Methods: A Deeper Look

OBJECTIVES

In this chapter you'll learn:

- How `static` methods and fields are associated with classes rather than objects.
- How the method call/return mechanism is supported by the method-call stack.
- 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.

6.1 Introduction

- Best way to develop and maintain a large program is to construct it from small, simple pieces, or modules.
  - divide and conquer.
- Topics in this chapter
  - `static` methods
  - Declare a method with more than one parameter
  - Method-call stack
  - Simulation techniques with random-number generation.
  - How to declare values that cannot change (i.e., constants) in your programs.
  - Method overloading.

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 and in other class libraries.
- Related classes are typically grouped into packages so that they can be imported into programs and reused.
  - You'll learn how to group your own classes into packages in Chapter 8.

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 de-bug than one that performs many tasks.

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, unbeknown to the boss.
- “Hiding” of implementation details promotes good software engineering.

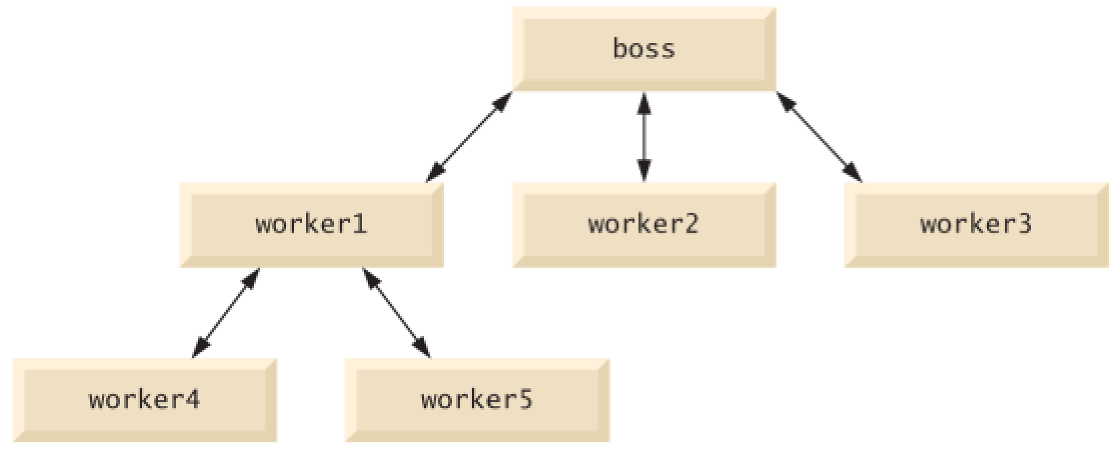


Fig. 6.1 | Hierarchical boss-method/worker-method relationship.

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.
- Method arguments may be constants, variables or expressions.

Software Engineering Observation 6.4

Class `Math` *is part of the* `java.lang` *package, which is implicitly imported by the compiler, so it's not necessary to import class* `Math` *to use its methods.*

Method	Description	Example

`abs( x )`	absolute value of *x*	`abs( 23.7 )` is `23.7`
		`abs( 0.0 )` is `0.0`
		`abs( -23.7 )` is `23.7`

`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0`
		`ceil( -9.8 )` is `-9.0`

`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`

`exp( x )`	exponential method e^x	`exp( 1.0 )` is `2.71828`
		`exp( 2.0 )` is `7.38906`

`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0`
		`floor( -9.8 )` is `-10.0`

`log( x )`	natural logarithm of *x* (base *e*)	`log( Math.E )` is `1.0`
		`log( Math.E * Math.E )` is `2.0`

`max( x, y )`	larger value of *x* and *y*	`max( 2.3, 12.7 )` is `12.7`
		`max( -2.3, -12.7 )` is `-2.3`

`min( x, y )`	smaller value of *x* and *y*	`min( 2.3, 12.7 )` is `2.3`
		`min( -2.3, -12.7 )` is `-12.7`

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

Method	Description	Example

`pow(` *x*`,` *y* `)`	*x* raised to the power *y* (i.e., x^y)	`pow( 2.0, 7.0 )` is `128.0`
		`pow( 9.0, 0.5 )` is `3.0`

`sin(` *x* `)`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0.0`
`sqrt(` *x* `)`	square root of *x*	`sqrt( 900.0 )` is `30.0`
`tan(` *x* `)`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0.0`

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

6.3 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.

6.3 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.

6.5 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

6.3 `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.

Why static methods cannot call non-static methods?

Can we allow non-static methods to call static methods?

6.5 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.

6.5 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;
    ```

6.6 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.

6.6 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.

6.6 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.
- If more method calls occur than can have their activation records stored on the program-execution stack, an error known as a stack overflow occurs.

6.7 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 6.4 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. 6.4 | Promotions allowed for primitive types.

6.7 Argument Promotion and Casting (Cont.)

- Converting values to types lower in the table of Fig. 6.4 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.

6.9 Case Study: Random-Number Generation

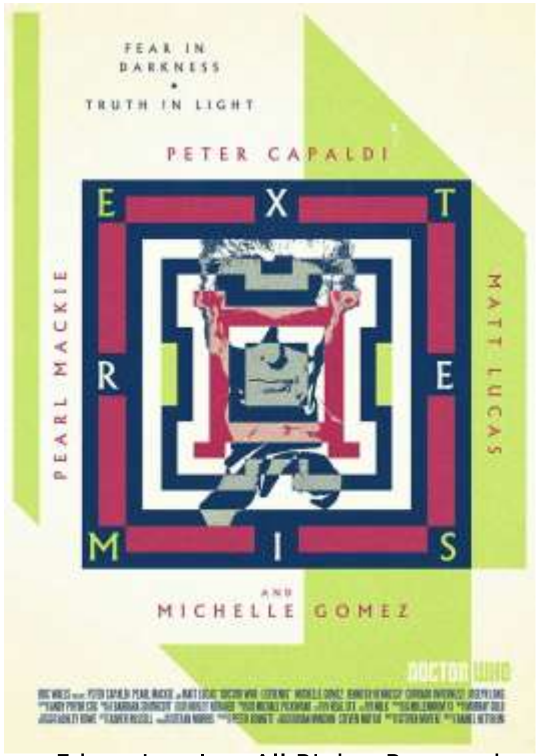
- Simulation and game playing
  - element of chance
  - Class `Random` (package `java.util`)
  - `static` method `random` of class `Math`.
- Objects of class `Random` can produce random `boolean`, `byte`, `float`, `double`, `int`, `long` and Gaussian values
- `Math` method `random` can produce only `double` values in the range `0.0` $\leq x <$ `1.0`.

```
static double    random()
```

Returns a `double` value with a positive sign, greater than or equal to `0.0` and less than `1.0`.

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

- Class `Random` produces pseudorandom numbers
  - A sequence of values produced by a complex mathematical calculation.
  - The calculation uses the current time of day to seed the random-number generator.



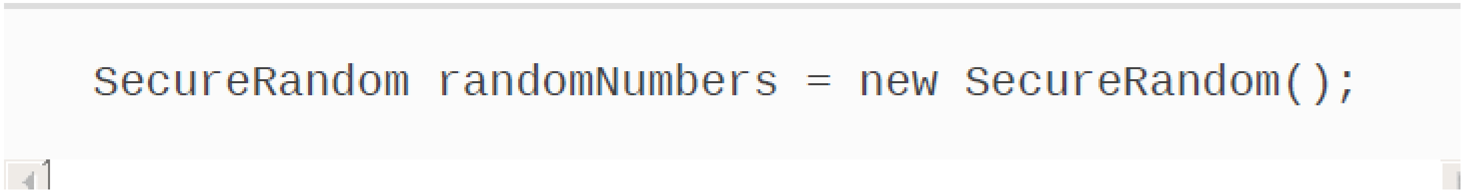
Secure Random

Java’s Random class produced *deterministic* values that could be *predicted* by malicious programmers.

Secure-Random objects produce **nondeterministic random numbers** that *cannot* be predicted.

Creating a SecureRandom Object

A new secure random-number generator object can be created as follows:



```
SecureRandom randomNumbers = new SecureRandom();
```

Obtaining a Random int Value

Consider the following statement:

```
int randomValue = randomNumbers.nextInt();
```

All 2^{32} possible `int` values are produced with (approximately) equal probability

`SecureRandom` method `nextInt` generates a random `int` value. If it truly produces values *at random*, then every value in the range should have an *equal chance* (or probability) of being chosen each time `nextInt` is called.

Changing the Range of Values Produced By nextInt

- Class SecureRandom provides another version of method nextInt that receives an int argument and returns a value from 0 up to, but not including, the argument's value.
- For example, for coin tossing, the following statement returns 0 or 1.

```
int randomValue = randomNumbers.nextInt(2);
```

SecureRandom Performance

Using SecureRandom instead of Random to achieve higher levels of security incurs a significant performance penalty. For “casual” applications, you might want to use class Random from package **java.util**—simply replace SecureRandom with **Random**.

Generalized Scaling and Shifting of Random Numbers

```
int face = 1 + randomNumbers.nextInt(6);
```

The width of the range is determined by the number 6 that’s passed as an argument to SecureRandom method nextInt, and the starting number of the range is the number 1 that’s added to randomNumbers.nextInt(6). We can generalize this result as

int number = *shiftingValue* + randomNumbers.nextInt(*scalingFactor)*

It’s also possible to choose integers at random from sets of values other than ranges of consecutive integers. For example, to obtain a random value from the sequence 2, 5, 8, 11 and 14, you could use the statement

```
        int number = 2 + 3 * randomNumbers.nextInt(5);
```

In general,

        int number = *shiftingValue* +
        differenceBetweenValues * randomNumbers.nextInt(*scalingFactor*)

6.9.2 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.

Void setSeed(long seed)

6.10 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.*

```java
// Fig. 6.8: Craps.java
// Craps class simulates the dice game craps.
import java.util.Random;

public class Craps
{
   // create random number generator for use in method rollDice
   private static final Random randomNumbers = new Random();

   // enumeration with constants that represent the game status
   private enum Status { CONTINUE, WON, LOST };

   // constants that represent common rolls of the dice
   private static final int SNAKE_EYES = 2;
   private static final int TREY = 3;
   private static final int SEVEN = 7;
   private static final int YO_LEVEN = 11;
   private static final int BOX_CARS = 12;

```

Fig. 6.8 | Craps class simulates the dice game craps. (Part 1 of 5.)

```java
      // plays one game of craps
      public static void main( String[] args )
      {
         int myPoint = 0; // point if no win or loss on first roll
         Status gameStatus; // can contain CONTINUE, WON or LOST

         int sumOfDice = rollDice(); // first roll of the dice

         // determine game status and point based on first roll
         switch ( sumOfDice )
         {
            case SEVEN: // win with 7 on first roll
            case YO_LEVEN: // win with 11 on first roll
               gameStatus = Status.WON;
               break;
            case SNAKE_EYES: // lose with 2 on first roll
            case TREY: // lose with 3 on first roll
            case BOX_CARS: // lose with 12 on first roll
               gameStatus = Status.LOST;
               break;
```

Fig. 6.8 | Craps class simulates the dice game craps. (Part 2 of 5.)

```java
         default: // did not win or lose, so remember point
            gameStatus = Status.CONTINUE; // game is not over
            myPoint = sumOfDice; // remember the point
            System.out.printf( "Point is %d\n", myPoint );
            break; // optional at end of switch
      } // end switch

      // while game is not complete
      while ( gameStatus == Status.CONTINUE ) // not WON or LOST
      {
         sumOfDice = rollDice(); // roll dice again

         // determine game status
         if ( sumOfDice == myPoint ) // win by making point
            gameStatus = Status.WON;
         else
            if ( sumOfDice == SEVEN ) // lose by rolling 7 before point
               gameStatus = Status.LOST;
      } // end while

```

Fig. 6.8 | Craps class simulates the dice game craps. (Part 3 of 5.)

```java
      // display won or lost message
      if ( gameStatus == Status.WON )
         System.out.println( "Player wins" );
      else
         System.out.println( "Player loses" );
   } // end main

   // roll dice, calculate sum and display results
   public static int rollDice()
   {
      // pick random die values
      int die1 = 1 + randomNumbers.nextInt( 6 ); // first die roll
      int die2 = 1 + randomNumbers.nextInt( 6 ); // second die roll

      int sum = die1 + die2; // sum of die values

      // display results of this roll
      System.out.printf( "Player rolled %d + %d = %d\n",
         die1, die2, sum );

      return sum; // return sum of dice
   } // end method rollDice
} // end class Craps
```

Fig. 6.8 | Craps class simulates the dice game craps. (Part 4 of 5.)

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

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

```
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
```

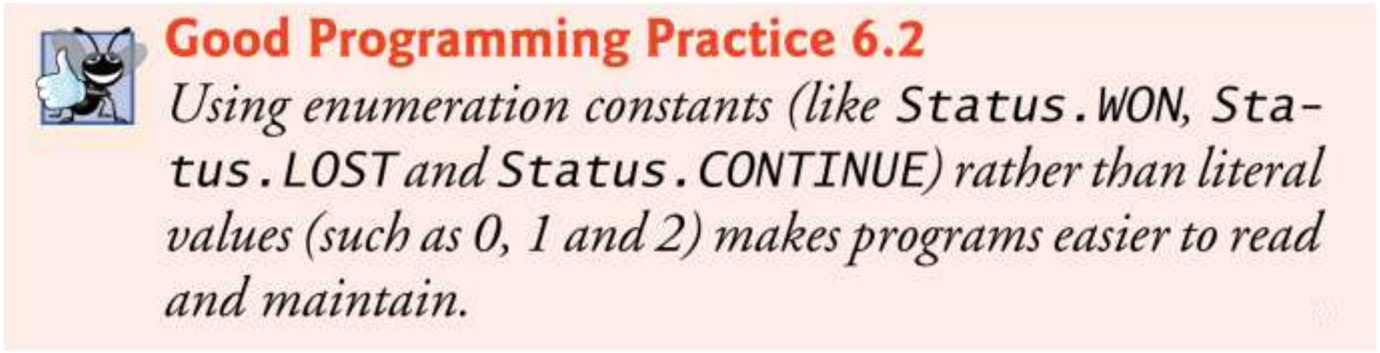
Fig. 6.8 | Craps class simulates the dice game craps. (Part 5 of 5.)

6.10 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.

6.10 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 6.2

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.*

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

- Why Some Constants Are Not Defined as `enum` Constants
  - Doing so would prevent us from using sumOfDice as the switch statement's controlling expression, because Java does not allow you to compare an int to an enum constant.
  - Java does not provide an easy way to convert an `int` value to a particular `enum` constant.
  - Translating an `int` into an `enum` constant could be done with a separate `switch` statement.
  - This would be cumbersome and not improve the readability of the program (thus defeating the purpose of using an `enum`).

6.11 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.

6.11 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.

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.

```java
// Fig. 6.9: Scope.java
// Scope class demonstrates field and local variable scopes.

public class Scope
{
   // field that is accessible to all methods of this class
   private static int x = 1;

   // method main creates and initializes local variable x
   // and calls methods useLocalVariable and useField
   public static void main( String[] args )
   {
      int x = 5; // method's local variable x shadows field x

      System.out.printf( "local x in main is %d\n", x );

      useLocalVariable(); // useLocalVariable has local x
      useField(); // useField uses class Scope's field x
      useLocalVariable(); // useLocalVariable reinitializes local x
      useField(); // class Scope's field x retains its value

      System.out.printf( "\nlocal x in main is %d\n", x );
   } // end main
```

Fig. 6.9 | Scope class demonstrates field and local variable scopes. (Part 1 of 3.)

```java

   // create and initialize local variable x during each call
   public static void useLocalVariable()
   {
      int x = 25; // initialized each time useLocalVariable is called

      System.out.printf(
         "\nlocal x on entering method useLocalVariable is %d\n", x );
      ++x; // modifies this method's local variable x
      System.out.printf(
         "local x before exiting method useLocalVariable is %d\n", x );
   } // end method useLocalVariable

   // modify class Scope's field x during each call
   public static void useField()
   {
      System.out.printf(
         "\nfield x on entering method useField is %d\n", x );
      x *= 10; // modifies class Scope's field x
      System.out.printf(
         "field x before exiting method useField is %d\n", x );
   } // end method useField
} // end class Scope
```

Fig. 6.9 | Scope class demonstrates field and local variable scopes. (Part 2 of 3.)

```
local x in main 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 main is 5
```

Fig. 6.9 | Scope class demonstrates field and local variable scopes. (Part 3 of 3.)

6.12 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.

```java
 1  // Fig. 6.10: MethodOverload.java
 2  // Overloaded method declarations.
 3
 4  public class MethodOverload
 5  {
 6     // test overloaded square methods
 7     public static void main( String[] args )
 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 main
12
13     // square method with int argument
14     public static 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
```

Fig. 6.10 | Overloaded method declarations. (Part 1 of 2.)

- 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.

```java
   // square method with double argument
   public static double square( double doubleValue )
   {
      System.out.printf( "\nCalled square with double argument: %f\n",
         doubleValue );
      return doubleValue * doubleValue;
   } // end method square with double argument
} // end class MethodOverload
```

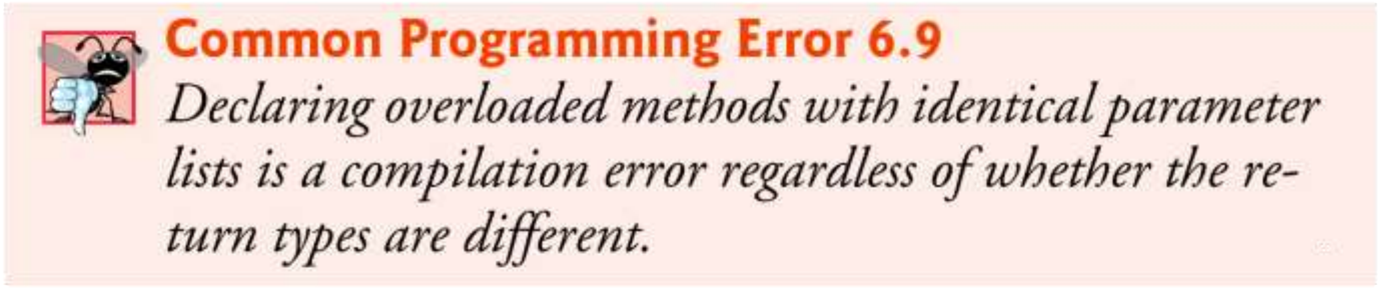
```
Called square with int argument: 7
Square of integer 7 is 49

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

Fig. 6.10 | Overloaded method declarations. (Part 2 of 2.)

6.12 Method Overloading (cont.)

- 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.*
- 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 6.9

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