COP 3330, Spring 2013

Introduction to Methods
Code examples for basic language features.

Instructor : Arup Ghosh
01-25-13

School of Electrical Engineering and Computer Science
University of Central Florida

Introduction

- Best way to develop and maintain a large program is to construct it from small, simple pieces, or modules.
  - divide and conquer.
- We will discuss -
- `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. – Remember?
- Method overloading.

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.

- Methods help you modularize a program by separating its tasks into self-contained units.

Program Modules in Java (Cont.)

- 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.

`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* `)`

`static` Methods, `static` Fields and Class `Math`

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

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`

`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`

`Math` class methods. (Part 2 of 2.)

`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.

`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.

`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.

Declaring Methods with Multiple Parameters

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

Code Example

- Programmer-declared method “maximum” with 3 double parameters.

Declaring Methods with Multiple Parameters (Cont.)

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

Declaring Methods with Multiple Parameters (Cont.)

- All objects have a `toString` method that returns a `String` representation of the object.

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

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.

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;
  ```

Java API Packages

- Java contains many predefined classes that are grouped into categories of related classes called packages.
- A great strength of Java is the Java API's thousands of classes.
- Overview of the packages in Java SE 6:
  - `download.oracle.com/javase/6/docs/api/overview-summary.html`
- Java API documentation
  - `download.oracle.com/javase/6/docs/api/`

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` (Class) method `random` can produce only `double` values in the range $0.0 \leq x < 1.0$.
- Documentation for class `Random`
  - `download.oracle.com/javase/6/docs/api/java/util/Random.html`

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.
- The range of values produced directly by `Random` method `nextInt` often differs from the range of values required in a particular Java application.
- `Random` method `nextInt` that receives an `int` argument returns a value from 0 up to, but not including, the argument's value.

Random-Number Generation (Cont.)

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

Code Example

- Example on Random Numbers generation.
- Rolling Die

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.
    ```

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.
- 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.

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.*
- Figure 6.10 illustrates the errors generated when two methods have the same signature and different return types.
- Overloaded methods can have different return types if the methods have different parameter lists.
- Overloaded methods need not have the same number of parameters.

Code Example

- Method overloading