



Chapter 6

Methods: A Deeper Look

Java™ How to Program, 8/e



OBJECTIVES

In this Chapter you'll learn:

- How **static** methods and fields are associated with an entire class rather than specific instances of the class.
- To use common **Math** methods available in the Java API.
- To understand the mechanisms for passing information between methods.
- How the method call/return mechanism is supported by the method-call stack and activation records.
- How packages group related classes.
- How to use random-number generation to implement game-playing applications.
- How the visibility of declarations is limited to specific regions of programs.
- What method overloading is and how to create overloaded methods.

6.2 Program Modules in Java

- ▶ Java programs combine new methods and classes that you write with predefined methods and classes available in the [Java Application Programming Interface \(API\)](#) and in other class libraries.
- ▶ Related classes are typically grouped into packages so that they can be imported into programs and reused.
- ▶ While developing programs you may need to use existing methods from classes in Java API.



6.2 Program Modules in Java (Cont.)

- ▶ Methods help you modularize a program by separating its tasks into self-contained units.
- ▶ Statements in method bodies
 - Written only once
 - Hidden from other methods
 - Can be reused from several locations in a program
- ▶ Divide-and-conquer approach
 - Constructing programs from small, simple pieces
- ▶ Software reusability
 - Use existing methods as building blocks to create new programs.
- ▶ Dividing a program into meaningful methods makes the program easier to debug and maintain.



Software Engineering Observation 6.2

To promote software reusability, every method should be limited to performing a single, well-defined task, and the name of the method should express that task effectively.



Error-Prevention Tip 6.1

A method that performs one task is easier to test and debug than one that performs many tasks.



Software Engineering Observation 6.3

If you cannot choose a concise name that expresses a method's task, your method might be attempting to perform too many tasks. Break such a method into several smaller methods.

6.2 Program Modules in Java (Cont.)

- ▶ Hierarchical form of management (Fig. 6.1).
 - A boss (the caller) asks a worker (the called method) to perform a task and report back (return) the results after completing the task.
 - The boss method does not know how the worker method performs its designated tasks.
 - The worker may also call other worker methods, unknown to the boss.
- ▶ “Hiding” of implementation details promotes good software engineering.

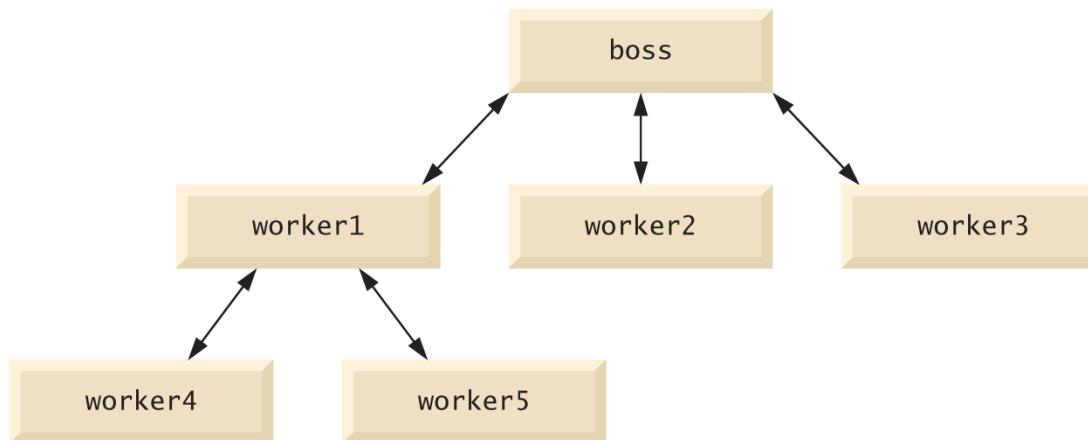


Fig. 6.1 | Hierarchical boss-method/worker-method relationship.



6.3 static Methods, static Fields and Class Math

- ▶ Sometimes a method performs a task that does not depend on the contents of any object.
 - Applies to the class in which it's declared as a whole
 - Known as a **static** method or a **class method**
- ▶ It's common for classes to contain convenient **static** methods to perform common tasks.
- ▶ To declare a method as **static**, place the keyword **static** before the return type in the method's declaration.
- ▶ Calling a **static** method
 - *ClassName.methodName(arguments)*
- ▶ **Class Math** provides a collection of **static** methods that enable you to perform common mathematical calculations (no need to import class Math, it is part of java.lang which is implicitly imported by the compiler).
- ▶ Method arguments may be constants, variables or expressions.



Method	Description	Example
<code>abs(<i>x</i>)</code>	absolute value of <i>x</i>	<code>abs(23.7)</code> is 23.7 <code>abs(0.0)</code> is 0.0 <code>abs(-23.7)</code> is 23.7
<code>ceil(<i>x</i>)</code>	rounds <i>x</i> to the smallest integer not less than <i>x</i>	<code>ceil(9.2)</code> is 10.0 <code>ceil(-9.8)</code> is -9.0
<code>cos(<i>x</i>)</code>	trigonometric cosine of <i>x</i> (<i>x</i> in radians)	<code>cos(0.0)</code> is 1.0
<code>exp(<i>x</i>)</code>	exponential method e^x	<code>exp(1.0)</code> is 2.71828 <code>exp(2.0)</code> is 7.38906
<code>floor(<i>x</i>)</code>	rounds <i>x</i> to the largest integer not greater than <i>x</i>	<code>floor(9.2)</code> is 9.0 <code>floor(-9.8)</code> is -10.0
<code>log(<i>x</i>)</code>	natural logarithm of <i>x</i> (base <i>e</i>)	<code>log(Math.E)</code> is 1.0 <code>log(Math.E * Math.E)</code> is 2.0
<code>max(<i>x</i>, <i>y</i>)</code>	larger value of <i>x</i> and <i>y</i>	<code>max(2.3, 12.7)</code> is 12.7 <code>max(-2.3, -12.7)</code> is -2.3
<code>min(<i>x</i>, <i>y</i>)</code>	smaller value of <i>x</i> and <i>y</i>	<code>min(2.3, 12.7)</code> is 2.3 <code>min(-2.3, -12.7)</code> is -12.7

Fig. 6.2 | Math class methods. (Part 1 of 2.)

Method	Description	Example
<code>pow(<i>x</i>, <i>y</i>)</code>	<i>x</i> raised to the power <i>y</i> (i.e., x^y)	<code>pow(2.0, 7.0)</code> is 128.0 <code>pow(9.0, 0.5)</code> is 3.0
<code>sin(<i>x</i>)</code>	trigonometric sine of <i>x</i> (<i>x</i> in radians)	<code>sin(0.0)</code> is 0.0
<code>sqrt(<i>x</i>)</code>	square root of <i>x</i>	<code>sqrt(900.0)</code> is 30.0
<code>tan(<i>x</i>)</code>	trigonometric tangent of <i>x</i> (<i>x</i> in radians)	<code>tan(0.0)</code> is 0.0

Fig. 6.2 | Math class methods. (Part 2 of 2.)



6.4 static Methods, static Fields and Class Math (Cont.)

- ▶ Math fields for common mathematical constants
 - Math.PI (3.141592653589793)
 - Math.E (2.718281828459045)
- ▶ Declared in class Math with the modifiers **public**, **final** and **static**
 - **public** allows you to use these fields in your own classes.
 - A field declared with keyword **final** is constant—its value cannot change after the field is initialized.
 - PI and E are declared **final** because their values never change.



6.4 static Methods, static Fields and Class Math (Cont.)

- ▶ A field that represents an attribute is also known *as an instance variable*
 - each object (instance) of the class *has a separate instance of the variable* in memory.
- ▶ Fields for which each object of a class *does not have a separate instance of the field* are declared **static** and are also known as **class variables**.
- ▶ All objects of a class containing **static** fields share one copy of those fields.
- ▶ Together the class variables (i.e., **static** variables) and instance variables represent the fields of a class.



6.4 static Methods, static Fields and Class Math (Cont.)

- ▶ Why is method **main** declared **static**?
 - The JVM attempts to invoke the **main** method of the class you specify
 - when no objects of the class have been created.
 - Declaring **main** as **static** allows the JVM to invoke **main** without creating an instance of the class.



6.5 Declaring Methods with Multiple Parameters

- ▶ Multiple parameters are specified as a comma-separated list.
- ▶ There must be one argument in the method call for each parameter (sometimes called a **formal parameter**) in the method declaration.
- ▶ Each argument must be consistent with the type of the corresponding parameter.



```
1 // Fig. 6.3: MaximumFinder.java
2 // Programmer-declared method maximum with three double parameters.
3 import java.util.Scanner;
4
5 public class MaximumFinder
6 {
7     // obtain three floating-point values and locate the maximum value
8     public void determineMaximum()
9     {
10         // create Scanner for input from command window
11         Scanner input = new Scanner( System.in );
12
13         // prompt for and input three floating-point values
14         System.out.print(
15             "Enter three floating-point values separated by spaces: " );
16         double number1 = input.nextDouble(); // read first double
17         double number2 = input.nextDouble(); // read second double
18         double number3 = input.nextDouble(); // read third double
19
20         // determine the maximum value
21         double result = maximum( number1, number2, number3 ); ←
22 }
```

Passing three arguments to method
maximum

Fig. 6.3 | Programmer-declared method `maximum` with three `double` parameters.
(Part 1 of 2.)



```
23     // display maximum value
24     System.out.println( "Maximum is: " + result );
25 } // end method determineMaximum
26
27 // returns the maximum of its three double parameters
28 public double maximum( double x, double y, double z )
29 {
30     double maximumValue = x; // assume x is the largest to start
31
32     // determine whether y is greater than maximumValue
33     if ( y > maximumValue )
34         maximumValue = y;
35
36     // determine whether z is greater than maximumValue
37     if ( z > maximumValue )
38         maximumValue = z;
39
40     return maximumValue;
41 } // end method maximum
42 } // end class MaximumFinder
```

Method `maximum` receives three parameters and returns the largest of the three

Fig. 6.3 | Programmer-declared method `maximum` with three `double` parameters.
(Part 2 of 2.)



```
1 // Fig. 6.4: MaximumFinderTest.java
2 // Application to test class MaximumFinder.
3
4 public class MaximumFinderTest
{
5     // application starting point
6     public static void main( String[] args )
7     {
8         MaximumFinder maximumFinder = new MaximumFinder();
9         maximumFinder.determineMaximum();
10    } // end main
11 } // end class MaximumFinderTest
```

```
Enter three floating-point values separated by spaces: 9.35 2.74 5.1
Maximum is: 9.35
```

```
Enter three floating-point values separated by spaces: 5.8 12.45 8.32
Maximum is: 12.45
```

```
Enter three floating-point values separated by spaces: 6.46 4.12 10.54
Maximum is: 10.54
```

Fig. 6.4 | Application to test class MaximumFinder.



Software Engineering Observation 6.5

Methods can return at most one value, but the returned value could be a reference to an object that contains many values.



Software Engineering Observation 6.6

Variables should be declared as fields of a class only if they are required for use in more than one method of the class or if the program should save their values between calls to the class's methods.



6.5 Declaring Methods with Multiple Parameters (Cont.)

- ▶ Implementing method `maximum` by reusing method `Math.max`
 - Two calls to `Math.max`, as follows:
 - `return Math.max(x, Math.max(y, z));`
 - The first specifies arguments `x` and `Math.max(y, z)`.
 - Before any method can be called, its arguments must be evaluated to determine their values.
 - If an argument is a method call, the method call must be performed to determine its return value.
 - The result of the first call is passed as the second argument to the other call, which returns the larger of its two arguments.



6.5 Declaring Methods with Multiple Parameters (Cont.)

- ▶ **String concatenation**
 - Assemble **String** objects into larger strings with operators + or +=.
- ▶ When both operands of operator + are **Strings**, operator + creates a new **String** object
 - characters of the right operand are placed at the end of those in the left operand
- ▶ Every primitive value and object in Java has a **String** representation.
 - All objects have a **toString** method that returns a **String** representation of the object.
- ▶ When one of the + operator's operands is a **String**, the other is converted to a **String**, then the two are concatenated.
- ▶ If a **boolean** is concatenated with a **String**, the **boolean** is converted to the **String** "true" or "false".



Common Programming Error 6.3

Confusing the `+` operator used for string concatenation with the `+` operator used for addition can lead to strange results. Java evaluates the operands of an operator from left to right. For example, if integer variable `y` has the value 5, the expression `"y + 2 = " + y + 2` results in the string `"y + 2 = 52"`, not `"y + 2 = 7"`, because first the value of `y` (5) is concatenated to the string `"y + 2 = "`, then the value 2 is concatenated to the new larger string `"y + 2 = 5"`. The expression `"y + 2 = " + (y + 2)` produces the desired result `"y + 2 = 7"`.



6.6 Notes on Declaring and Using Methods

- ▶ Three ways to call a method:
 - Using a method name by itself to call another method of the same class
 - Using a variable that contains a reference to an object, followed by a dot (.) and the method name to call a method of the referenced object
 - Using the class name and a dot (.) to call a **static** method of a class



6.6 Notes on Declaring and Using Methods (Cont.)

- ▶ A **non-static** method can call any method of the same class directly and can manipulate any of the class's fields directly.
- ▶ A **static** method can call *only other static methods* of the same class directly and can manipulate *only static fields* in the same class directly.
 - To access the class's non-**static** members, a **static** method must use a reference to an object of the class.



6.6 Notes on Declaring and Using Methods (Cont.)

- ▶ Three ways to return control to the statement that calls a method:
 - When the program flow reaches the method-ending right brace
 - When the following statement executes
`return;`
 - When the method returns a result with a statement like
`return expression;`



Common Programming Error 6.4

Declaring a method outside the body of a class declaration or inside the body of another method is a syntax error.



Common Programming Error 6.5

Omitting the return-value-type, possibly void, in a method declaration is a syntax error.



Common Programming Error 6.6

Placing a semicolon after the right parenthesis enclosing the parameter list of a method declaration is a syntax error.



Common Programming Error 6.7

Redeclaring a parameter as a local variable in the method's body is a compilation error.



Common Programming Error 6.8

Forgetting to return a value from a method that should return a value is a compilation error. If a return type other than void is specified, the method must contain a return statement that returns a value consistent with the method's return type. Returning a value from a method whose return type has been declared void is a compilation error.



6.7 Method-Call Stack and Activation Records

- ▶ **Stack** data structure
 - Analogous to a pile of dishes
 - A dish is placed on the pile at the top (referred to as **pushing** the dish onto the stack).
 - A dish is removed from the pile from the top (referred to as **popping** the dish off the stack).
- ▶ **Last-in, first-out (LIFO)** data structures
 - The last item pushed (inserted) on the stack is the first item popped (removed) from the stack.



6.7 Method-Call Stack and Activation Records (Cont.)

- ▶ When a program calls a method, the called method must know how to return to its caller
 - The return address of the calling method is pushed onto the **program-execution** (or **method-call**) **stack**.
- ▶ If a series of method calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order.
- ▶ The program-execution stack also contains the memory for the local variables used in each invocation of a method during a program's execution.
 - Stored as a portion of the program-execution stack known as the **activation record** or **stack frame** of the method call.



6.7 Method-Call Stack and Activation Records (Cont.)

- ▶ When a method call is made, the activation record for that method call is pushed onto the program-execution stack.
- ▶ When the method returns to its caller, the method's activation record is popped off the stack and those local variables are no longer known to the program.



6.8 Argument Promotion and Casting

- ▶ **Argument promotion**
 - Converting an argument's value, if possible, to the type that the method expects to receive in its corresponding parameter.
- ▶ Conversions may lead to compilation errors if Java's **promotion rules** are not satisfied.
- ▶ **Promotion rules**
 - specify which conversions are allowed.
 - apply to expressions containing values of two or more primitive types and to primitive-type values passed as arguments to methods.
- ▶ Each value is promoted to the “highest” type in the expression.
- ▶ Figure 6.5 lists the primitive types and the types to which each can be promoted.

Type	Valid promotions
double	None
float	double
long	float or double
int	long, float or double
char	int, long, float or double
short	int, long, float or double (but not char)
byte	short, int, long, float or double (but not char)
boolean	None (boolean values are not considered to be numbers in Java)

Fig. 6.5 | Promotions allowed for primitive types.



6.8 Argument Promotion and Casting (Cont.)

- ▶ Converting values to types lower in the table of Fig. 6.5 will result in different values if the lower type cannot represent the value of the higher type
- ▶ In cases where information may be lost due to conversion, the Java compiler requires you to use a cast operator to explicitly force the conversion to occur—otherwise a compilation error occurs.



6.10 Case Study: Random-Number Generation

- ▶ Simulation and game playing
 - element of chance
 - Class `Random` (package `java.util`)
 - static method `random` of class `Math`.
- ▶ Objects of class `Random` can produce random `boolean`, `byte`, `float`, `double`, `int`, `long` values
- ▶ `Math` method `random` can produce only `double` values in the range $0.0 \leq x < 1.0$.
- ▶ Documentation for class `Random`
 - java.sun.com/javase/6/docs/api/java/util/Random.html



6.10 Case Study: Random-Number Generation (Cont.)

- ▶ Class `Random` produces **pseudorandom numbers**
 - means that you can generate the same random sequence if you provide the "key" to that sequence, referred to as the "seed".
 - If you do not "seed" your Random number generator, it is seeded with *some* (usually based on system time) random number by default, and therefore produces the different sequence every time that you run your program.
 - Class `Random` → The calculation uses the current time of day to **seed** the random-number generator.
- ▶ `Random` method `nextInt` that receives an `int` argument returns a value from 0 up to, but not including, the argument's value.



6.10 Case Study: Random-Number Generation (Cont.)

▶ Rolling a Six-Sided Die

- `face = 1 + randomNumbers.nextInt(6);`
- The argument 6—called the **scaling factor**—represents the number of unique values that `nextInt` should produce (0–5)
- This is called **scaling** the range of values
- A six-sided die has the numbers 1–6 on its faces, not 0–5.
- We **shift** the range of numbers produced by adding a **shifting value**—in this case 1—to our previous result



```
1 // Fig. 6.7: RandomIntegers.java
2 // Shifted and scaled random integers.
3 import java.util.Random; // program uses class Random
4
5 public class RandomIntegers
6 {
7     public static void main( String[] args )
8     {
9         Random randomNumbers = new Random(); // random number generator
10        int face; // stores each random integer generated
11
12        // loop 20 times
13        for ( int counter = 1; counter <= 20; counter++ )
14        {
15            // pick random integer from 1 to 6
16            face = 1 + randomNumbers.nextInt( 6 );
17
18            System.out.printf( "%d ", face ); // display generated value
19
20            // if counter is divisible by 5, start a new line of output
21            if ( counter % 5 == 0 )
22                System.out.println();
23        } // end for
24    } // end main
25 } // end class RandomIntegers
```

Program uses class Random from package java.util

Creates a Random object

Produces integers in the range 1 through 6

Fig. 6.7 | Shifted and scaled random integers. (Part I of 2.)



The image could be displayed. Your computer may not have enough memory to open the image, or the image may have been corrupted. Reload your browser, and then open the file again. If the file is still damaged, you may need to delete the image and find another.



6.10 Case Study: Random-Number Generation (Cont.)

- ▶ Fig 6.8: Rolling a Six-Sided Die 6000 Times
- ▶ Count number of times 1s, 2s, ...6s encountered



```
1 // Fig. 6.8: RollDie.java
2 // Roll a six-sided die 6000 times.
3 import java.util.Random;
4
5 public class RollDie
6 {
7     public static void main( String[] args )
8     {
9         Random randomNumbers = new Random(); // random number generator
10
11     int frequency1 = 0; // maintains count of 1s rolled
12     int frequency2 = 0; // count of 2s rolled
13     int frequency3 = 0; // count of 3s rolled
14     int frequency4 = 0; // count of 4s rolled
15     int frequency5 = 0; // count of 5s rolled
16     int frequency6 = 0; // count of 6s rolled
17
18     int face; // stores most recently rolled value
19
20     // tally counts for 6000 rolls of a die
21     for ( int roll = 1; roll <= 6000; roll++ )
22     {
23         face = 1 + randomNumbers.nextInt( 6 ); // number from 1 to 6
24     }
}
```

Fig. 6.8 | Rolling a six-sided die 6000 times. (Part 1 of 3.)



```
25 // determine roll value 1-6 and increment appropriate counter
26 switch ( face ) ←
27 {
28     case 1:
29         ++frequency1; // increment the 1s counter
30         break;
31     case 2:
32         ++frequency2; // increment the 2s counter
33         break;
34     case 3:
35         ++frequency3; // increment the 3s counter
36         break;
37     case 4:
38         ++frequency4; // increment the 4s counter
39         break;
40     case 5:
41         ++frequency5; // increment the 5s counter
42         break;
43     case 6:
44         ++frequency6; // increment the 6s counter
45         break; // optional at end of switch
46     } // end switch
47 } // end for
```

Value from 1 through 6 used to update appropriate counter

Fig. 6.8 | Rolling a six-sided die 6000 times. (Part 2 of 3.)



```
49     System.out.println( "Face\tFrequency" ); // output headers
50     System.out.printf( "1\t%d\n2\t%d\n3\t%d\n4\t%d\n5\t%d\n6\t%d\n",
51                     frequency1, frequency2, frequency3, frequency4,
52                     frequency5, frequency6 );
53 } // end main
54 } // end class RollDie
```

Face	Frequency
1	982
2	1001
3	1015
4	1005
5	1009
6	988

Face	Frequency
1	1029
2	994
3	1017
4	1007
5	972
6	981

Fig. 6.8 | Rolling a six-sided die 6000 times. (Part 3 of 3.)



6.10.1 Generalized Scaling and Shifting of Random Numbers

- ▶ Generalize the scaling and shifting of random numbers:

```
number = shiftingValue +  
randomNumbers.nextInt(scalingFactor);
```

where *shiftingValue* specifies the first number in the desired range of consecutive integers and *scalingFactor* specifies how many numbers are in the range.



6.10.2 Random-Number Repeatability for Testing and Debugging

- ▶ When debugging an application, it's sometimes useful to repeat the exact same sequence of pseudorandom numbers.
- ▶ To do so, create a **Random** object as follows:
 - `Random randomNumbers =
 new Random(seedvalue);`
 - **seedvalue** (of type **long**) seeds the random-number calculation.
- ▶ You can set a **Random** object's seed at any time during program execution by calling the object's **set** method.



Error-Prevention Tip 6.2

While developing a program, create the Random object with a specific seed value to produce a repeatable sequence of numbers each time the program executes. If a logic error occurs, fix the error and test the program again with the same seed value—this allows you to reconstruct the same sequence of numbers that caused the error. Once the logic errors have been removed, create the Random object without using a seed value, causing the Random object to generate a new sequence of random numbers each time the program executes.

ENUM TYPES

- ▶ An *enum type* is a **special data type** that enables for **a variable** to be a set of **predefined constants**.
- ▶ The **variable must be equal to one of the values** that have been predefined for it.
- ▶ Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.



ENUM TYPES

- ▶ Because they are constants, the names of an enum type's fields are in uppercase letters.
- ▶ In the Java programming language, you define an enum type by using the enum keyword.
- ▶ For example, you would specify a days-of-the-week enum type as:

```
public enum Day { SUNDAY, MONDAY, TUESDAY,  
    WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }
```

ENUM TYPES

- ▶ You should use enum types any time you need to represent a fixed set of constants.
- ▶ That includes natural enum types such as
 - the planets in our solar system and
 - data sets where you know all possible values at compile time
 - for example, the choices on a menu, command line flags, and so on.
 - Example from Eclipse!



ENUM TYPES

- ▶ Java programming language enum types are much more powerful than their counterparts in other languages.
- ▶ The enum declaration defines a *class* (called an *enum type*).
- ▶ The enum class body can include methods and other fields.
- ▶ The compiler automatically adds some special methods when it creates an enum.
- ▶ For example, they have a static values method that returns an array containing all of the values of the enum in the order they are declared.
- ▶ The values() method returns an array that contains a list of the enumeration constants.
- ▶ For example, this code from the Day class example below iterates over all the days.
 - ```
for (int i= 0; i < Day.values().length; i++)
{
 System.out.printf(Day.values()[i]);
}
```



## 6.11 Case Study: A Game of Chance; Introducing Enumerations

### ► Basic rules for the dice game Craps:

- You roll two dice. Each die has six faces, which contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, you win. If the sum is 2, 3 or 12 on the first throw (called “craps”), you lose (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes your “point.” To win, you must continue rolling the dice until you “make your point” (i.e., roll that same point value). You lose by rolling a 7 before making your point.



```
1 // Fig. 6.9: Craps.java
2 // Craps class simulates the dice game craps.
3 import java.util.Random;
4
5 public class Craps
6 {
7 // create random number generator for use in method rollDice
8 private static final Random randomNumbers = new Random();
9
10 // enumeration with constants that represent the game status
11 private enum Status { CONTINUE, WON, LOST };
12
13 // constants that represent common rolls of the dice
14 private static final int SNAKE_EYES = 2;
15 private static final int TREY = 3;
16 private static final int SEVEN = 7;
17 private static final int YO_LEVEN = 11;
18 private static final int BOX_CARS = 12;
19}
```

Declares constants for the game status

Declares constants representing common rolls of the dice

**Fig. 6.9** | Craps class simulates the dice game craps. (Part 1 of 4.)



```
20 // plays one game of craps
21 public void play()
22 {
23 int myPoint = 0; // point if no win or loss on first roll
24 Status gameStatus; // can contain CONTINUE, WON or LOST
25
26 int sumOfDice = rollDice(); // first roll of the dice
27
28 // determine game status and point based on first roll
29 switch (sumOfDice)
30 {
31 case SEVEN: // win with 7 on first roll
32 case YO_LEVEN: // win with 11 on first roll
33 gameStatus = Status.WON;
34 break;
35 case SNAKE_EYES: // lose with 2 on first roll
36 case TREY: // lose with 3 on first roll
37 case BOX_CARS: // lose with 12 on first roll
38 gameStatus = Status.LOST;
39 break;

```

Variable that stores the game status

Roll the dice to start playing the game

Player wins on the first roll; set gameStatus to WON

Player loses on the first roll; set gameStatus to LOST

**Fig. 6.9** | Craps class simulates the dice game craps. (Part 2 of 4.)



```
40 default: // did not win or lose, so remember point
41 gameStatus = Status.CONTINUE; // game is not over
42 myPoint = sumOfDice; // remember the point
43 System.out.printf("Point is %d\n", myPoint);
44 break; // optional at end of switch
45 } // end switch
46
47 // while game is not complete
48 while (gameStatus == Status.CONTINUE) // not WON or LOST
49 {
50 sumOfDice = rollDice(); // roll dice again
51
52 // determine game status
53 if (sumOfDice == myPoint) // win by making point
54 gameStatus = Status.WON;
55 else
56 if (sumOfDice == SEVEN) // lose by rolling 7 before point
57 gameStatus = Status.LOST;
58 } // end while
59
```

Player did not win or lose; set gameStatus to CONTINUE

Loop while game is not over

Roll the dice again

Made your point; set gameStatus to WON

Rolled 7; set gameStatus to WON

**Fig. 6.9** | Craps class simulates the dice game craps. (Part 3 of 4.)



```
60 // display won or lost message
61 if (gameStatus == Status.WON)←
62 System.out.println("Player wins");
63 else
64 System.out.println("Player loses");
65 } // end method play
66
67 // roll dice, calculate sum and display results
68 public int rollDice()
69 {
70 // pick random die values
71 int die1 = 1 + randomNumbers.nextInt(6); // first die roll
72 int die2 = 1 + randomNumbers.nextInt(6); // second die roll
73
74 int sum = die1 + die2; // sum of die values
75
76 // display results of this roll
77 System.out.printf("Player rolled %d + %d = %d\n",
78 die1, die2, sum);
79
80 return sum; // return sum of dice
81 } // end method rollDice
82 } // end class Craps
```

Display a message indicating whether the user won or lost

**Fig. 6.9** | Craps class simulates the dice game craps. (Part 4 of 4.)



```
1 // Fig. 6.10: CrapsTest.java
2 // Application to test class Craps.
3
4 public class CrapsTest
{
5 public static void main(String[] args)
6 {
7 Craps game = new Craps();
8 game.play(); // play one game of craps
9 } // end main
10 } // end class CrapsTest
```

```
Player rolled 5 + 6 = 11
Player wins
```

```
Player rolled 5 + 4 = 9
Point is 9
Player rolled 2 + 2 = 4
Player rolled 2 + 6 = 8
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
Player wins
```

**Fig. 6.10** | Application to test class Craps. (Part 1 of 2.)



```
Player rolled 1 + 2 = 3
Player loses
```

```
Player rolled 2 + 6 = 8
Point is 8
Player rolled 5 + 1 = 6
Player rolled 2 + 1 = 3
Player rolled 1 + 6 = 7
Player loses
```

**Fig. 6.10** | Application to test class Craps. (Part 2 of 2.)



## 6.11 Case Study: A Game of Chance; Introducing Enumerations (Cont.)

- ▶ Notes:
  - **myPoint** is initialized to 0 to ensure that the application will compile.
  - If you do not initialize **myPoint**, the compiler issues an error, because **myPoint** is not assigned a value in every **case** of the **switch** statement, and thus the program could try to use **myPoint** before it is assigned a value.
  - **gameStatus** is assigned a value in every **case** of the **switch** statement—thus, it's guaranteed to be initialized before it's used and does not need to be initialized.



## 6.11 Case Study: A Game of Chance; Introducing Enumerations (Cont.)

### ► **enum type Status**

- An **enumeration** in its simplest form declares a set of constants represented by identifiers.
- Special kind of class that is introduced by the keyword **enum** and a type name.
- Braces delimit an **enum** declaration's body.
- Inside the braces is a comma-separated list of **enumeration constants**, each representing a unique value.
- The identifiers in an **enum** must be unique.
- Variables of an **enum** type can be assigned only the constants declared in the enumeration.



## Good Programming Practice 6.1

*Use only uppercase letters in the names of enumeration constants. This makes the constants stand out and reminds you that enumeration constants are not variables.*



## Good Programming Practice 6.2

*Using enumeration constants (like Status.WON, Status.LOST and Status.CONTINUE) rather than literal values (such as 0, 1 and 2) makes programs easier to read and maintain.*

## 6.12 Scope of Declarations

- ▶ Declarations introduce names that can be used to refer to such Java entities.
- ▶ The **scope** of a declaration is the portion of the program that can refer to the declared entity by its name.
  - Such an entity is said to be “in scope” for that portion of the program.



## 6.12 Scope of Declarations (Cont.)

- ▶ Basic scope rules:
  - The scope of a parameter declaration is the body of the method in which the declaration appears.
  - The scope of a local-variable declaration is from the point at which the declaration appears to the end of that block.
  - The scope of a local-variable declaration that appears in the initialization section of a **for** statement's header is the body of the **for** statement and the other expressions in the header.
  - A method or field's scope is the entire body of the class.
- ▶ Any block may contain variable declarations.
- ▶ If a local variable or parameter in a method has the same name as a field of the class, the field is “hidden” until the block terminates execution—this is called **shadowing**.



## Common Programming Error 6.10

*A compilation error occurs when a local variable is declared more than once in a method.*

4.10



### Error-Prevention Tip 6.3

*Use different names for fields and local variables to help prevent subtle logic errors that occur when a method is called and a local variable of the method shadows a field in the class.*



```
1 // Fig. 6.11: Scope.java
2 // Scope class demonstrates field and local variable scopes.
3
4 public class Scope
{
5
6 // field that is accessible to all methods of this class
7 private int x = 1;
8
9 // method begin creates and initializes local variable x
10 // and calls methods useLocalVariable and useField
11 public void begin()
12 {
13 int x = 5; // method's local variable x shadows field x
14
15 System.out.printf("local x in method begin is %d\n", x);
16
17 useLocalVariable(); // useLocalVariable has local x
18 useField(); // useField uses class Scope's field x
19 useLocalVariable(); // useLocalVariable reinitializes local x
20 useField(); // class Scope's field x retains its value
21
22 System.out.printf("\nlocal x in method begin is %d\n", x);
23 } // end method begin
```

Class scope

Method scope

**Fig. 6.11** | Scope class demonstrating scopes of a field and local variables. (Part I of 2.)



```
1 // create and initialize local variable x during each call
2 public void useLocalVariable()
3 {
4 int x = 25; // initialized each time useLocalVariable is called ← Method scope
5
6
7 System.out.printf(
8 "\nlocal x on entering method useLocalVariable is %d\n", x);
9 ++x; // modifies this method's local variable x
10 System.out.printf(
11 "local x before exiting method useLocalVariable is %d\n", x);
12 } // end method useLocalVariable
13
14 // modify class Scope's field x during each call
15 public void useField()
16 {
17 System.out.printf(
18 "\nfield x on entering method useField is %d\n", x);
19 x *= 10; // modifies class Scope's field x ← Uses instance variable x
20 System.out.printf(
21 "field x before exiting method useField is %d\n", x);
22 } // end method useField
23 } // end class Scope
```

**Fig. 6.11** | Scope class demonstrating scopes of a field and local variables. (Part 2 of 2.)

---

```
1 Fig. 6.12: ScopeTest.java
2 // Application to test class Scope.
3
4 public class ScopeTest
{
5 // application starting point
6 public static void main(String[] args)
7 {
8 Scope testScope = new Scope();
9 testScope.begin();
10 } // end main
11 } // end class ScopeTest
```

---

**Fig. 6.12** | Application to test class Scope. (Part 1 of 2.)

```
local x in method begin is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in method begin is 5
```

**Fig. 6.12** | Application to test class Scope. (Part 2 of 2.)

# 6.13 Method Overloading

- ▶ **Method overloading**
  - Methods of the same name declared in the same class
  - Must have different sets of parameters
- ▶ Compiler selects the appropriate method to call by examining the number, types and order of the arguments in the call.
- ▶ Used to create several methods with the same name that perform the same or similar tasks, but on different types or different numbers of arguments.
- ▶ For the example in the next slides:
  - Literal integer values are treated as type `int`, so the method call in line 9 invokes the version of `square` that specifies an `int` parameter.
  - Literal floating-point values are treated as type `double`, so the method call in line 10 invokes the version of `square` that specifies a `double` parameter.

```
1 // Fig. 6.13: MethodOverload.java
2 // Overloaded method declarations.
3
4 public class MethodOverload
{
5
6 // test overloaded square methods
7 public void testOverloadedMethods()
8 {
9 System.out.printf("Square of integer 7 is %d\n", square(7));
10 System.out.printf("Square of double 7.5 is %f\n", square(7.5));
11 } // end method testOverloadedMethods
12
13 // square method with int argument
14 public int square(int intValue) ←
15 {
16 System.out.printf("\nCalled square with int argument: %d\n",
17 intValue);
18 return intValue * intValue;
19 } // end method square with int argument
20}
```

Annotations:

- Calls square with an int parameter
- Calls square with a double parameter
- square method that receives an int

**Fig. 6.13** | Overloaded method declarations. (Part I of 2.)



```
21 // square method with double argument
22 public double square(double doubleValue) ←
23 {
24 System.out.printf("\nCalled square with double argument: %f\n",
25 doubleValue);
26 return doubleValue * doubleValue;
27 } // end method square with double argument
28 } // end class MethodOverload
```

square method that receives a double

**Fig. 6.13** | Overloaded method declarations. (Part 2 of 2.)



```
1 // Fig. 6.14: MethodOverloadTest.java
2 // Application to test class MethodOverload.
3
4 public class MethodOverloadTest
{
5 public static void main(String[] args)
6 {
7 MethodOverload methodOverload = new MethodOverload();
8 methodOverload.testOverloadedMethods();
9 } // end main
10 } // end class MethodOverloadTest
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

**Fig. 6.14** | Application to test class MethodOverload.

## 6.14 Method Overloading

- ▶ Distinguishing Between Overloaded Methods
  - The compiler distinguishes overloaded methods by their **signatures**—the methods' names and the number, types and order of their parameters.
- ▶ Return types of overloaded methods
  - *Method calls cannot be distinguished by return type.*
- ▶ Figure 6.15 illustrates the errors generated when two methods have the same signature and different return types.
- ▶ Overloaded methods can have different return types if the methods have different parameter lists.
- ▶ Overloaded methods need not have the same number of parameters.



## Common Programming Error 6.11

*Declaring overloaded methods with identical parameter lists is a compilation error regardless of whether the return types are different.*

```
1 // Fig. 6.15: MethodOverloadError.java
2 // Overloaded methods with identical signatures
3 // cause compilation errors, even if return types are different.
4
5 public class MethodOverloadError
6 {
7 // declaration of method square with int argument
8 public int square(int x)
9 {
10 return x * x;
11 }
12
13 // second declaration of method square with int argument
14 // causes compilation error even though return types are different
15 public double square(int y) ←
16 {
17 return y * y;
18 }
19 } // end class MethodOverloadError
```

Generates a  
compilation error

**Fig. 6.15** | Overloaded method declarations with identical signatures cause compilation errors, even if the return types are different. (Part 1 of 2.)

```
MethodOverloadError.java:15: square(int) is already defined in
MethodOverloadError
 public double square(int y)
 ^
1 error
```

**Fig. 6.15** | Overloaded method declarations with identical signatures cause compilation errors, even if the return types are different. (Part 2 of 2.)