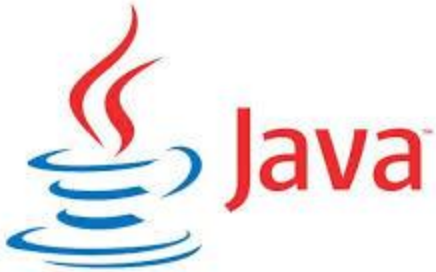
CS212: Object-Oriented Programming

Method Overloading, Using this Reference

Hirra Anwar



OBJECTIVES

- To use common Math methods available in the Java API.
- How the method call/return mechanism is supported by the method call stack and activation records.
- To understand how the visibility of declarations is limited to specific regions of programs.
- What method overloading is and how to create overloaded methods.
- How to refer to the current object's members using the this reference?
- How to use copy constructor to pass objects of the same type as initializers?

Methods in Java

- Called functions or procedures in other languages
- Modularize programs by separating its tasks into self-contained units
- Enable a divide-and-conquer approach
- Are reusable in later programs
- Prevent repeating code

Static Methods, static Fields and Class Math

- Constants
  - Keyword **`final`**
  - Cannot be changed after initialization
- `static` fields (or class variables)
  - Are fields where one copy of the variable is shared among all objects of the class
- `Math.PI` and `Math.E` are `final static` fields of the `Math` class

MATH CLASS 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 *ex*	`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., *xy*)	`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

Declaring Methods with Multiple Parameters

- Multiple parameters can be declared by specifying a comma-separated list.
  - Arguments passed in a method call must be consistent with the number, types and order of the parameters
    - Sometimes called formal parameters

```
1  // Fig. 6.3: MaximumFinder.java
2  // Programmer-declared method maximum.
3  import java.util.Scanner;
4
5  public class MaximumFinder
6  {
7     // obtain three floating-point values and locate the maximum value
8     public void determineMaximum()
9     {
10       // create Scanner for input from command window
11       Scanner input = new Scanner( System.in );
12
13       // obtain user input
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
23       // display maximum value
24       System.out.println( "Maximum is: " + result );
25    } // end method determineMaximum
26
```

Prompt the user to enter and read three **`double`** values

Call method **`maximum`**

Display maximum value

```
27    // returns the maximum of its three double parameters
28    public 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    } // end method maximum
42 } // end class MaximumFinder
```

Declare the **maximum** method

Compare **y** and **maximumValue**

Compare **z** and **maximumValue**

Return the maximum value

```
1  // Fig. 6.4: MaximumFinderTest.java
2  // Application to test class MaximumFinder.
3
4  public class MaximumFinderTest
5  {
6     // application starting point
7     public static void main( String args[] )
8     {
9        MaximumFinder maximumFinder = new MaximumFinder();
10       maximumFinder.determineMaximum();
11    } // end main
12 } // end class MaximumFinderTest
```

Create a **`MaximumFinder`** object

Call the **`determineMaximum`** method

```
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
```

Declaring Methods with Multiple Parameters (Cont.)

- Reusing method `Math.max`
  - The expression `Math.max( x, Math.max( y, z ) )` determines the maximum of `y` and `z`, and then determines the maximum of `x` and that value
- String concatenation
  - Using the `+` operator with two `String`s concatenates them into a new `String`
  - Using the `+` operator with a `String` and a value of another data type concatenates the `String` with a `String` representation of the other value
    - When the other value is an object, its `toString` method is called to generate its `String` representation

Declaring and Using Methods

- Three ways to call a method:
  - Use a method name by itself to call another method of the same class
  - Use a variable containing a reference to an object, followed by a dot (`.`) and the method name to call a method of the referenced object
  - Use the class name and a dot (`.`) to call a `static` method of a class
- `static` methods cannot call non-`static` methods of the same class directly

Declaring and Using Methods

- Three ways to return control to the calling statement:
  - If method does not return a result:
    - Program flow reaches the method-ending right brace or
    - Program executes the statement `return;`
  - If method does return a result:
    - Program executes the statement `return` *expression*`;`
      - *expression* is first evaluated and then its value is returned to the caller

Common Programming Errors

- Declaring a method outside the body of a class declaration or inside the body of another method is a syntax error.
- Omitting the return-value-type in a method declaration is a syntax error.
- Placing a semicolon after the right parenthesis enclosing the parameter list of a method declaration is a syntax error.
- Redeclaring a method parameter as a local variable in the method's body is a compilation error.
- Forgetting to return a value from a method that should return a value is a compilation error.

Java API Packages

- Including the declaration
  `import java.util.Scanner;`
  allows the programmer to use `Scanner` instead of `java.util.Scanner`
- Java API documentation
  - https://docs.oracle.com/javase/7/docs/api/index.html
  - https://docs.oracle.com/javase/8/docs/api/index.html
- Other Resources
  - https://docs.oracle.com/javase/8/

Java API packages (a subset)

Package	Description
`java.applet`	The Java Applet Package contains a class and several interfaces required to create Java applets—programs that execute in Web browsers. (Applets are discussed in Chapter 20, Introduction to Java Applets; interfaces are discussed in Chapter 10, Object_-Oriented Programming: Polymorphism.)
`java.awt`	The Java Abstract Window Toolkit Package contains the classes and interfaces required to create and manipulate GUIs in Java 1.0 and 1.1. In current versions of Java, the Swing GUI components of the `javax.swing` packages are often used instead. (Some elements of the `java.awt` package are discussed in Chapter 11, GUI Components: Part 1, Chapter 12, Graphics and Java2D, and Chapter 22, GUI Components: Part 2.)
`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. (You will learn more about this package in Chapter 11, GUI Components: Part 1 and Chapter 22, GUI Components: Part 2.)
`java.io`	The Java Input/Output Package contains classes and interfaces that enable programs to input and output data. (You will learn more about this package in Chapter 14, Files and Streams.)
`java.lang`	The Java Language Package contains classes and interfaces (discussed throughout this text) that are required by many Java programs. This package is imported by the compiler into all programs, so the programmer does not need to do so.

Java API packages (a subset)

Package	Description
`java.net`	The Java Networking Package contains classes and interfaces that enable programs to communicate via computer networks like the Internet. (You will learn more about this in Chapter 24, Networking.)
`java.text`	The Java Text Package contains classes and interfaces that enable programs to manipulate numbers, dates, characters and strings. The package provides internationalization capabilities that enable a program to be customized to a specific locale (e.g., a program may display strings in different languages, based on the user's country).
`java.util`	The Java Utilities Package contains utility classes and interfaces that enable such actions as date and time manipulations, random-number processing (class `Random`), the storing and processing of large amounts of data and the breaking of strings into smaller pieces called tokens (class `StringTokenizer`). (You will learn more about the features of this package in Chapter 19, Collections.)
`javax.swing`	The Java Swing GUI Components Package contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs. (You will learn more about this package in Chapter 11, 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`. (You will learn more about this package in Chapter 11, GUI Components: Part 1 and Chapter 22, GUI Components: Part 2.)

Scope of Declarations

- Basic scope rules
  - Scope of a parameter declaration is the body of the method in which appears
  - Scope of a local-variable declaration is from the point of declaration to the end of that block
  - Scope of a local-variable declaration in the initialization section of a `for` header is the rest of the `for` header and the body of the `for` statement
  - Scope of a method or field of a class is the entire body of the class

Scope of Declarations

- Shadowing
  - A field is shadowed (or hidden) if a local variable or parameter has the same name as the field
    - This lasts until the local variable or parameter goes out of scope

```
1  // Fig. 6.11: 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 int x = 1;
8
9     // method begin creates and initializes local variable x
10    // and calls methods useLocalVariable and useField
11    public void begin()
12    {
13       int x = 5; // method's local variable x shadows field x
14
15       System.out.printf( "local x in method begin 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
```

Shadows field **x**

Display value of local variable **x**

```
22       System.out.printf( "\nlocal x in method begin is %d\n", x );
23    } // end method begin
24
25    // create and initialize local variable x during each call
26    public 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    } // end method useLocalVariable
36
37    // modify class Scope's field x during each call
38    public 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    } // end method useField
46 } // end class Scope
```

Shadows field **x**

Display value of local variable **x**

Display value of field **x**

```
1  // Fig. 6.12: ScopeTest.java
2  // Application to test class Scope.
3
4  public class ScopeTest
5  {
6     // application starting point
7     public static void main( String args[] )
8     {
9        Scope testScope = new Scope();
10       testScope.begin();
11    } // end main
12 } // end class ScopeTest
```

```
local x in method begin 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 method begin is 5
```

Method Overloading

- Multiple methods with the same name, but different types, number or order of parameters in their parameter lists
- Compiler decides which method is being called by matching the method call's argument list to one of the overloaded methods' parameter lists
  - A method's name and number, type and order of its parameters form its **signature**
- Differences in return type are irrelevant in method overloading
  - Overloaded methods can have different return types
  - Methods with different return types but the same signature cause a compilation error.

```
1  // Fig. 6.13: MethodOverload.java
2  // Overloaded method declarations.
3
4  public class MethodOverload
5  {
6     // test overloaded square methods
7     public void testOverloadedMethods()
8     {
9        System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
10       System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
11    } // end method testOverloadedMethods
12
13    // square method with int argument
14    public int square( int intValue )
15    {
16       System.out.printf( "\nCalled square with int argument: %d\n",
17          intValue );
18       return intValue * intValue;
19    } // end method square with int argument
20
21    // square method with double argument
22    public double square( double doubleValue )
23    {
24       System.out.printf( "\nCalled square with double argument: %f\n",
25          doubleValue );
26       return doubleValue * doubleValue;
27    } // end method square with double argument
28 } // end class MethodOverload
```

Correctly calls the "**square** of **int**" method

Correctly calls the "**square** of **double**" method

Declaring the "**square** of **int**" method

Declaring the "**square** of **double**" method

```
1  // Fig. 6.14: MethodOverloadTest.java
2  // Application to test class MethodOverload.
3
4  public class MethodOverloadTest
5  {
6     public static void main( String args[] )
7     {
8        MethodOverload methodOverload = new MethodOverload();
9        methodOverload.testOverloadedMethods();
10    } // end main
11 } // end class MethodOverloadTest
```

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

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

```
1  // Fig. 6.15: MethodOverloadError.java
2  // Overloaded methods with identical signatures
3  // cause compilation errors, even if return types are different.
4
5  public class MethodOverloadError
6  {
7     // declaration of method square with int argument
8     public int square( int x )
9     {
10       return x * x;
11    }
12
13    // second declaration of method square with int argument
14    // causes compilation error even though return types are different
15    public double square( int y )
16    {
17       return y * y;
18    }
19 } // end class MethodOverloadError
```

Same method signature

```
MethodOverloadError.java:15: square(int) is already defined in
MethodOverloadError
   public double square( int y )
                 ^
1 error
```

Compilation error

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

Referring to the Current Object's Members with the this Reference

- Any object can access a reference to itself with keyword `this`
- Non-`static` methods implicitly use `this` when referring to the object's instance variables and other methods
- Can be used to access instance variables when they are shadowed by local variables or method parameters
- A `.java` file can contain more than one class
  - But only one class in each `.java` file can be `public`

```
1  // Fig. 8.4: ThisTest.java
2  // this used implicitly and explicitly to refer to members of an object.
3
4  public class ThisTest
5  {
6     public static void main( String args[] )
7     {
8        SimpleTime time = new SimpleTime( 15, 30, 19 );
9        System.out.println( time.buildString() );
10    } // end main
11 } // end class ThisTest
12
13 // class SimpleTime demonstrates the "this" reference
14 class SimpleTime
15 {
16    private int hour;   // 0-23
17    private int minute; // 0-59
18    private int second; // 0-59
19
20    // if the constructor uses parameter names identical to
21    // instance variable names the "this" reference is
22    // required to distinguish between names
23    public SimpleTime( int hour, int minute, int second )
24    {
25       this.hour = hour;       // set "this" object's hour
26       this.minute = minute;   // set "this" object's minute
27       this.second = second;   // set "this" object's second
28    } // end SimpleTime constructor
29
```

Create new **SimpleTime** object

Declare instance variables

Method parameters shadow instance variables

Using this to access the object's instance variables

```
30    // use explicit and implicit "this" to call toUniversalString
31    public String buildString()
32    {
33       return String.format( "%24s: %s\n%24s: %s",
34          "this.toUniversalString()", this.toUniversalString(),
35          "toUniversalString()", toUniversalString() );
36    } // end method buildString
37
38    // convert to String in universal-time format (HH:MM:SS)
39    public String toUniversalString()
40    {
41       // "this" is not required here to access instance variables,
42       // because method does not have local variables with same
43       // names as instance variables
44       return String.format( "%02d:%02d:%02d",
45          this.hour, this.minute, this.second );
46    } // end method toUniversalString
47 } // end class SimpleTime
```

Using **this** explicitly and implicitly to call **toUniversalString**

Use of **this** not necessary here

```
this.toUniversalString(): 15:30:19
     toUniversalString(): 15:30:19
```

Common Programming Errors

- It is often a logic error when a method contains a parameter or local variable that has the same name as a field of the class. In this case, use reference `this` if you wish to access the field of the class—otherwise, the method parameter or local variable will be referenced.
- Avoid method parameter names or local variable names that conflict with field names. This helps prevent subtle, hard-to-locate bugs.

Performance Tip

- Java conserves storage by maintaining only one copy of each method per class—this method is invoked by every object of the class. Each object, on the other hand, has its own copy of the class's instance variables (i.e., non-`static` fields). Each method of the class implicitly uses `this` to determine the specific object of the class to manipulate.

Time Class Case Study: Overloaded Constructors

- Overloaded constructors
  - Provide multiple constructor definitions with different signatures
- No-argument constructor
  - A constructor invoked without arguments
- The `this` reference can be used to invoke another constructor
  - Allowed only as the first statement in a constructor's body

```java
// Fig. 8.5: Time2.java
// Time2 class declaration with overloaded constructors.

public class Time2
{
   private int hour;   // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // Time2 no-argument constructor: initializes each instance variable
   // to zero; ensures that Time2 objects start in a consistent state
   public Time2()
   {
      this( 0, 0, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 no-argument constructor

   // Time2 constructor: hour supplied, minute and second defaulted to 0
   public Time2( int h )
   {
      this( h, 0, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 one-argument constructor

   // Time2 constructor: hour and minute supplied, second defaulted to 0
   public Time2( int h, int m )
   {
      this( h, m, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 two-argument constructor

```

No-argument constructor

Invoke three-argument constructor

```
29    // Time2 constructor: hour, minute and second supplied
30    public Time2( int h, int m, int s )
31    {
32       setTime( h, m, s ); // invoke setTime to validate time
33    } // end Time2 three-argument constructor
34
35    // Time2 constructor: another Time2 object supplied
36    public Time2( Time2 time )
37    {
38       // invoke Time2 three-argument const
39       this( time.getHour(), time.getMinute(), time.getSecond() );
40    } // end Time2 constructor with a Time2 object argument
41
42    // Set Methods
43    // set a new time value using universal time; ensure that
44    // the data remains consistent by setting invalid values to
45    public void setTime( int h, int m, int s )
46    {
47       setHour( h );   // set the hour
48       setMinute( m ); // set the minute
49       setSecond( s ); // set the second
50    } // end method setTime
51
```

Call **`setTime`** method

Constructor takes a reference to another **`Time2`** object as a parameter

Could have directly accessed instance variables of object **`time`** here

```
52    // validate and set hour
53    public void setHour( int h )
54    {
55       hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
56    } // end method setHour
57
58    // validate and set minute
59    public void setMinute( int m )
60    {
61       minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
62    } // end method setMinute
63
64    // validate and set second
65    public void setSecond( int s )
66    {
67       second = ( ( s >= 0 && s < 60 ) ? s : 0 );
68    } // end method setSecond
69
70    // Get Methods
71    // get hour value
72    public int getHour()
73    {
74       return hour;
75    } // end method getHour
76
```

```
77    // get minute value
78    public int getMinute()
79    {
80       return minute;
81    } // end method getMinute
82
83    // get second value
84    public int getSecond()
85    {
86       return second;
87    } // end method getSecond
88
89    // convert to String in universal-time format (HH:MM:SS)
90    public String toUniversalString()
91    {
92       return String.format(
93          "%02d:%02d:%02d", getHour(), getMinute(), getSecond() );
94    } // end method toUniversalString
95
96    // convert to String in standard-time format (H:MM:SS AM or PM)
97    public String toString()
98    {
99       return String.format( "%d:%02d:%02d %s",
100         ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
101         getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ) );
102   } // end method toString
103} // end class Time2
```

```
1  // Fig. 8.6: Time2Test.java
2  // Overloaded constructors used to initialize Time2 objects.
3
4  public class Time2Test
5  {
6     public static void main( String args[] )
7     {
8        Time2 t1 = new Time2();           // 00:00:00
9        Time2 t2 = new Time2( 2 );        // 02:00:00
10       Time2 t3 = new Time2( 21, 34 );   // 21:34:00
11       Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
12       Time2 t5 = new Time2( 27, 74, 99 ); // 00:00:00
13       Time2 t6 = new Time2( t4 );       // 12:25:42
14
15       System.out.println( "Constructed with:" );
16       System.out.println( "t1: all arguments defaulted" );
17       System.out.printf( "   %s\n", t1.toUniversalString() );
18       System.out.printf( "   %s\n", t1.toString() );
19
```

Call overloaded constructors

```
20        System.out.println(
21           "t2: hour specified; minute and second defaulted" );
22        System.out.printf( "   %s\n", t2.toUniversalString() );
23        System.out.printf( "   %s\n", t2.toString() );
24
25        System.out.println(
26           "t3: hour and minute specified; second defaulted" );
27        System.out.printf( "   %s\n", t3.toUniversalString() );
28        System.out.printf( "   %s\n", t3.toString() );
29
30        System.out.println( "t4: hour, minute and second specified" );
31        System.out.printf( "   %s\n", t4.toUniversalString() );
32        System.out.printf( "   %s\n", t4.toString() );
33
34        System.out.println( "t5: all invalid values specified" );
35        System.out.printf( "   %s\n", t5.toUniversalString() );
36        System.out.printf( "   %s\n", t5.toString() );
37
```

```
38       System.out.println( "t6: Time2 object t4 specified" );
39       System.out.printf( "   %s\n", t6.toUniversalString() );
40       System.out.printf( "   %s\n", t6.toString() );
41    } // end main
42 } // end class Time2Test
```

```
t1: all arguments defaulted
   00:00:00
   12:00:00 AM
t2: hour specified; minute and second defaulted
   02:00:00
   2:00:00 AM
t3: hour and minute specified; second defaulted
   21:34:00
   9:34:00 PM
t4: hour, minute and second specified
   12:25:42
   12:25:42 PM
t5: all invalid values specified
   00:00:00
   12:00:00 AM
t6: Time2 object t4 specified
   12:25:42
   12:25:42 PM
```

Questions/Answers & Discussion