Methods In Java

Java How to Program,
Late Objects Version, 10/e

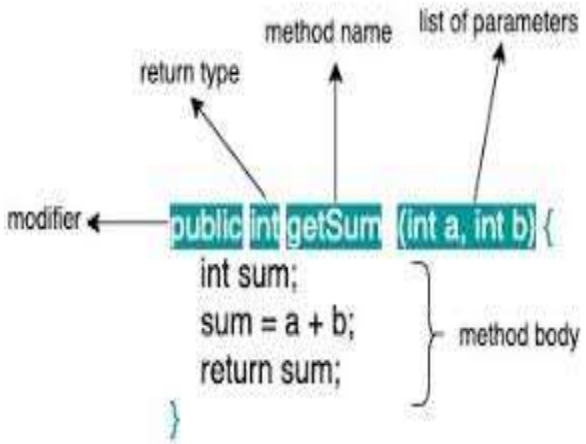
References & Reading

- The content is mainly selected (sometimes modified) from the original slides provided by the authors of the textbook

- Readings
  - Chapter 5: Methods
  - Chapter 18: Recursion

Outline

- Introduction
- Program Modules in Java
- `static` Methods
- `static` Fields
- Local variables
- Declaring Methods
- Notes on Declaring and Using Methods
- Argument Promotion and Casting
- Case Study: Random-Number Generation
- Scope of Declarations
- Method Overloading
- Method-Call Stack and Activation Records
- Argument Promotion and Casting
- Case Study: Random-Number Generation
- Scope of Declarations
- Method Overloading
- Recursion Concepts
- Example Using Recursion: Factorials
- Recursion and the Method-Call Stack
- Recursion vs. Iteration
- `Passing arrays`



Introduction

- You write Java programs by combining new methods and classes that you write with predefined available in the Java Application Programming Interface (or the Java class library) and in various other class libraries

- Example of predefined classes and methods:
  - Classes: `System`, `Scanner`, `String`, etc.
  - Methods: `print()`, `println()`, `printf()`, `nextInt()`, etc.

5.2 Program Modules in Java (cont.)

- Best way to develop and maintain a large program is to construct it from small, simple pieces. This technique is called divide and conquer.
- Methods facilitate the design, implementation, operation and maintenance of large programs.
- Related classes are typically grouped into *packages* so that they can be *imported* into programs and *reused*
- Imported:
  - `import java.util.Scanner`
    - `java.util` is a package
    - `Scanner` is a class
- Reused:
  - `Scanner input = new Scanner(System.in)`
  - `input.nextInt();`
- The Java API provides a rich collection of predefined classes

Reusability of Program Modules in Java

- The statements in method bodies are written only once, are hidden from other methods and can be reused (*called*) from several locations in a program.
- Software reusability
  - using existing methods as building blocks to create new programs
- Often, you can create programs from existing classes and methods rather than by building customized code
- Dividing a program into meaningful methods makes the program easier to debug and maintain.

Software Engineering Observation 5.2

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 5.1

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

Software Engineering Observation 5.3

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 ones.

Invoking Methods

- A method is invoked by a method call
- When the called method completes its task, it returns control—and possibly a result—to the caller
- Similar to the hierarchical form of management (Fig. 5.1)
  - 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
  - This “hiding” of implementation details promotes good software engineering

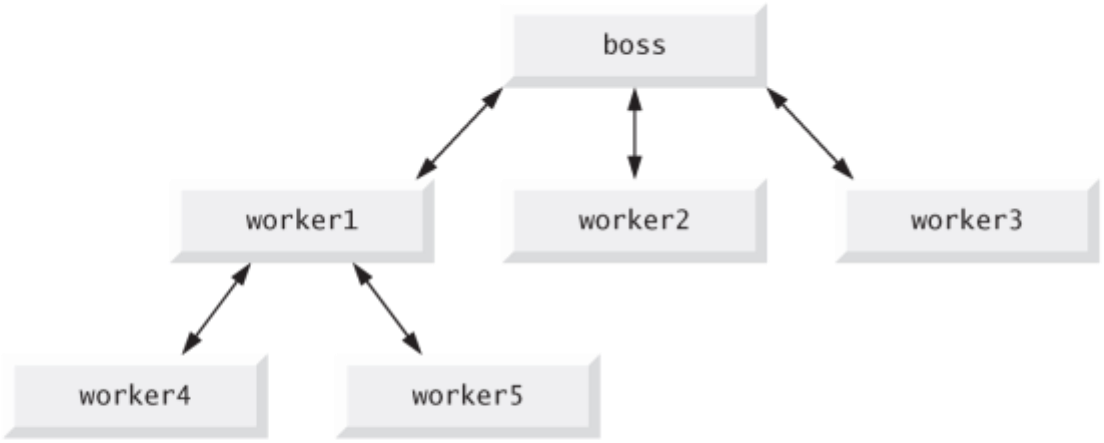
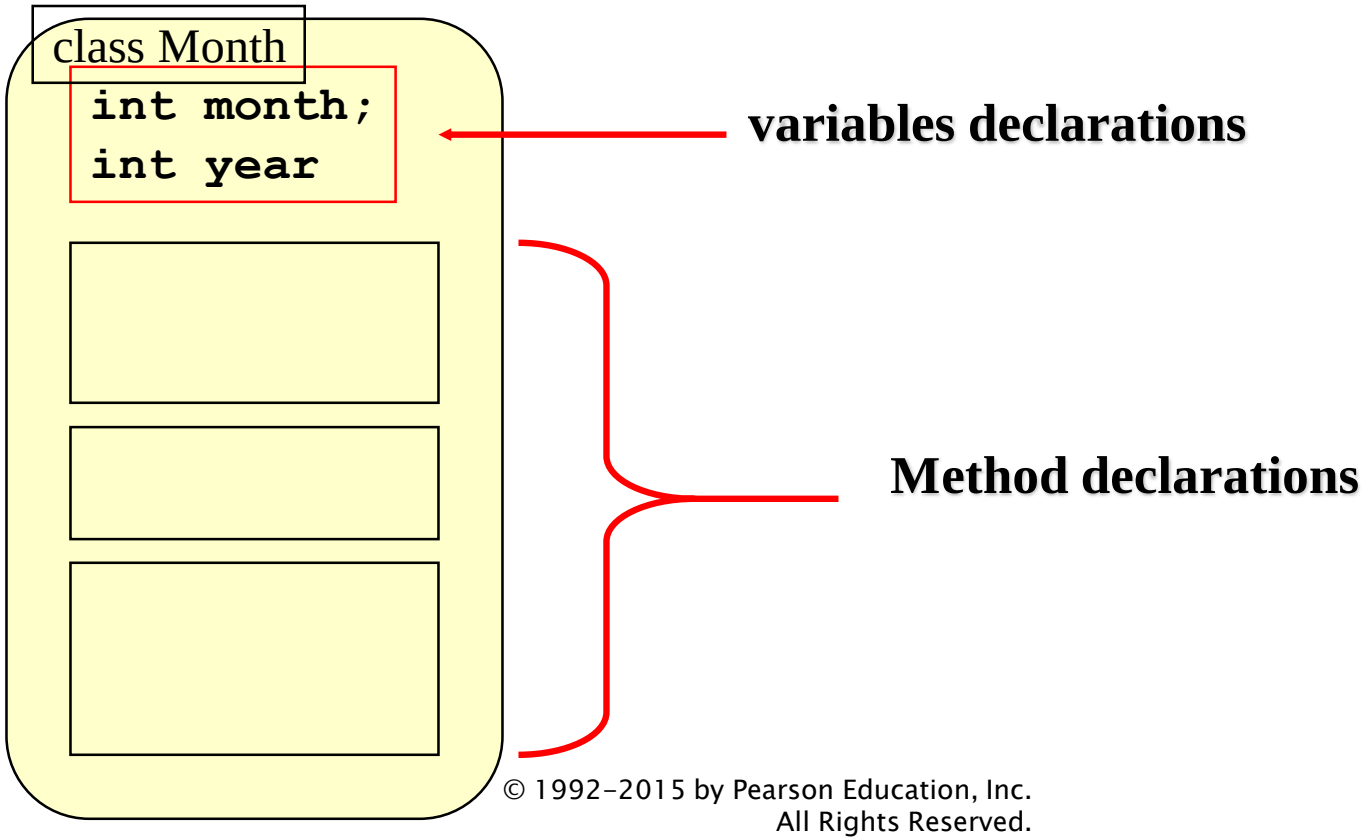


Fig. 5.1 | Hierarchical boss-method/worker-method relationship.

Defining Classes

A class contains data declarations (static and instance variables) and method declarations (behaviors)



Static Methods

- Sometimes a method performs a task that does not depend on an object.
- Such a method applies to the class in which it's declared as a whole and is known as a **static method** or a **class method**.
- To declare a method as static, place the keyword **static** before the return type in the method's declaration.
- For any class imported into program, call the class's static methods by specifying the name of the class followed by a dot (.) and the method name

  ClassName.methodName(arguments)

Math Class Methods as a static method

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
`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

Math Class Static Methods

- Class Math provides a collection of methods that enable you to perform common mathematical calculations.
- all Math class methods are static
- Each is called by preceding its name with the class name Math and the dot (.) separator
- *Class* Math *is part of the* java.lang *package,*
  - *it is implicitly imported by the compiler*
  - *it's not necessary to import class* Math *to use its methods.*

```
System.out.println(Math.sqrt(900.0));
```

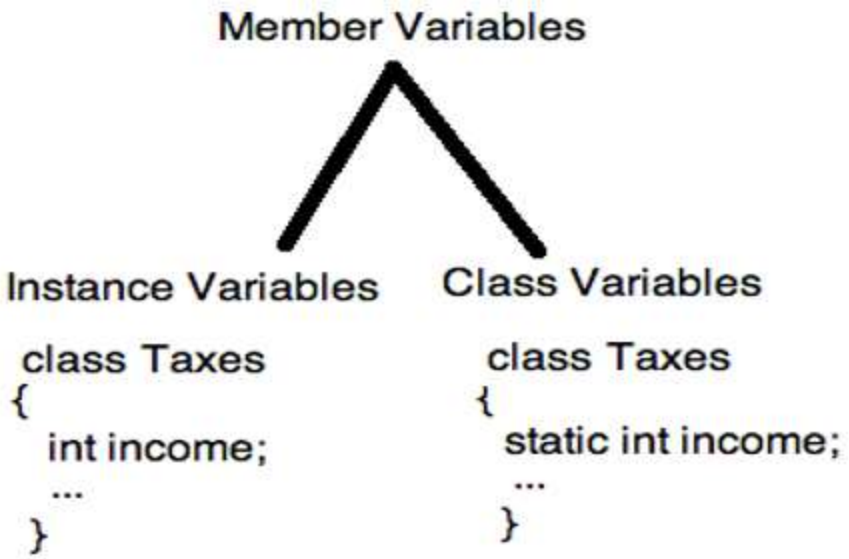
Class Variables (Static Fields)

- There are variables for which each object of a class does *not* need its own separate copy, Such variables are declared **static**
- They are called as **class variables**.
- When objects of a class containing static variables are created, all the objects of that class share *one* copy of those variables.
- Together a class's static variables and instance variables are known as its **fields**.
- A class variable is any field declared with the static modifier;
- Tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated.
- The keyword final could be added to indicate that the value of the class variable is constant and will never change.

Instance Variables (Non-Static Fields)

- Objects store their individual states in "non-static fields" also known as *instance variables*. They are declared without the static keyword.
- Their values are corresponding to each *instance* of a class (to each object).

- So a class member variables can be categorized as follows:



Local Variables

- A method will often store its temporary state in its *local variables*.
- The syntax for declaring a local variable is similar to declaring a field .

  int count = 0;
- There is no special keyword designating a variable as local
  - determination comes from the location which is between the opening and closing braces of a method.
  - Local variables are only visible to the methods in which they are declared
  - they are not accessible from the rest of the class.

Accessibility Between Static And Non-static Methods

- A common use for static methods is to access static fields.
- Not all combinations of instance and class variables and methods are allowed:
- Instance methods can access instance variables and instance methods directly.
- Instance methods can access class variables and class methods directly.
- Class methods can access class variables and class methods directly.
- Class methods ***cannot*** access instance variables or instance methods directly else if:
  - they use its object reference
- class methods cannot use the this keyword as there is no instance for this to refer to.

Constants

The static modifier, in combination with the final modifier is used to define constants.

The final modifier indicates that the value of this field cannot change.

```
static final double PI = 3.141592653589793;
```

Constants defined in this way cannot be reassigned, and it is a compile-time error if the program tries to do so.

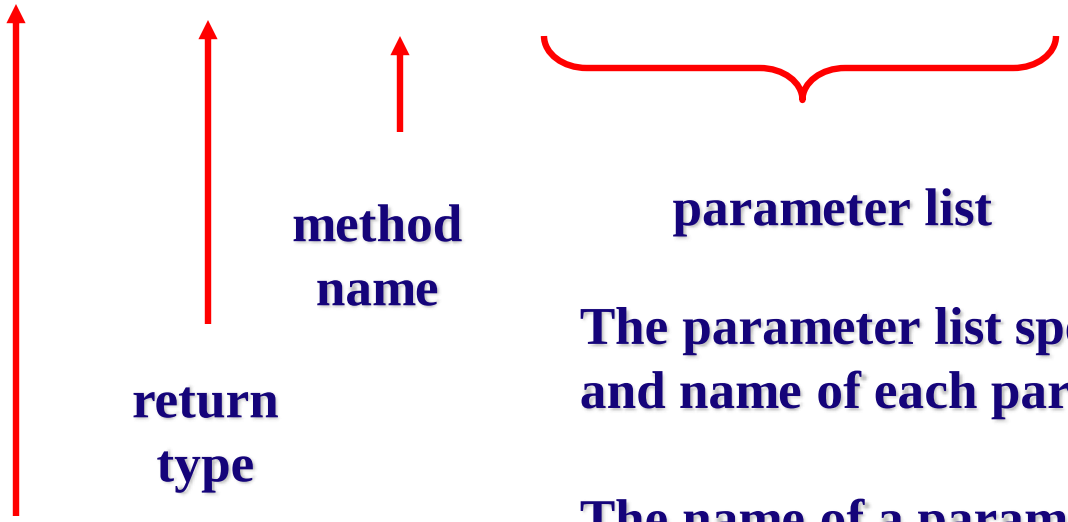
Methods

- A program that provides some functionalities can be long and contains many statements.
- A method groups a sequence of statements and should provide a well-defined, easy-to-understand functionality.
  - a method takes input, performs actions, and produces output
- In Java, each method is defined within specific class

Method Declaration: Header

- A method declaration begins with a *method header*

```
class MyClass
{
   static  int    min  ( int num1, int num2 )
     …
```



The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal parameters*

Method Declaration: Body

The header is followed by the *method body:*

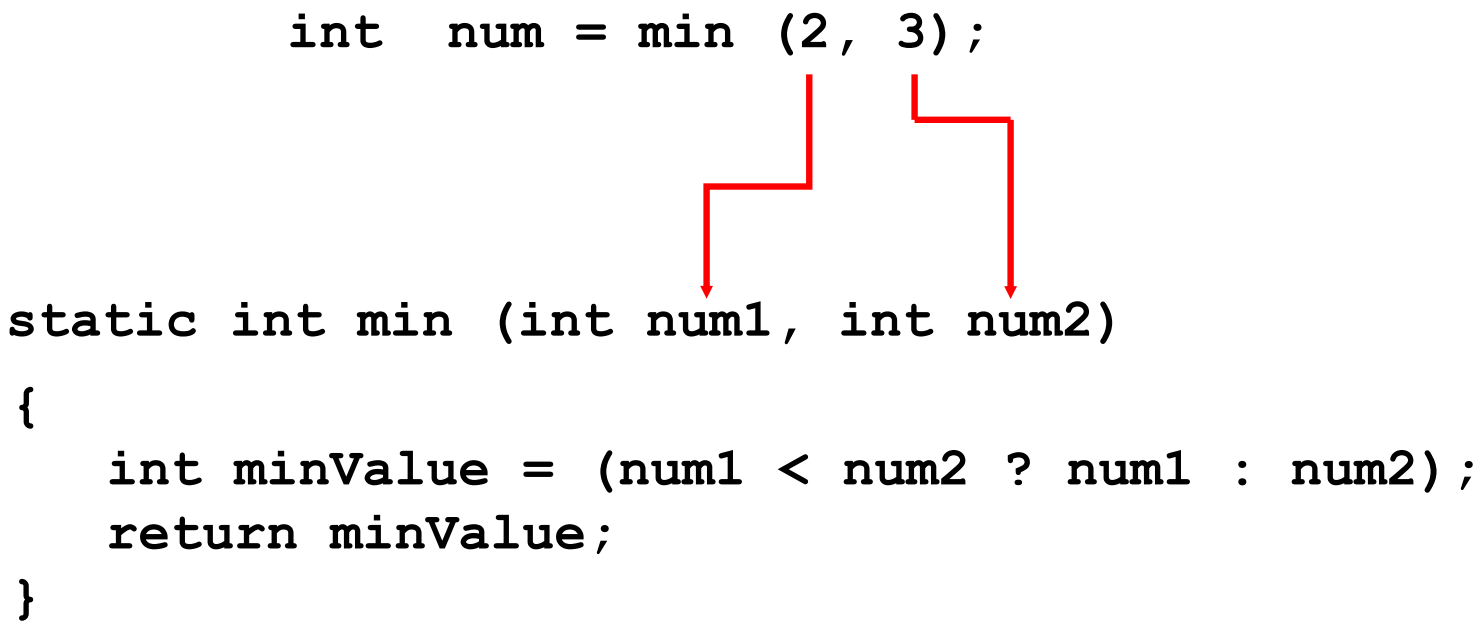
```
class MyClass
{   …
    static int min(int num1, int num2)
    {
        int minValue = num1 < num2 ? num1 : num2;
        return minValue;
    }
    …
 }
```

Note:

- The return type of a method indicates the type of value that the method sends back to the calling location.
- A method that does not return a value has a void return type.

Calling a Method

- Each time a method is called, the values of the *actual arguments* in the invocation are assigned to the *formal arguments.*



Declaring Methods

- A `public` method is "available to the public" and it can be called from methods of other classes.
- A `static` methods in the same class can call each other directly
  - Any other class that uses a static `method` must fully qualify the method name with the class name.
- Class names, method names and variable names are all identifiers and by convention all use the same *camel case* naming scheme we discussed in Chapter 2.
- Class names begin with an initial *uppercase* letter, and method names and variable names begin with an initial *lowercase* letter.
- For a method that requires additional information to perform its task, the method can specify one or more parameters between its parenthesis separated by commas.
  - Each parameter must specify a type and an identifier
- A method's parameters are considered to be local variables of that method and can be used only in that method's body

Common Programming Error 5.1

Declaring method parameters of the same type as `float x, y` *instead of* `float x, float y` *is a syntax error—a type is required for each parameter in the parameter list.*

Common Programming Error 5.2

A compilation error occurs if the number of arguments in a method call does not match the number of parameters in the method declaration.

An Example: Class `MaximumFinder`

- Class `MaximumFinder` (Fig. 5.3) has two methods—`main` (lines 8–25) and `maximum` (lines 28–41).
- The `maximum` method determines and returns the largest of three `double` values .
- Most methods do not get called automatically
  - You must call method `maximum` explicitly to tell it to perform its task

```
1  // Fig. 5.3: 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 static void main(String[] args)
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
```

Fig. 5.3 | Programmer-declared method `maximum` with three `double` parameters. (Part 1 of 3.)

```
23          // display maximum value
24          System.out.println("Maximum is: " + result);
25       }
26
27       // returns the maximum of its three double parameters
28       public static 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       }
42    } // end class MaximumFinder
```

Fig. 5.3 | Programmer-declared method **maximum** with three **double** parameters. (Part 2 of 3.)

```
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. 5.3 | Programmer-declared method `maximum` with three `double` parameters. (Part 3 of 3.)

5.7 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.

  Ex:

  - `Math.pow(2,3)`: 2 and 3 are converted automatically to `double`

- Such conversions may lead to compilation errors if Java's promotion rules are not satisfied as in the case that a function expects a parameter of type **int** and it receives a **double**
- These rules specify which conversions are allowed—that is, which ones can be performed *without losing data*
- The promotion rules 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.

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. 5.4 | Promotions allowed for primitive types.

Common Programming Error 5.8

Casting a primitive-type value to another primitive type may change the value if the new type is not a valid promotion. For example, casting a floating-point value to an integer value may introduce truncation errors (loss of the fractional part) into the result.

```
public class PromotionsTest {
   public static void main(String[] args) {
        short   n1=1;
        int     n2=5;
        char    n3='a';
        float   n4=0;
        double  n5=5.3;
        boolean n6=true;

        System.out.println("Method with type int");
        function1(n1);
        function1(n2);
        function1(n3);
        function1(n4);
        function1(n5);
        function1(n6);

        System.out.println("Method with type double");
        function2(n1);
        function2(n2);
        function2(n3);
        function2(n4);
        function2(n5);
        function2(n6);
    } // main end

    public static void function1(int a)
    {   System.out.println(a); }

    public static void function2(double b)
    {   System.out.println(b); }
} // class end
```

5 min in class

Which calls are allowed and which are not?

Promotion Testing

```
public class PromotionTest {
   public static void main(String[] args) {
       short   n1=1;       int     n2=5;       char    n3='a';       float   n4=0;       double  n5=5.3;       boolean n6=true;

       System.out.println("Method with type int");
       function1(n1);
       function1(n2);
       function1(n3);
      //function1(n4);          // incompatible types: possible lossy conversion from float to int
      //function1(n5);          // incompatible types: possible lossy conversion from double to int
      //function1(n6);          // incompatible types: boolean can not be converted into int

       System.out.println("Method with type double");
       function2(n1);
       function2(n2);
       function2(n3);
       function2(n4);
       function2(n5);
      //function2(n6); //incompatable types: bolean can not be converted into double
   } // main end

   public static void function1(int a)
   {   System.out.println(a); }

   public static void function2(double b)
   {   System.out.println(b); }
} // class end
```

```
run:
Method with type int
1
5
97
Method with type double
1.0
5.0
97.0
0.0
5.3
BUILD SUCCESSFUL (total time: 0 seconds)
```

Generating Random numbers

- The element of chance can be introduced in a program via a variable (an object) of type (of class) SecureRandom (package java.security).

  SecureRandom randomNumbers = new SecureRandom();

- Such variables (objects) can produce random **boolean, byte, float, double, int, long and Gaussian values using the methods:**

  randomNumbers.nextBoolean(), randomNumbers.nextBytes(), randomNumbers.nextInt(), etc.

- SecureRandom variables (objects) produce nondeterministic random numbers that cannot be predicted. For xample, each time nextInt () is called for example implies that every integer value in the range of int data type should have an equal chance (or probability) of being chosen.

- Class `RandomSecure` 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.
- scaling factor—represents the number of unique values that `nextInt` should randomly select from them.
  - Example: we have 6 unique values (from 0 to 5) produced randomly by randomNumbers.nextInt(6)

Case study1: Generating 20 random integers from the range 1 to 6.

```
1  // Fig. 5.6: RandomIntegers.java
2  // Shifted and scaled random integers.
3  import java.security.SecureRandom; // program uses class SecureRandom
4
5  public class RandomIntegers
6  {
7     public static void main(String[] args)
8     {
9        // randomNumbers object will produce secure random numbers
10       SecureRandom randomNumbers = new SecureRandom();
11
12       // loop 20 times
13       for (int counter = 1; counter <= 20; counter++)
14       {
15          // pick random integer from 1 to 6
16          int 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       }
24    }
25 } // end class RandomIntegers
```

```
1  5  3  6  2
5  2  6  5  2
4  4  4  2  6
3  1  6  2  2
```

```
6  5  4  2  6
1  2  5  1  3
6  3  2  2  1
6  4  2  6  4
```

Fig. 5.6 | Shifted and scaled random integers.

Fig. 5.6 | Shifted and scaled random integers. (Part 1 of 2.)

Case study2: Counting the frequencies of getting random integers from the range 1 to 6 when a Die rolled 6 million times.

```
1  // Fig. 5.7: RollDie.java
2  // Roll a six-sided die 6,000,000 times.
3  import java.security.SecureRandom;
4
5  public class RollDie
6  {
7     public static void main(String[] args)
8     {
9        // randomNumbers object will produce secure random numbers
10       SecureRandom randomNumbers = new SecureRandom();
11
12       int frequency1 = 0; // count of 1s rolled
13       int frequency2 = 0; // count of 2s rolled
14       int frequency3 = 0; // count of 3s rolled
15       int frequency4 = 0; // count of 4s rolled
16       int frequency5 = 0; // count of 5s rolled
17       int frequency6 = 0; // count of 6s rolled
18
19       // tally counts for 6,000,000 rolls of a die
20       for (int roll = 1; roll <= 6000000; roll++)
21       {
22          int face = 1 + randomNumbers.nextInt(6); // number from 1 to 6
23
```

Fig. 5.7 | Roll a six-sided die 6,000,000 times. (Part 1 of 3.)

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

Fig. 5.7 | Roll a six-sided die 6,000,000 times. (Part 2 of 3.)

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

```
Face    Frequency
1       999501
2       1000412
3       998262
4       1000820
5       1002245
6       998760
```

```
Face    Frequency
1       999647
2       999557
3       999571
4       1000376
5       1000701
6       1000148
```

Fig. 5.7 | Roll a six-sided die 6,000,000 times. (Part 3 of 3.)

Scopes of variables declarations

- Declarations introduce names that can be used to refer to classes, methods, variables and parameters.
- 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.
- 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 class method or field's scope is the entire body of the class.

Scopes of variables declarations

- A static field is a class variable which is declared at the class level; outside of any method.
- *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
  - Called shadowing
- Figure 5.9 demonstrates scoping issues with `static` and local variables.

An Example Showing Different Scopes of variables

```
public class ScopeTest
{ private static int x =100; // class variable defined outside of any method
  public static void main(String[] args)
  {
     int y=x+100; // here x is the class variable that is defined static
     System.out.println("Value of y is: "+x);
     int x=5; //local-variable declaration seen within the main method
     System.out.println("Value of x in the main method is: "+x);
     If(x>0)
       {   int z=x*100; //here x is the local variable just defined inside main method
           System.out.println("Value of z is: "+z);
       }
     function1(10);
     System.out.println("Value of x in the main method after calling function1 is: "+x);
     function2();
     function2();
   }
   public static void function1(int x)
     {System.out.println("Value of the passed argument: "+x);}
    public static void function2()
      {   x=x+10; // accessing and modifying the class variable x
       System.out.println("Value of the class variable x : "+x);}
}
```

```
run:
Value of y is: 100
Value of x in the main method is: 5
Value of z is: 500
Value of a copy of x inside function1 body block: 10
Value of x in the main method after calling function1 is: 5
Value of a copy of the class variable x multipied by 5: 110
Value of a copy of the class variable x multipied by 5: 120
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
public class scope {

  private static  int x=100;

    public static void main(String[] args)
    {
        int x=5;
        System.out.println(x);
        if(x>0)
        { int y=2;
            System.out.println(y); }

        function1(10);
        System.out.println(x);
        function2();
    }

    public static void function1(int x)
    {  System.out.println(x);     }

    public static void function2()
    {  System.out.println(x);     }

}
```

```
1  // Fig. 5.9: 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 static int x = 1;
8
9     // method main creates and initializes local variable x
10    // and calls methods useLocalVariable and useField
11    public static void main(String[] args)
12    {
13       int x = 5; // method's local variable x shadows field x
14
15       System.out.printf("local x in main 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 main is %d%n", x);
23    }
24
```

Fig. 5.9 | Scope class demonstrates field and local-variable scopes. (Part 1 of 3.)

```
25      // create and initialize local variable x during each call
26      public static void useLocalVariable()
27      {
28         int x = 25; // initialized each time useLocalVariable is called
29
30         System.out.printf(
31            "%nlocal x on entering method useLocalVariable is %d%n", x);
32         ++x; // modifies this method's local variable x
33         System.out.printf(
34            "local x before exiting method useLocalVariable is %d%n", x);
35      }
36
37      // modify class Scope's field x during each call
38      public static void useField()
39      {
40         System.out.printf(
41            "%nfield x on entering method useField is %d%n", x);
42         x *= 10; // modifies class Scope's field x
43         System.out.printf(
44            "field x before exiting method useField is %d%n", x);
45      }
46   } // end class Scope
```

Fig. 5.9 | **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. 5.9 | **Scope** class demonstrates field and local-variable scopes. (Part 3 of 3.)

Method Overloading

- A class may define multiple methods with the same name---this is called method overloading
  - usually perform the same task on different data types

- Example: The PrintStream class defines multiple println methods, i.e., println is overloaded:

  println (String s)

  println (int i)

  println (double d)

  …

- The following lines use the System.out.print method for different data types:

```
System.out.println ("The total is:");
double total = 0;
System.out.println (total);
```

Method Overloading: Signature

- The compiler must be able to determine which version of the method is being invoked
- This is by analyzing the parameters, which form the *signature* of the method
  - the signature includes the type and order of the parameters
    - if multiple methods match a method call, the compiler picks the best match
    - if none matches exactly but some implicit conversion can be done to match a method, then the method is invoke with implicit conversion.
  - the return type of the method is not part of the signature

Common Programming Error 5.9

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

Method Overloading

Version 1

```
double tryMe (int x)
{
   return x + .375;
}
```

Version 2

```
double tryMe (int x, double y)
{
   return x * y;
}
```

Invocation

```
result = tryMe (25, 4.32)
```

Another Example:

```
double tryMe ( int x )
{
    return x + 5;
}
```

```
double tryMe ( double x )
{
    return x * .375;
}
```

```
double tryMe (double x, int y)
{
    return x + y;
}
```

Which tryMe will be called?

```
tryMe( 1 );

tryMe( 1.0 );

tryMe( 1.0, 2);

tryMe( 1, 2);

tryMe( 1.0, 2.0);
```

Method-Call Stack and Activation Records

- Stack data structure is analogous to a pile of dishes that when a dish is placed on the pile, it's normally placed at the top (referred to as pushing onto the stack). Similarly, when a dish is removed from the pile, it's normally removed from the top (referred to as popping off the stack).

- When a program *calls* a method, the called method must know how to *return* to its caller, so the *return address* of the calling method is *pushed* onto the method-call stack.
- If a series of method calls occurs, the successive return addresses are pushed onto the stack in **last-in, first-out (LIFO)** order.

- The method-call stack also contains the memory for the *local variables* (including the method parameters) used in each invocation of a method such that:
  - This data, stored as a portion of the method-call stack, is known as the stack frame (or activation record) of the method call.

- When a method returns to its caller, the stack frame for the method call is *popped* off the stack and those local variables are no longer known to the program

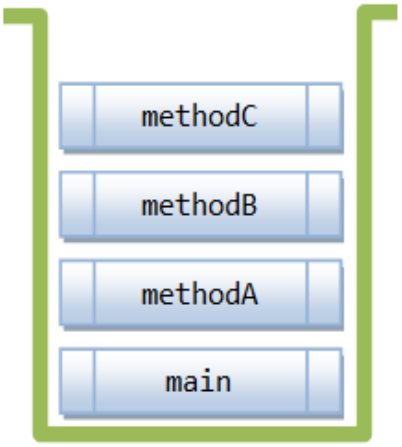
```java
/**
 * Example from and figure from:
 *
https://www3.ntu.edu.sg/home/ehchua/programming/java/J5a_ExceptionAsse
rt.html
 */

public class MethodCallStackDemo
{
   public static void main(String[] args) {
      System.out.println("Enter main()");
      methodA();
      System.out.println("Exit main()");
   }

   public static void methodA() {
      System.out.println("Enter methodA()");
      methodB();
      System.out.println("Exit methodA()");
   }

   public static void methodB() {
      System.out.println("Enter methodB()");
      methodC();
      System.out.println("Exit methodB()");
   }

   public static void methodC() {
      System.out.println("Enter methodC()");
      System.out.println("Exit methodC()");
   }
}
```



Method Call Stack
(Last-in-First-out Queue)

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exit methodC()
Exit methodB()
Exit methodA()
Exit main()
```

18.2 Recursion Concepts

- A recursive function is a function that calls itself, either directly, or indirectly (through another function).
- Recursion is an important topic discussed at length in upper-level computer science courses.
- When a recursive method is called to solve a problem, it actually is capable of solving only the *simplest case(s)*, or base case(s).
  - If the method is called with a *base case*, it returns a result.
- If the method is called with a more complex problem, it divides the problem into two conceptual pieces
  - a piece that the method knows how to do and
  - a piece that it does not know how to do.
- To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler

18.2 Recursion Concepts (cont.)

- Because this new problem resembles the original problem, the method calls a fresh copy of itself to work on the smaller problem
  - this is a recursive call
  - also called the recursion step
- The recursion step normally includes a `return` statement,
- The recursion step executes while the original method call is still active.
- For recursion to eventually terminate, each time the method calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on a base case.
  - When the method recognizes the base case, it returns a result to the previous copy of the method.
  - A sequence of returns ensues until the original method call returns the final result to the caller.

Example of Using Recursion: Factorials

- Factorial of a positive integer *n* is denoted n! and is defined as follows:

  $n! = n * (n - 1) * (n - 2) * \ldots * 1$

  Note that:

  $1! = 1$, $0! = 1$ and * is the multiplication operation.

- The factorial of a positive integer number k can be calculated iteratively (non-recursively):

```
factorial = 1;
for ( int counter = k; counter >= 1; counter-- )
    factorial *= counter;
```

- Recursive declaration of the factorial calculation for a positive integers *n* is arrived at by observing the following relationship:

  $\boldsymbol{n! = n * (n - 1)!}$

Recursion vs. Iteration

- Both iteration and recursion are *based on a control statement*
- Both iteration and recursion involve *repetition*
- Iteration and recursion each involve a *termination test*
- Both iteration and recursion can occur infinitely
- Each recursive call causes another copy of the method (actually, only the method's variables, stored in the activation record) to be created
  - this set of copies can consume considerable memory space.
- Since iteration occurs within a method, repeated method calls and extra memory assignment are avoided.

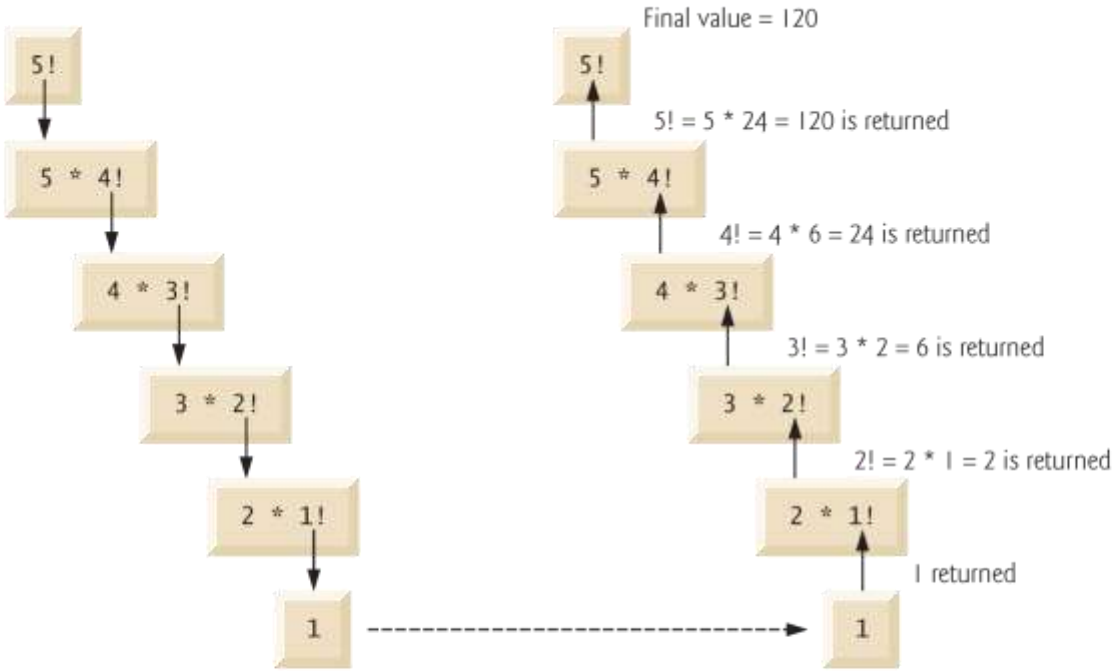


Fig. 18.2 | Recursive evaluation of 5!.

Common Programming Error 18.1

Either omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case can cause a logic error known as ***infinite recursion****, where recursive calls are continuously made until memory is exhausted or the method-call stack overflows. This error is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.*

```
1  // Fig. 18.3: FactorialCalculator.java
2  // Recursive factorial method.
3
4  public class FactorialCalculator
5  {
6     // recursive method factorial (assumes its parameter is >= 0)
7     public static long factorial(long number)
8     {
9        if (number <= 1) // test for base case
10          return 1; // base cases: 0! = 1 and 1! = 1
11       else // recursion step
12          return number * factorial(number - 1);
13    }
14
15    // output factorials for values 0-21
16    public static void main(String[] args)
17    {
18       // calculate the factorials of 0 through 21
19       for (int counter = 0; counter <= 21; counter++)
20          System.out.printf("%d! = %d%n", counter, factorial(counter));
21    }
22 } // end class FactorialCalculator
```

Fig. 18.3 | Recursive factorial method. (Part 1 of 2.)

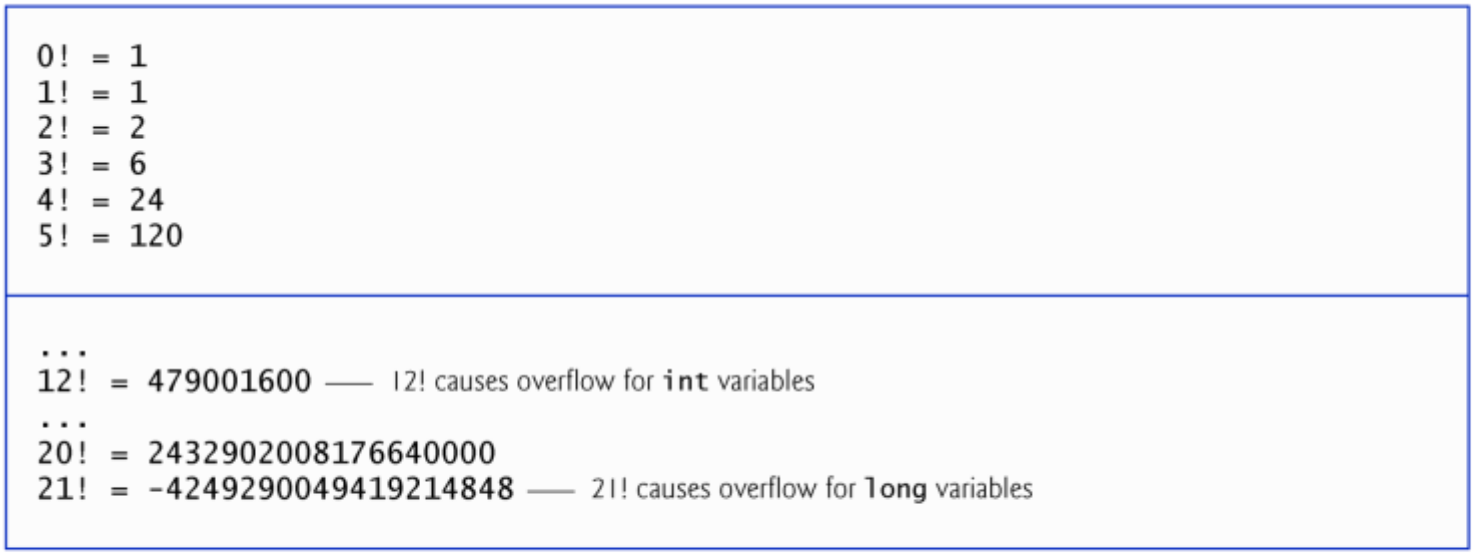


Fig. 18.3 | Recursive factorial method. (Part 2 of 2.)

Notes:

- We use type `long` so the program can calculate factorials greater than 12!.
- The `factorial` method produces large values so quickly that we exceed the largest `long` value when we attempt to calculate 21!.
- Package `java.math` provides classes `BigInteger` and `BigDecimal` explicitly for arbitrary precision calculations that cannot be performed with primitive types.

Example Using Recursion: Fibonacci Series

- The Fibonacci series may be defined recursively as follows:

  fibonacci(0) = 0
  fibonacci(1) = 1
  fibonacci(*n*) = *fibonacci(n – 1) + fibonacci(n – 2)*

- *Two base cases for Fibonacci series are:*

  fibonacci(0) is defined to be 0

  fibonacci(1) to be 1

- Since, Fibonacci numbers tend to become large quickly, We can use type BigInteger as the parameter type and the return type of method fibonacci.

Tracing

fibonacci(0) = **0**
fibonacci(1) = **1**
fibonacci(2) = 1 + 0
fibonacci(3) = 1 + 1
fibonacci(4) = 2 + 1
fibonacci(5) = 3 + 2
fibonacci(6) = 5 + 3
fibonacci(7) = 8 + 5
...
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)

```
1  // Fig. 18.5: FibonacciCalculator.java
2  // Recursive fibonacci method.
3
4
5  public class FibonacciCalculator
6  {
7
8
9     // recursive declaration of method fibonacci
10    public static  long     fibonacci( long     number)
11    {
12       if  ( ( number == 0 ) ||
13             ( number == 1 ) )           // base cases
14          return number;
15       else // recursion step
16          return
17                fibonacci( number - 1 ) +
18                fibonacci( number - 2 );
19    }
20
21    // displays the fibonacci values from 0-40
22    public static void main(String[] args)
23    {
24       for (int counter = 0; counter <= 40; counter++)
25          System.out.printf("Fibonacci of %d is: %d%n", counter,
26            fibonacci(BigInteger.valueOf(counter)));
27    }
28 } // end class FibonacciCalculator
```

Fig. 18.5 | Recursive fibonacci method.

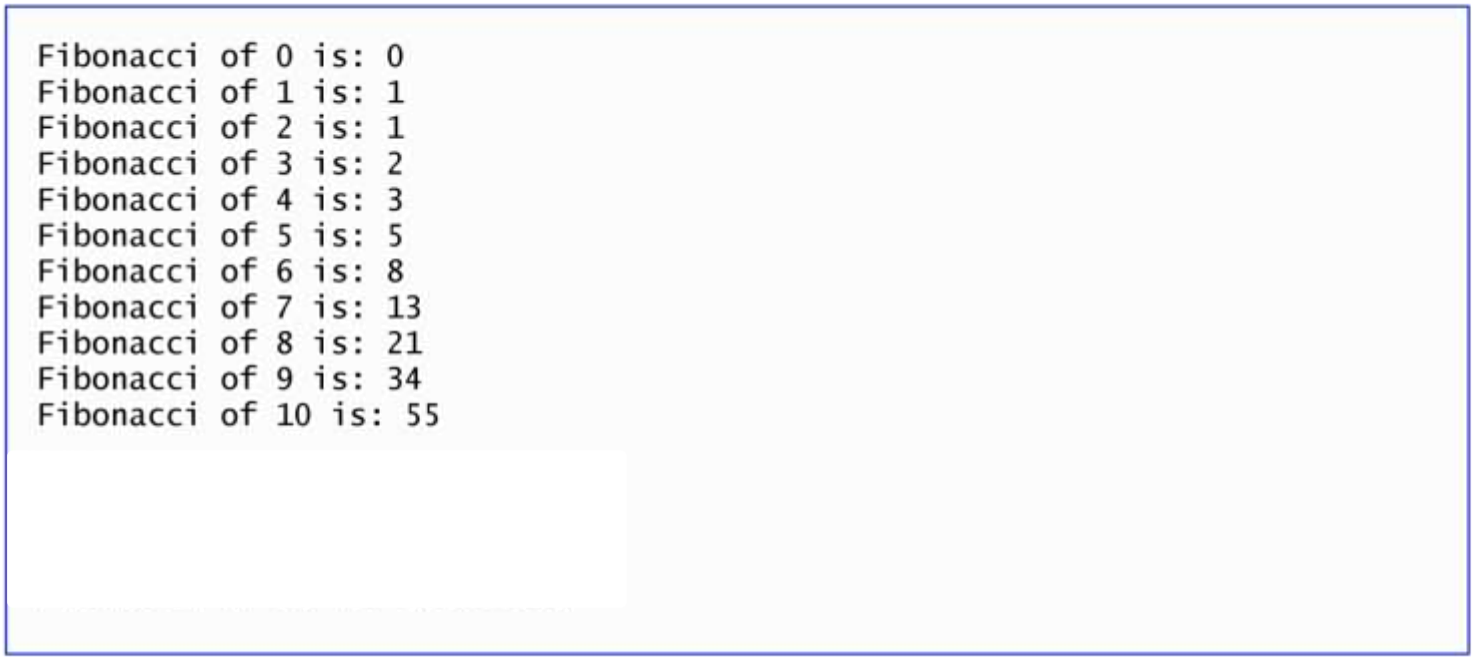


Fig. 18.5 | Recursive fibonacci method.

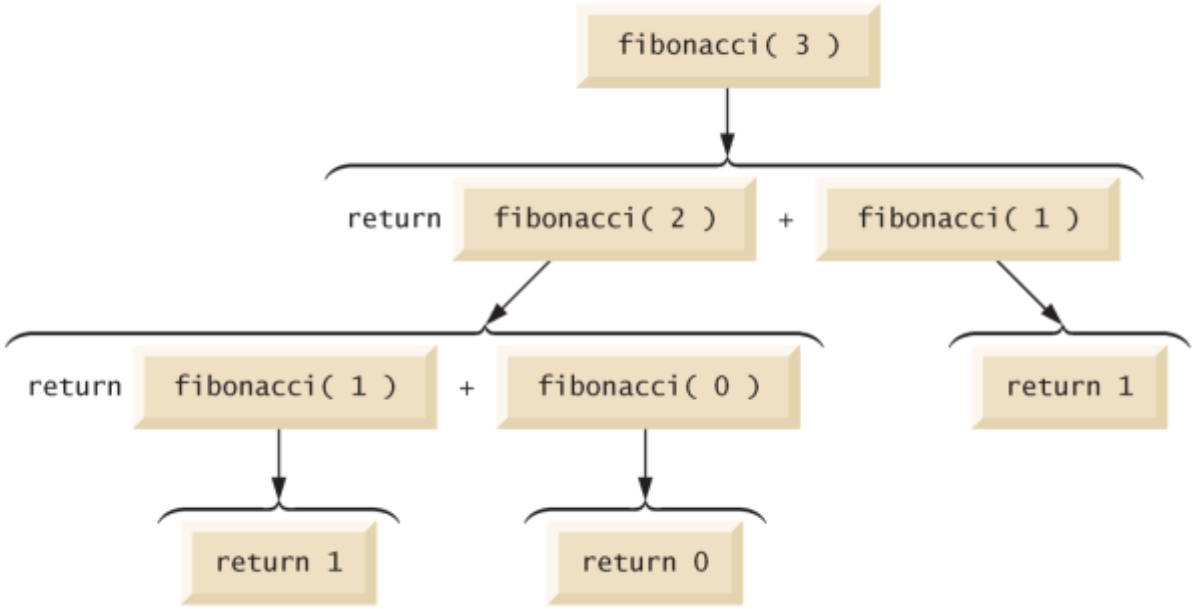


Fig. 18.6 | Set of recursive calls for `fibonacci(3)`.

Performance Tip 18.1

Avoid Fibonacci-style recursive programs, because they result in an exponential "explosion" of method calls.

```
// Passing arrays and individual array elements to methods.
public class PassArray  // main creates array and calls modifyArray and modifyElement
}
 public static void main(String[] args)
  {
   int[] array = { 1, 2, 3, 4, 5 };
   System.out.printf("%n The values of the original array are: ");  // output original array elements
       for (int value : array)
      System.out.printf(" %d", value);
       System.out.printf("%n The values of the modified array are: "); // output modified array elements
       modifyArray(array); // pass array reference
       for (int value : array)
      System.out.printf(" %d", value);
       System.out.printf("%n The value of array[3] before modifyElement: %d", array[3]);
       modifyElement(array[3]); // attempt to modify array[3]
       System.out.printf ("%n The value of array[3] after modifyElement: %d%n", array[3]);
   }
   // multiply each element of an array by 2—Call by Reference
   public static void modifyArray(int[] arr)
   {
        for (int counter = 0; counter < arr.length; counter++)
               arr[counter] *= 2;
   }
   // multiply argument by 2- Call by value
   public static void modifyElement(int x)
   {
        x *= 2;
       System.out.printf("The value of x in modifyElement: %d", x);
   }
}
```

```
run:

 The values of the original array are: 1 2 3 4 5
 The values of the modified array are: 2 4 6 8 10
 The value of array[3] before modifyElement: 8
 The value of x in modifyElement: 16
 The value of array[3] after modifyElement: 8
BUILD SUCCESSFUL (total time: 0 seconds)
```