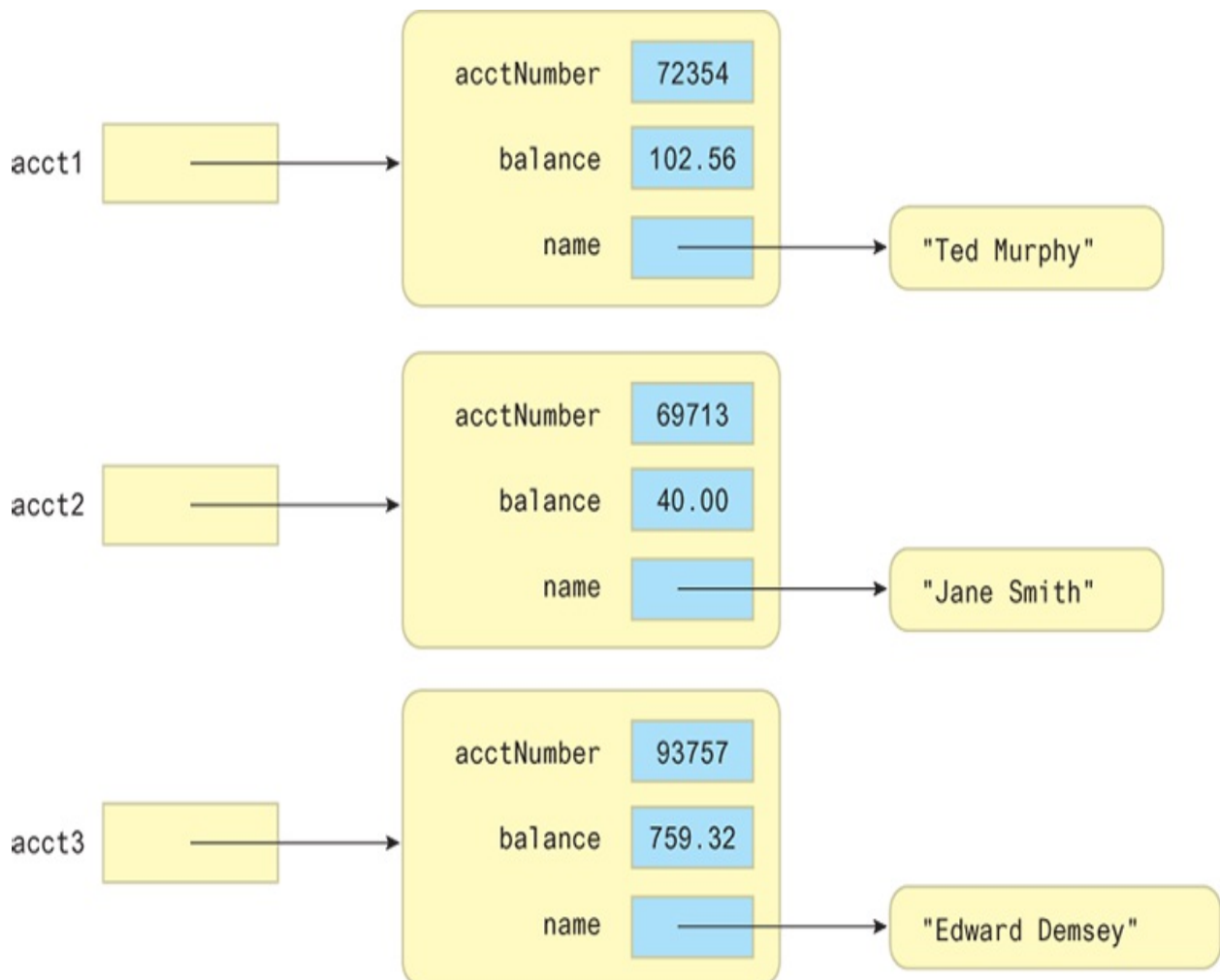


Discussion of the `Account` class.

The constructor of the `Account` class accepts three parameters that are used to initialize the instance data. The `deposit` and `withdraw` methods perform the basic transactions on the account, adjusting the balance based on the parameters. There is also an `addInterest`

method that updates the balance by adding in the interest earned. These methods represent valid ways to change the balance, so a classic mutator such as `setBalance` is not provided.

The status of the three `Account` objects just after they were created in the `Transactions` program could be depicted as follows:



The various methods that update the balance of the account could be more rigorously designed. Checks should be made to ensure that the parameter values are valid, such as preventing the withdrawal of a

negative amount (which would essentially be a deposit). This processing is discussed in the next chapter.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 4.16 Why is a method invoked through (or on) a particular object? What is the exception to that rule?

SR 4.17 What does it mean for a method to return a value?

SR 4.18 What does the `return` statement do?

SR 4.19 Is a `return` statement required?

SR 4.20 Explain the difference between an actual parameter and a formal parameter.

SR 4.21 Write a method called `getFaceDown` for the `Die` class that returns the current “face down” value of the die. *Hint:* On a standard die, the sum of any two opposite faces is seven.

SR 4.22 In the `Transactions` program:

- a. How many `Account` objects are created?
- b. How many arguments (actual parameters) are passed to the `withdraw` method when it is invoked on the `acct2` object?
- c. How many arguments (actual parameters) are passed to the `addInterest` method when it is invoked on the `acct3` object?

SR 4.23 Which of the `Account` class methods would you classify as accessor methods? As mutator methods? As service methods?

4.5 Constructors Revisited

As we stated in [Chapter 2](#), a constructor is similar to a method that is invoked when an object is instantiated. When we define a class, we usually define a constructor to help us set up the class. In particular, we often use a constructor to initialize the variables associated with each object.

A constructor differs from a regular method in two ways. First, the name of a constructor is the same name as the class. Therefore, the name of the constructor in the `Die` class is `Die`, and the name of the constructor in the `Account` class is `Account`. Second, a constructor cannot return a value and does not have a return type specified in the method header.

A common mistake made by programmers is to put a `void` return type on a constructor. As far as the compiler is concerned, putting any return type on a constructor, even `void`, turns it into a regular method that happens to have the same name as the class. As such, it cannot be invoked as a constructor. This leads to error messages that are sometimes difficult to decipher.

Key Concept

A constructor cannot have any return type, even `void`.

Generally, a constructor is used to initialize the newly instantiated object. For instance, the constructor of the `Die` class sets the face value of the die to 1 initially. The constructor of the `Account` class sets the values of the instance variables to the values passed in as parameters to the constructor.

We don't have to define a constructor for every class. Each class has a *default constructor* that takes no parameters. The default constructor is used if we don't provide our own. This default constructor generally has no effect on the newly created object.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 4.24 What are constructors used for?

SR 4.25 How are constructors defined?