Java Programming

Methods in Java,
Java API packages,
Using Arrays

Review of Lecture 2

- **Declaring classes**:
  - **public class ClassName**
  - Declaring instance variables private
  - Declaring **setter** and **getter** methods public
  - Declaring **constructors**
  - use **camel case names**
- **Parameters of a method**
  - Specify their type
  - Separate multiple parameters by comma
- **Local variables**
- **set** methods are **void**
- **get** methods return a value using return statement

Review of Lecture 2

- **Driver class**:
  - Provide **interaction with the user**
  - **Create objects of the class:**

    Account myAccount = **new Account();**
  - **Use default or multiple argument constructors**
  - Call **set** and **get** methods on objects:

    myAccount.**setName**(theName);
- Formatting output with **printf**
- Using **JOptionPane** dialog boxes for IO
  - **showInputDialog**
  - **showMessageDialog**
- **Data hiding**
  - By declaring instance variables private

Lesson 3 Objectives

- Create and use **programmer defined methods** in Java classes.
- Declare **static variables**.
- Create and use **static methods** in Java classes.
- Implement **method overloading**.
- Use **arrays** and pass them as arguments to Java methods

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.

Program Modules in Java (Cont.)

- Classes and 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
  - Use existing classes and methods as building blocks to create new pro-grams
- **Divide-and-conquer** approach
  - Constructing programs from small, simple pieces - makes the program easier to debug and maintain.

Hierarchical Relationship Between Method Calls

- Hierarchical form of management.
- 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.

Hierarchical Relationship Between Method Calls (Cont)

- Hierarchical form of management

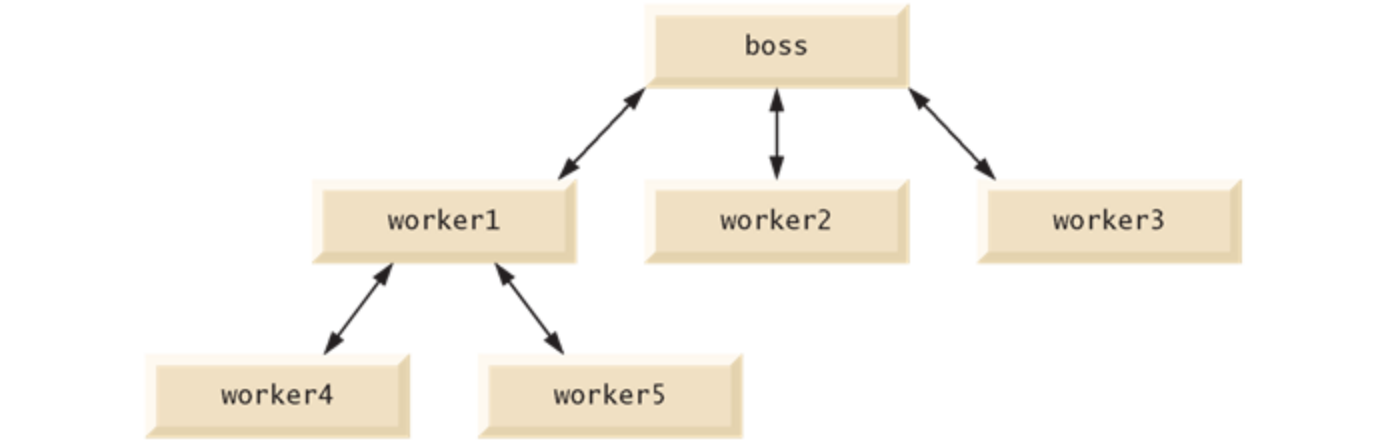

Fig. 6.1 | Hierarchical boss-method/worker-method relationship.

static Methods, static Fields and Class Math

- Sometimes a method performs a task that does not depend on an 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:

```
public static double maximum(double x, double y, double z)
```

- Calling a static method

```
ClassName.methodName(arguments)
```

static Methods, static Fields and Class Math (Cont)

- **Math** Class Methods
- Class Math provides a collection of static methods that enable you to perform common mathematical calculations.
- Method arguments may be constants, variables or expressions.

Math class static 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.)

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

- Recall that each object of a class maintains its own copy of every instance variable of the class.
- There are variables for which each object of a class does not need its own separate copy.
- Such variables are declared static and are also known 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.

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

- Math Class static Constants PI and E
- Math fields for commonly used 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.

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 - at this point 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.

Declaring Methods with Multiple Parameters

- Multiple parameters are specified as a **comma-separated list**.

```
public static double maximum(double x, double y, double z)
```

- 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.

Declaring Methods

- Class MaximumFinder (Fig. 6.3) has two methods - **main** and **maximum**
- 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

```
double result = maximum(number1, number2, number3);
```

Declaring Methods (Cont.)

```
/ Fig. 6.3: MaximumFinder.java
// Programmer-declared method maximum with three double parameters.
import java.util.Scanner;

public class MaximumFinder
{
   // obtain three floating-point values and determine maximum value
   public static void main(String[] args)
   {
      // create Scanner for input from command window
      Scanner input = new Scanner(System.in);
      // prompt for and input three floating-point values
      System.out.print(
         "Enter three floating-point values separated by spaces: ");
      double number1 = input.nextDouble(); // read first double
      double number2 = input.nextDouble(); // read second double
      double number3 = input.nextDouble(); // read third double

      // determine the maximum value
      double result = maximum(number1, number2, number3);

      // display maximum value
      System.out.println("Maximum is: " + result);
   }

   // returns the maximum of its three double parameters
   public static double maximum(double x, double y, double z)
   {
      double maximumValue = x; // assume x is the largest to start

      // determine whether y is greater than maximumValue
      if (y > maximumValue)
         maximumValue = y;

      // determine whether z is greater than maximumValue
      if (z > maximumValue)
         maximumValue = z;

      return maximumValue;
   }
} // end class MaximumFinder
```

Declaring Methods (Cont.)

- A **public method** is “available to the public”
  - Can be called from methods of other classes
- **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
- For now, we begin every method declaration with the keywords public and static

Declaring Methods (Cont.)

- Return type
  - Specifies the type of data a method returns (that is, gives back to its *caller*) to the calling method after performing its task
- In some cases, you'll define methods that perform a task but will not return any information
  - Such methods use the return type **void**.

Declaring Methods (Cont.)

- The method name follows the return type
- Class names, method names and variable names are all identifiers and by convention all use the same *camel case* naming scheme we discussed before.
- 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 that represent that additional information.
  - Defined in a comma-separated parameter-list located in the parentheses that follow the method name
  - 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

Declaring Methods (Cont.)

- **Method header**
  - Modifiers, return type, method name and parameters
- **Method body**
  - Delimited by left and right braces
  - Contains one or more statements that perform the method's task
- A return statement returns a value (or just control) to the point in the program from which the method was called

Declaring Methods (Cont.)

- **String concatenation** allows you to assemble String objects into larger strings by using operators + or +=
- When both operands of operator + are String objects, operator + creates a new String object in which the characters of the right operand are placed at the end of those in the left operand
- *Every primitive value and object in Java can be represented as a String*
- 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"
  - When an object is concatenated with a String, the object's toString method is implicitly called to obtain the String representation of the object

Declaring Methods (Cont.)

Implementing Method maximum by Reusing Method Math.max

- The entire body of our maximum method could also be implemented with two calls to Math.max, as follows:
    - return Math.max(x, Math.max(y, z));
- The outer call to Math.max specifies arguments x and Math.max(y, z).

Declaring Methods (Cont.)

- Non-static methods are typically called **instance methods**.
- A static method can call other static methods of the same class directly and can manipulate static variables in the same class directly.
- To access the class's instance variables and instance methods, a static method must use a reference to an object of the class.

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 an object's variable name followed by a dot (.) and the method name to call a non-static method of the object
  - Using the class name and a dot (.) to call a static method of a class

Declaring and Using Methods

- There are three ways to return control to the statement that calls a method
- If the method does not return a result, control returns when the program flow reaches the method-ending right brace *or* when the statement return; executes
- If the method returns a result, the statement
    - return *expression;*

  evaluates the *expression*, then returns the result to the caller

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
- Such conversions may lead to compilation errors if Java's promotion rules are not satisfied
- 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
- Figure 6.4 lists the primitive types and the types to which each can be promoted

Argument Promotion and Casting

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.

Method-Call Stack and Stack Frames

- **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 onto the stack is the first item popped from the stack.

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 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 method call stack also contains the memory for the local variables (including the method parameters) used in each invocation of a method during a program's execution.
- Stored as a portion of the method call stack known as the **stack frame** (or activation record) of the method call.

Method-Call Stack and Activation Records (Cont.)

- When a method call is made, the stack frame for that method call is pushed onto the method call stack.
- When the method returns to its caller, the stack frame 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 stack frames stored on the program-execution stack, an error known as a **stack overflow** occurs.

Java API Packages

- Java contains many predefined classes that are grouped into categories of related classes called *packages*
- Known as the Java Application Programming Interface (Java API), or the Java class library
- Some key Java API packages are described in Fig. 6.5
- Overview of the packages in Java
  - http://docs.oracle.com/javase/7/docs/api/overview-summary.html
  - http://download.java.net/jdk8/docs/api/overview-summary.html
  - Additional information about a predefined Java class's methods
  - http://docs.oracle.com/javase/7/docs/api/
  - Each static method will be listed with the word "static" preceding its return type

Java API Packages

Package	Description
`java.awt.event`	The **Java Abstract Window Toolkit Event Package** contains classes and interfaces that enable event handling for GUI components in both the `java.awt` and `javax.swing` packages. (See Chapter 12, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2.)
`java.awt.geom`	The **Java 2D Shapes Package** contains classes and interfaces for working with Java's advanced two-dimensional graphics capabilities. (See Chapter 13, Graphics and Java 2D.)
`java.io`	The **Java Input/Output Package** contains classes and interfaces that enable programs to input and output data. (See Chapter 15, Files, Streams and Object Serialization.)
`java.lang`	The **Java Language Package** contains classes and interfaces (discussed throughout the book) that are required by many Java programs. This package is imported by the compiler into all programs.
`java.net`	The **Java Networking Package** contains classes and interfaces that enable programs to communicate via computer networks like the Internet. (See online Chapter 28, Networking.)
`java.security`	The **Java Security Package** contains classes and interfaces for enhancing application security.

Fig. 6.5 | Java API packages (a subset). (Part 1 of 4.)

Java API Packages

Package	Description
`java.sql`	The **JDBC Package** contains classes and interfaces for working with databases. (See Chapter 24, Accessing Databases with JDBC.)
`java.util`	The **Java Utilities Package** contains utility classes and interfaces that enable storing and processing of large amounts of data. Many of these classes and interfaces have been updated to support Java SE 8's new lambda capabilities. (See Chapter 16, Generic Collections.)
`java.util.concurrent`	The **Java Concurrency Package** contains utility classes and interfaces for implementing programs that can perform multiple tasks in parallel. (See Chapter 23, Concurrency.)
`javax.swing`	The **Java Swing GUI Components Package** contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs. This package still uses some elements of the older `java.awt` package. (See Chapter 12, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2.)
`javax.swing.event`	The **Java Swing Event Package** contains classes and interfaces that enable event handling (e.g., responding to button clicks) for GUI components in package `javax.swing`. (See Chapter 12, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2.)

Fig. 6.5 | Java API packages (a subset). (Part 2 of 4.)

Case Study: Random-Number Generation

- The **element of chance** can be introduced in a program via an object of class SecureRandom (package java.security).
- Such objects can produce random boolean, byte, float, double, int, long and Gaussian values.
- SecureRandom objects produce nondeterministic random numbers that cannot be predicted.
- Deterministic random numbers have been the source of many software security breaches.

Random-Number Generation

- If truly *random*, then every value in the range should have an *equal chance* (or probability) of being chosen each time ***nextInt*** is called
- Class **Random** 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

Random-Number Generation

- scaling factor - represents the number of unique values that nextInt should produce
- shift the range of numbers produced by adding a shifting value
- 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

Random-Number Generation

```
// Fig. 6.6: RandomIntegers.java
// Shifted and scaled random integers.
import java.security.SecureRandom; // program uses class SecureRandom

public class RandomIntegers
{
   public static void main(String[] args)
   {
      // randomNumbers object will produce secure random numbers
      SecureRandom randomNumbers = new SecureRandom();

      // loop 20 times
      for (int counter = 1; counter <= 20; counter++)
      {
         // pick random integer from 1 to 6
         int face = 1 + randomNumbers.nextInt(6);
         System.out.printf("%d  ", face); // display generated value

         // if counter is divisible by 5, start a new line of output
         if (counter % 5 == 0)
            System.out.println();
      }
   }
} // end class RandomIntegers
```

Case Study: A Game of Chance; Introducing Enumerations

- 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.
- The application in Fig. 6.8 simulates the game of craps, using methods to implement the game's logic

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

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

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

   // enum type 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;

   // 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
```

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

```
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;
      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;
   }
```

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

```
// 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;
   }

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

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

```
// 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;
   }
} // end class Craps
```

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

- Type Status is a private member of class Craps, because Status will be used only in that class
- Status is an enum type
  - 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 enum 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 enum

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

- Why we declare some constants as public final static int rather than as enum constants:
  - Java does *not* allow an int to be compared to an enum constant
- Unfortunately, Java does not provide an easy way to convert an int value to a particular enum constant

Scope of 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

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.

Scope of Declarations (Cont.)

- *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 6.9 demonstrates scoping issues with static and local variables.

Scope of Declarations (Cont.)

```
// 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
```

Scope of Declarations (Cont.)

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

  // 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);
  }
```

Scope of Declarations (Cont.)

```
// 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 class Scope
```

Method Overloading

- Methods of the *same* name can be declared in the same class, as long as they have *different* sets of parameters
  - Called method overloading
  - The 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 that perform the *same* or *similar* tasks on *different* types or *different* numbers of arguments
- Math methods abs, min and max are overloaded with four versions each:
  - One with two double parameters
  - One with two float parameters
  - One with two int parameters
  - One with two long parameters

Method Overloading (Cont.)

```
// Fig. 6.10: MethodOverload.java
// Overloaded method declarations.

public class MethodOverload
{
   // test overloaded square methods
   public static void main(String[] args)
   {
      System.out.printf("Square of integer 7 is %d%n", square(7));
      System.out.printf("Square of double 7.5 is %f%n", square(7.5));
   }

   // square method with int argument
   public static int square(int intValue)
   {
      System.out.printf("%nCalled square with int argument: %d%n",
         intValue);
      return intValue * intValue;
   }
```

Method Overloading (Cont.)

```
// 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 class MethodOverload
```

Method Overloading (Cont.)

- Literal integer values are treated as type int
- Literal floating-point values are treated as type double
- By default, floating-point values are displayed with six digits of precision if the precision is not specified in the format specifier.

Method Overloading (Cont.)

- Compiler distinguishes overloaded methods by their signatures
  - A combination of the method's *name* and the *number*, *types* and *order* of its parameters, but *not* its return type.
- Internally, the compiler uses longer method names that include the original method name, the types of each parameter and the exact order of the parameters to determine whether the methods in a class are unique in that class.
- *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

Arrays in Java

- Arrays in Java are objects so they are **reference types**.
- Elements can be **either primitive or reference types**.
- Refer to a particular element in an array by using the element's index.
- Array-access expression—the name of the array followed by the index of the particular element in square brackets, [].
- The first element in every array has index zero.
- The highest index in an array is one less than the number of elements in the array.
- Array names follow the same conventions as other variable names.

Arrays in Java

- Every array object knows its own length and stores it in a length instance variable.
- length cannot be changed because it's a final variable.

Enhanced for statement

- Iterates through the elements of an array without using a counter - avoids the possibility of “stepping outside” the array.
- Syntax: for (parameter : arrayName)
  statement
- Example:

```
for (int value : array)
        System.out.printf("   %d", value);
```

- parameter has a type and an identifier and arrayName is the array through which to iterate.
- Parameter type must be consistent with the array’s element type.
- The enhanced for statement simplifies the code for iterating through an array.

Passing arrays as arguments to methods

- To pass an array argument to a method, specify the **name of the array without any brackets**.
- Since every array object “knows” its own length, we need not pass the array length as an additional argument.
- To receive an array, the method’s parameter list must specify an array parameter.
- When an argument to a method is an entire array or an individual array element of a reference type, the **called method receives a copy of the reference**.
- When an argument to a method is an individual array element of a primitive type, the **called method receives a copy of the element’s value**.
- Such primitive values are called scalars or scalar quantities.

PassArray example

References

- Textbook
- Java documentation