6

Methods (II)

Exam 1

- Wednesday, April 8
- in-class (5% of overall grade)
- open book
- covers Chap. 1 – 6
- some theory, emphasis on coding practice
- test review and practice on Monday, April 6

Last time

- Chapter 6: all about methods
- static methods and static fields
- constants (use `final` keyword)
- mathematical methods (`java.lang.Math`)
- random number generation
- practice code: using Math and Random packages

Objectives

- review and finish Chapter 6
- scoping rules
- argument promotion
- method overloading

Chapter 6 review

- Methods – units of code to perform some function
- `static` methods can be called without the need for an object of the class
- `static` variables (fields) – one copy per class
- Constants are defined using the `final` keyword

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 (Cont.)

- **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**

Outline

Scope.java

(1 of 2)

```java
// Fig. 6.11: Scope.java
// Scope class demonstrates field and local variable scopes.

public class Scope
{
   // field that is accessible to all methods of this class
   private int x = 1;

   // method begin creates and initializes local variable x
   // and calls methods useLocalVariable and useField
   public void begin()
   {
      int x = 5; // method's local variable x shadows field x

      System.out.printf( "local x in method begin 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

```

Shadows field **x**

Display value of local variable **x**

Outline

Scope.java

(2 of 2)

```
      System.out.printf( "\nlocal x in method begin is %d\n", x );
   } // end method begin

   // create and initialize local variable x during each call
   public 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 );
   } // end method useLocalVariable

   // modify class Scope's field x during each call
   public 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 method useField
} // end class Scope
```

Shadows field **x**

Display value of local variable **x**

Display value of field **x**

Outline

ScopeTest.java

```java
// Fig. 6.12: ScopeTest.java
// Application to test class Scope.

public class ScopeTest
{
   // application starting point
   public static void main( String args[] )
   {
      Scope testScope = new Scope();
      testScope.begin();
   } // end main
} // 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
```

Argument Promotion and Casting

- **Argument promotion**
  - **Java will promote a method call argument to match its corresponding method parameter according to the promotion rules**
  - **Values in an expression are promoted to the "highest" type in the expression (a temporary copy of the value is made)**
  - **Converting values to lower types results in a compilation error, unless the programmer explicitly forces the conversion to occur**
    - **Place the desired data type in parentheses before the value**
      - **example: `(int) 4.5`**

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. 6.5 | **Promotions allowed for primitive types.**

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
- Differences in return type are irrelevant in method overloading

Outline

```java
// Fig. 6.13: MethodOverload.java
// Overloaded method declarations.

public class MethodOverload
{
   // test overloaded square methods
   public void testOverloadedMethods()
   {
      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 ) );
   } // end method testOverloadedMethods

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

   // square method with double argument
   public double square( double doubleValue )
   {
      System.out.printf( "\nCalled square with double argument: %f\n",
         doubleValue );
      return doubleValue * doubleValue;
   } // end method square with double argument
} // 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

MethodOverload.java

Outline

MethodOverloadTest.java

```java
// Fig. 6.14: MethodOverloadTest.java
// Application to test class MethodOverload.

public class MethodOverloadTest
{
   public static void main( String args[] )
   {
      MethodOverload methodOverload = new MethodOverload();
      methodOverload.testOverloadedMethods();
   } // end main
} // 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
```

Outline

MethodOverload Error.java

```java
// Fig. 6.15: MethodOverloadError.java
// Overloaded methods with identical signatures
// cause compilation errors, even if return types are different.

public class MethodOverloadError
{
   // declaration of method square with int argument
   public int square( int x )
   {
      return x * x;
   }

   // second declaration of method square with int argument
   // causes compilation error even though return types are different
   public double square( int y )
   {
      return y * y;
   }
} // 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