# Chapter 6 Methods

## Problem

- Find the sum of integers from 1 to 10, from 20 to 30, and from 35 to 45, respectively.

```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 37; i++)
  sum += i;
System.out.println("Sum from 20 to 37 is " + sum);

sum = 0;
for (int i = 35; i <= 49; i++)
  sum += i;
System.out.println("Sum from 35 to 49 is " + sum);
```

2

## Solution

- *Methods can be used to define reusable code and organize and simplify coding.*

```java
public static int sum(int i1, int i2) {
  int result = 0;
  for (int i = i1; i <= i2; i++)
    result += i;

  return result;
}

public static void main(String[] args) {
  System.out.println("Sum from 1 to 10 is " + sum(1, 10));
  System.out.println("Sum from 20 to 37 is " + sum(20, 37));
  System.out.println("Sum from 35 to 49 is " + sum(35, 49));
}
```



## Objectives

- To define methods with formal parameters (§6.2).
- To invoke methods with actual parameters (i.e., arguments) (§6.2).
- To define methods with a return value (§6.3).
- To define methods without a return value (§6.4).
- To pass arguments by value (§6.5).
- To develop reusable code that is modular, easy to read, easy to debug, and easy to maintain (§6.6).
- To write a method that converts hexadecimals to decimals (§6.7).
- To use method overloading and understand ambiguous overloading (§6.8).
- To determine the scope of variables (§6.9).
- To apply the concept of method abstraction in software development (§6.10).
- To design and implement methods using stepwise refinement (§6.11).

## 6.2 Defining a Method

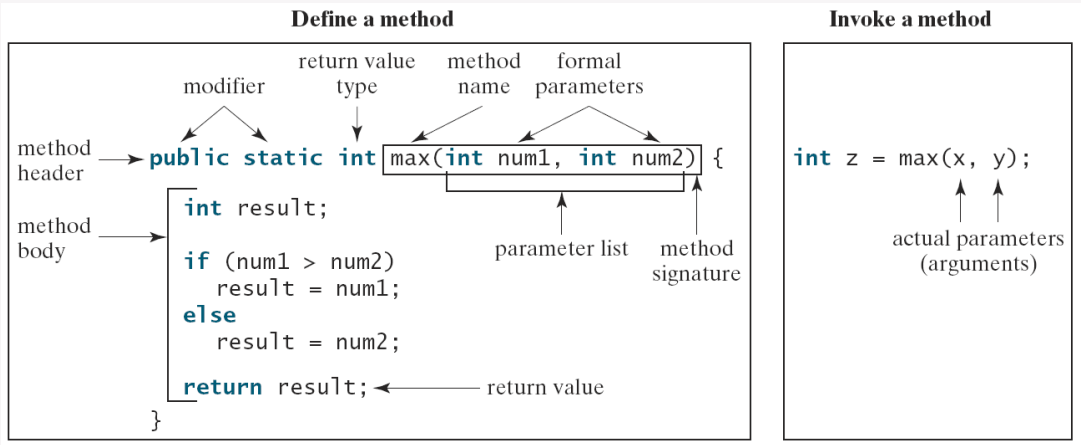
- A method definition consists of its method name, parameters, return value type, and body.
- The syntax for defining a method is as follows:

```
modifier returnValueType methodName(list of parameters) {
  // Method body;
}
```

- A method definition consists of a *method header* and a *method body*.
- The *method header specifies the modifiers, return value type, method name, and parameters* of the method.

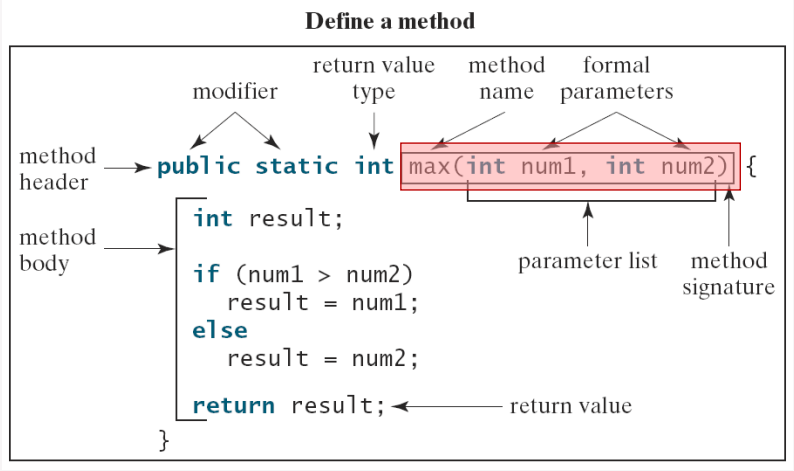
## Defining Methods

- A method is a collection of statements that are grouped together to perform an operation.



## Method Signature

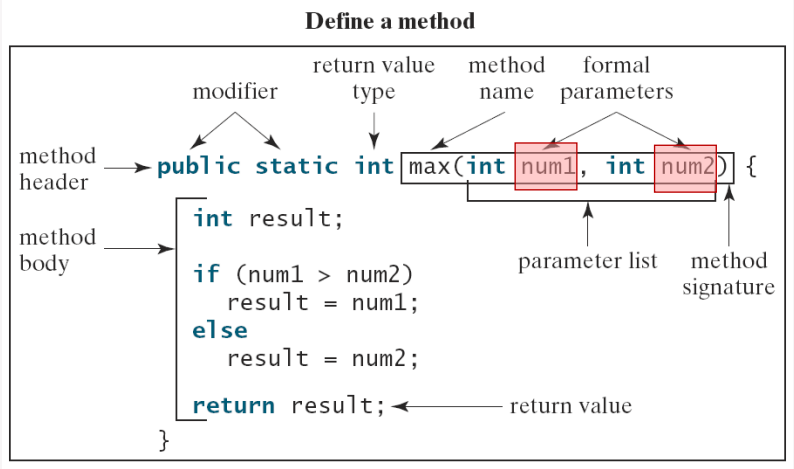
- *Method signature* is the combination of the method name and the parameter list.




7

## Formal Parameters

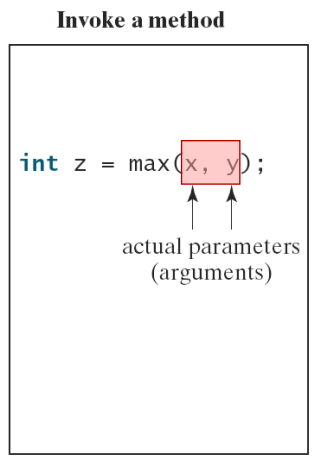
- The variables defined in the method header are known as *formal parameters*.




8

## Actual Parameters

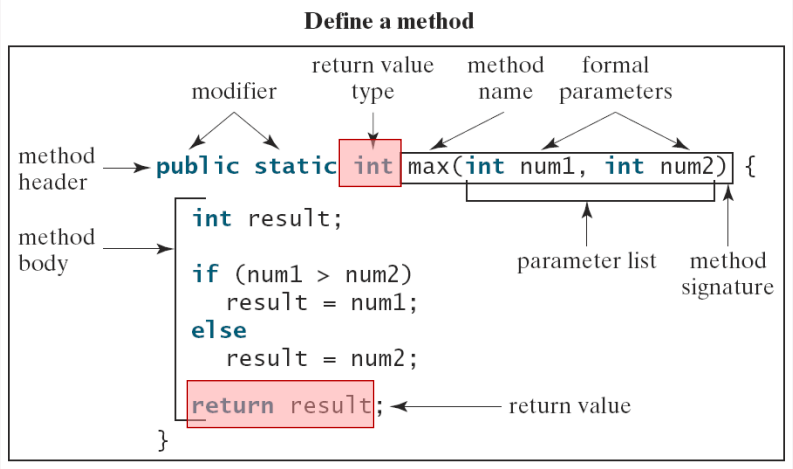
- When a method is invoked, you pass a value to the parameter. This value is referred to as *actual parameter or argument*.






## Return Value Type

- A method may return a value. The returnValueType is the data type of the value the method returns. If the method does not return a value, the returnValueType is the keyword void.



## 6.3 Calling a Method

- Calling a method executes the code in the method.
- In a method definition, you define what the method is to do. To execute the method, you have to call or invoke it.
- If a method returns a value, a call to the method is usually treated as a value.
- If a method returns void, a call to the method must be a statement.

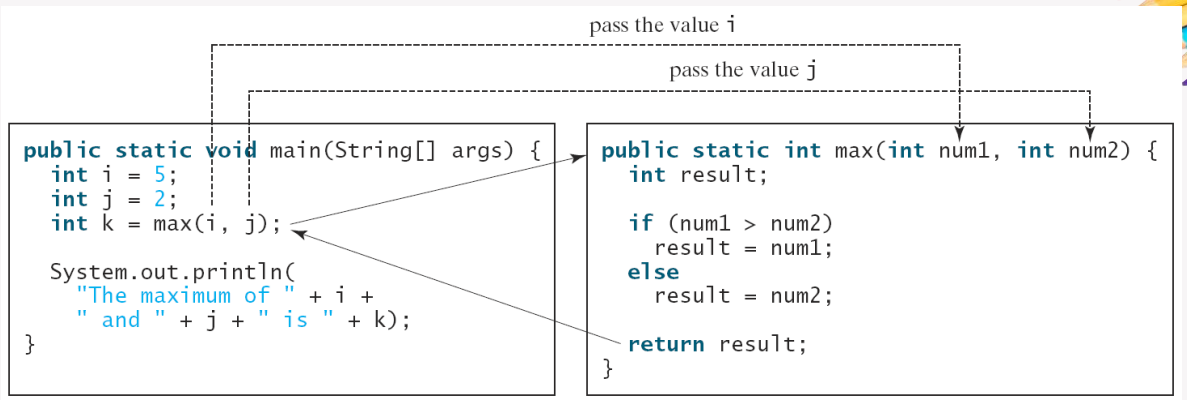
- Testing the max method
- This program demonstrates calling a method max to return the largest of the int values

TestMax

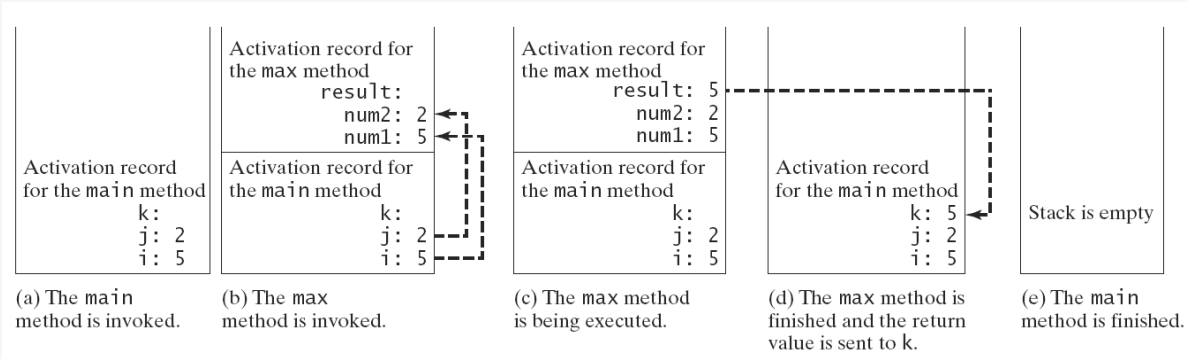
Run

11

## Calling a Method



## Call Stacks




(a) The `main` method is invoked. (b) The `max` method is invoked. (c) The `max` method is being executed. (d) The `max` method is finished and the return value is sent to `k`. (e) The `main` method is finished.

## CAUTION

- A **return statement** is required for a value-returning method.
- The method shown below in (a) is logically correct, but it has a compilation error because the Java compiler thinks it possible that this method does not return any value.

```java
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else if (n < 0)
    return -1;
}
```

(a)

Should be →

```java
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else
    return -1;
}
```

(b)

## Reuse Methods from Other Classes

- NOTE: One of the benefits of methods is for reuse.
- The max method can be invoked from any class besides TestMax.
- If you create a new class Test, you can invoke the max method using **ClassName.methodName** (e.g., TestMax.max).

15

## 6.4 void vs. Value-Returning Methods

- *A void method does not return a value.*
- The method performs some actions. A call to a void method must be a statement.
- The differences between a void and value-returning method:
  - The **printGrade** method is a void method.
  - The **getGrade** method returns a character grade based on the numeric score value.

TestVoidMethod | Run

TestReturnGradeMethod | Run

16

## 6.5 Passing Arguments by Values

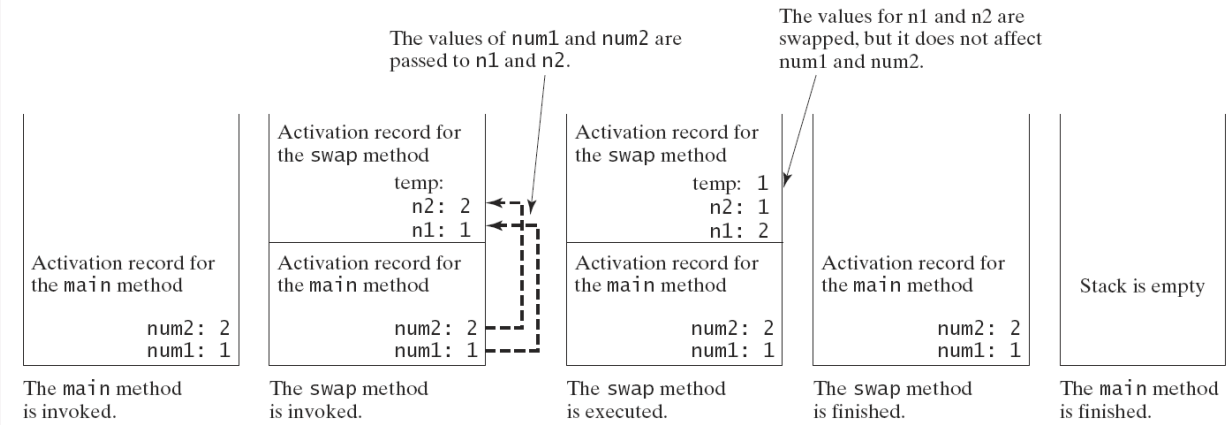
- When calling a method, you need to provide arguments, which must be given in the same order as their respective parameters in the method signature.
- When you invoke a method with an argument, the value of the argument is passed to the parameter. This is referred to as *pass-by-value*.
- Testing Pass by value

Increment | Run

- This program demonstrates passing values to the methods.

TestPassByValue | Run

## Pass by Value



## 6.6 Modularizing Code

- *Modularizing makes the code easy to maintain and debug and enables the code to be reused.*
- Methods can be used to reduce redundant coding and enable code reuse.
- Methods can also be used to modularize code and improve the quality of the program.

GreatestCommonDivisorMethod | Run

PrimeNumberMethod | Run

19

## Case Study: Converting Hexadecimals to Decimals

- Write a method that converts a hexadecimal number into a decimal number.
- Given a hexadecimal number $h_n h_{n-1} h_{n-2} \ldots h_2 h_1 h_0$, the equivalent decimal value is

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_1 \times 16^1 + h_0 \times 16^0 = (\ldots((h_n \times 16 + h_n - 1) \times 16 + h_n - 2) \times 16 + \ldots + h_1) \times 16 + h_0$$

Hex2Dec | Run

20

## 6.8 Overloading Methods

- *Overloading methods enables you to define the methods with the* ***same name*** *as long as* ***their signatures are different****.*
- Overloading the max Method

```
public static double max(double num1, double num2) {
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

TestMethodOverloading  Run



## Ambiguous Invocation

- *Method overloading, two methods* have the same name but different parameter lists within one class.
- The Java compiler determines which method to use based on the method signature.
- Sometimes there may be two or more possible matches for an invocation of a method, but the compiler cannot determine the most specific match.
- This is referred to as *ambiguous invocation*.
- Ambiguous invocation is a compile error.

## Ambiguous Invocation

```
public class AmbiguousOverloading {
  public static void main(String[] args) {
    System.out.println(max(1, 2));
  }

  public static double max(int num1, double num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }

  public static double max(double num1, int num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
}
```

23

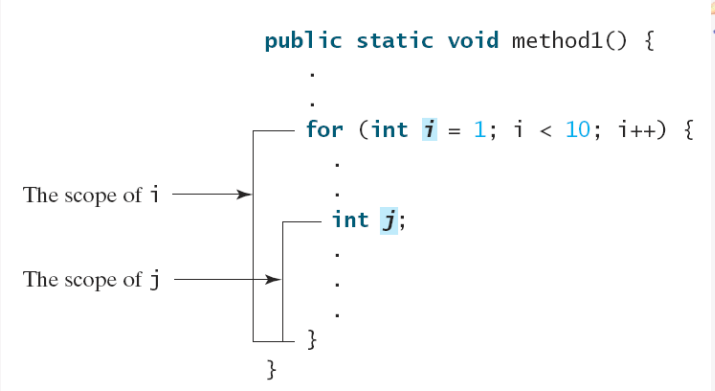
## 6.9 The Scope of Variables

- *The scope of a variable is the part of the program where the variable can be referenced.*
- A variable defined inside a method is referred to as a *local variable*.
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.
- A local variable must be declared and assigned a value before it can be used.
- A parameter is actually a local variable. The scope of a method parameter covers the entire method.

24

## Scope of Local Variables

- A variable declared in the initial action part of a for loop header has its scope in the entire loop.
- But a variable declared inside a for loop body has its scope limited in the loop body from its declaration and to the end of the block that contains the variable.






## Scope of Local Variables

- You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks.

It is fine to declare i in two nonnested blocks.

```
public static void method1() {
  int x = 1;
  int y = 1;

  for (int i = 1; i < 10; i++) {
    x += i;
  }

  for (int i = 1; i < 10; i++) {
    y += i;
  }
}
```

It is wrong to declare i in two nested blocks.

```
public static void method2() {

  int i = 1;
  int sum = 0;

  for (int i = 1; i < 10; i++)
    sum += i;
  }

}
```

## Case Study: Generating Random Characters

- Computer programs process numerical data and characters. You have seen many examples that involve numerical data. It is also important to understand characters and how to process them.
- As introduced in Section 2.9, each character has a unique Unicode between 0 and FFFF in hexadecimal (65535 in decimal). To generate a random character is to generate a random integer between 0 and 65535 using the following expression: (note that since 0 <= Math.random() < 1.0, you have to add 1 to 65535.)

```
(int)(Math.random() * (65535 + 1))
```

27

## Case Study: Generating Random Characters, cont.

- A random integer between (int)'a' and (int)'z' is

```
(int)((int)'a' + Math.random() * ((int)'z' - (int)'a' + 1))
```

- The expression can be simplified as follows:

```
'a' + Math.random() * ('z' - 'a' + 1)
```

- A random lowercase letter is

```
(char)('a' + Math.random() * ('z' - 'a' + 1))
```

- a random character between any two characters **ch1 and ch2 with ch1 < ch2** can be generated as follows:

```
(char)(ch1 + Math.random() * (ch2 - ch1 + 1))
```

28

## The RandomCharacter Class

```java
public class RandomCharacter {
  /** Generate a random character between ch1 and ch2 */
  public static char getRandomCharacter(char ch1, char ch2) {
    return (char)(ch1 + Math.random() * (ch2 - ch1 + 1));
  }

  /** Generate a random lowercase letter */
  public static char getRandomLowerCaseLetter() {
    return getRandomCharacter('a', 'z');
  }

  /** Generate a random uppercase letter */
  public static char getRandomUpperCaseLetter() {
    return getRandomCharacter('A', 'Z');
  }

  /** Generate a random digit character */
  public static char getRandomDigitCharacter() {
    return getRandomCharacter('0', '9');
  }

  /** Generate a random character */
  public static char getRandomCharacter() {
    return getRandomCharacter('\u0000', '\uFFFF');
  }
}
```

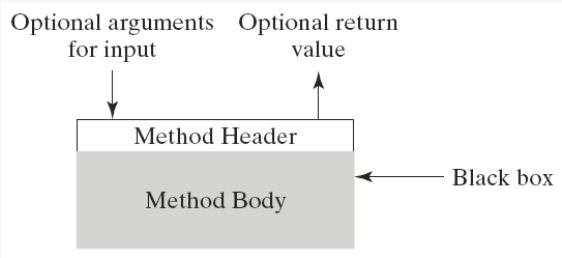
RandomCharacter

TestRandomCharacter

Run

## 6.11 Method Abstraction and Stepwise Refinement

- *Method abstraction* is achieved by separating the use of a method from its implementation.
- The client can use a method without knowing how it is implemented. The details of the implementation are encapsulated in the method and hidden from the client who invokes the method.
- This is also known as *information hiding* or *encapsulation*.
- You can think of the method body as a black box that contains the detailed implementation for the method.
  - reuse, information hiding, reduce complexity



## Stepwise Refinement

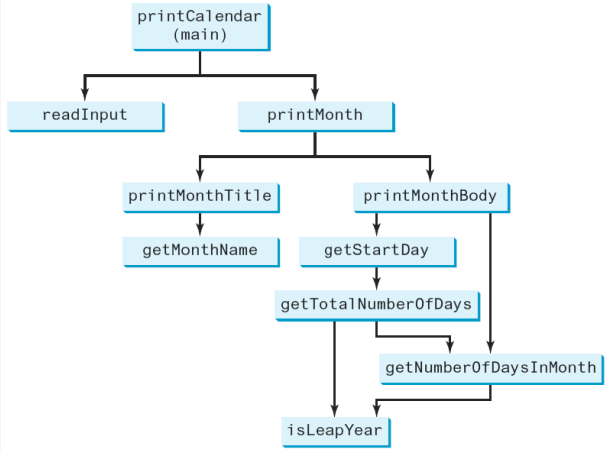
- The concept of method abstraction can be applied to the process of developing programs.
- When writing a large program, you can use the “divide and conquer” strategy, also known as stepwise refinement, to decompose it into subproblems.
- The subproblems can be further decomposed into smaller, more manageable problems.

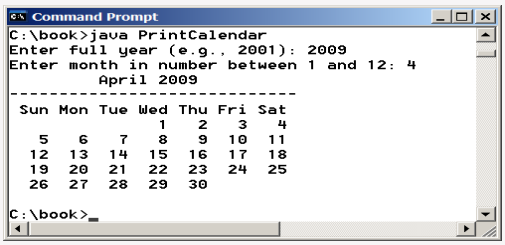
- Benefits of Stepwise Refinement
  - Simpler Program
  - Reusing Methods
  - Easier Developing, Debugging, and Testing
  - Better Facilitating Teamwork

31

## PrintCalender Case Study

- Let us use the PrintCalendar example to demonstrate the stepwise refinement approach.




PrintCalendar

Run

32

# Chapter Summary

## Chapter Summary

- The method header specifies the *modifiers, return value type, method name, and parameters* of the method.
- The method name and the parameter list together constitute the ***method signature***.
- When you invoke a method with an argument, the value of the argument is passed to the parameter. This is referred to as *pass-by-value*.
- A method can be overloaded. This means that two methods can have the same name, as long as their method parameter lists differ.
- A variable declared in a method is called a local variable.
- Method abstraction modularizes programs in a neat, hierarchical manner.

# *Programming Exercises*

*2, 3, 4, 9, 13, 14,*

*16, 18, 24, 26, 30,*

*31, 34, 36, 37, 39*