Advanced Programming

Static Methods and Static Variables

References and Class Parameters
Static Methods

- A static method is one that can be used without a calling object.
- A static method still belongs to a class, and its definition is given inside the class definition.
- When a static method is defined, the keyword `static` is placed in the method header:
  ```java
  public static returnedType myMethod(parameters)
  {
  . . . }
  ```
- Static methods are invoked using the class name in place of a calling object:
  ```java
  returnedValue = MyClass.myMethod(arguments);
  ```
/**
 * Class with static methods for circles and spheres.
 */

public class RoundStuff
{
    public static final double PI = 3.14159;

    /**
     * Return the area of a circle of the given radius.
     */
    public static double area(double radius)
    {
        return (PI*radius*radius);
    }

    /**
     * Return the volume of a sphere of the given radius.
     */
    public static double volume(double radius)
    {
        return ((4.0/3.0)*PI*radius*radius*radius);
    }
}
import java.util.Scanner;

public class RoundStuffDemo {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter radius:");
        double radius = keyboard.nextDouble();

        System.out.println("A circle of radius " + radius + " inches");
        System.out.println("has an area of " + RoundStuff.area(radius) + " square inches.");
        System.out.println("A sphere of radius " + radius + " inches");
        System.out.println("has an volume of " + RoundStuff.volume(radius) + " cubic inches.");
    }
}
Pitfall: Invoking a Nonstatic Method Within a Static Method

- A static method cannot refer to an instance variable of the class, and it cannot invoke a nonstatic method of the class
  - A static method has no `this`, so it cannot use an instance variable or method that has an implicit or explicit `this` for a calling object
  - A static method can invoke another static method, however
  - An exception, where static method can call a non-static one:

```java
MyClass anObject = new MyClass();
anObject.myMethod();
```
Tip: You Can Put a \texttt{main} in any Class

- Although the main method is often by itself in a class separate from the other classes of a program, it can also be contained within a regular class definition
  - In this way the class in which it is contained can be used to create objects in other classes, or it can be run as a program
  - A main method so included in a regular class definition is especially useful when it contains diagnostic code for the class
A Class with a **main** Added

(Part 1 of 4)

```java
Display 5.3  Another Class with a main Added

1 import java.util.Scanner;

2 /**
3 Class for a temperature (expressed in degrees Celsius).
4 */
5 public class Temperature
6 {
7    private double degrees;  // Celsius
8
9    public Temperature()
10    {
11        degrees = 0;
12    }
13
14    public Temperature(double initialDegrees)
15    {
16        degrees = initialDegrees;
17    }
18
19    public void setDegrees(double newDegrees)
20    {
21        degrees = newDegrees;
22    }
23
24    private double getDegrees()
25    {
26        return degrees;
27    }
28
29    public static double convertCelsiusToFahrenheit(double celsius)
30    {
31        return (celsius * 9/5) + 32;
32    }
33
34    public static void main(String[] args)
35    {
36        Scanner scanner = new Scanner(System.in);
37
38        System.out.println("Enter a temperature in Celsius:");
39        double tempCelsius = scanner.nextDouble();
40
41        System.out.println("Temperature in Fahrenheit:");
42        System.out.println(convertCelsiusToFahrenheit(tempCelsius));
43        scanner.close();
44    }
45}

Note that this class has a main method and both static and nonstatic methods.
```
A Class with a **main** Added

(Part 2 of 4)

**Display 5.3**  **Another Class with a main Added**

```java
20     public double getDegrees()  
21     {                            
22         return degrees;        
23     }

24     public String toString()   
25     {                           
26         return (degrees + " C");  
27     }

28     public boolean equals(Temperature otherTemperature)  
29     {  
30         return (degrees == otherTemperature.degrees);  
31     }
32
(continued)
```
A Class with a **main** Added
(Part 3 of 4)

Display 5.3  **Another Class with a **main** Added**

```java
/**
 * Returns number of Celsius degrees equal to
 * degreesF Fahrenheit degrees.
 */
public static double toCelsius(double degreesF)
{
    return 5*(degreesF - 32)/9;
}

public static void main(String[] args)
{
    double degreesF, degreesC;

    Scanner keyboard = new Scanner(System.in);
    System.out.println("Enter degrees Fahrenheit:");
    degreesF = keyboard.nextDouble();
    degreesC = toCelsius(degreesF);
```
A Class with a **main** Added  
(Part 4 of 4)

Display 5.3  **Another Class with a main Added**

```java
52    Temperature temperatureObject = new Temperature(degreesC);
53    System.out.println("Equivalent Celsius temperature is "+
54         temperatureObject.toString());
55  }
56 }
```

*Because main is a static method, toString must have a specified calling object like temperatureObject.*

**Sample Dialogue**

Enter degrees Fahrenheit:  
212  
Equivalent Celsius temperature is 100.0 C
Static Variables

• A *static variable* is a variable that belongs to the class as a whole, and not just to one object
  – There is only one copy of a static variable per class, unlike instance variables where each object has its own copy
• All objects of the class can read and change a static variable
• Although a static method cannot access an instance variable, a static method can access a static variable

• A static variable is declared like an instance variable, with the addition of the modifier `static`

  ```java
  private static int myStaticVariable;
  ```
Static Variables

- Static variables can be declared and initialized at the same time
  
  ```java
  private static int myStaticVariable = 0;
  ```
- If not explicitly initialized, a static variable will be automatically initialized to a default value
  - `boolean` static variables are initialized to `false`
  - Other primitive types static variables are initialized to the zero of their type
  - Class type static variables are initialized to `null`
- It is always preferable to explicitly initialize static variables rather than rely on the default initialization
public class InvocationCounter {
    private static int numberOfInvocations = 0;

    public void demoMethod() {
        numberOfInvocations++;
        // In a real example, more code would go here.
    }

    public void outputCount() {
        numberOfInvocations++;
        System.out.println("Number of invocations so far = "
                + numberOfInvocations);
    }

    public static int numberSoFar() {
        numberOfInvocations++;
        return numberOfInvocations;
    }
}
public static void main(String[] args)
{
    int i;
    InvocationCounter object1 = new InvocationCounter();
    for (i = 1; i <= 5; i++)
    {
        object1.demoMethod();
        object1.outPutCount();
    }

    InvocationCounter object2 = new InvocationCounter();
    for (i = 1; i <= 5; i++)
    {
        object2.demoMethod();
        object2.outPutCount();
    }

    System.out.println("Total number of invocations = " + numberSoFar());
}

public static void main(String[] args) {
    int i;
    InvocationCounter object1 = new InvocationCounter();
    for (i = 1; i <= 5; i++)
        object1.demoMethod();
    object1.outputCount();

    InvocationCounter object2 = new InvocationCounter();
    for (i = 1; i <= 5; i++)
    {
        object2.demoMethod();
        object2.outputCount();
    }

    System.out.println("Total number of invocations = "
                    + numberSoFar());
}

Sample Dialogue

Number of invocations so far = 6
Number of invocations so far = 8
Number of invocations so far = 10
Number of invocations so far = 12
Number of invocations so far = 14
Number of invocations so far = 16
Total number of invocations = 17
Static Variables (cont.)

• A static variable should always be defined private, unless it is also a defined constant
  – The value of a static defined constant cannot be altered, therefore it is safe to make it `public`
  – In addition to `static`, the declaration for a static defined constant must include the modifier `final`, which indicates that its value cannot be changed
    
    ```java
    public static final int BIRTH_YEAR = 1954;
    ```

• When referring to such a defined constant outside its class, use the name of its class in place of a calling object

  ```java
  int year = MyClass.BIRTH_YEAR;
  ```
The Math Class

- The Math class provides a number of standard mathematical methods
  - It is found in the java.lang package, so it does not require an import statement
  - All of its methods and data are static, therefore they are invoked with the class name Math instead of a calling object
  - The Math class has two predefined constants, E (e, the base of the natural logarithm system) and PI (π, 3.1415 . . )
    
    area = Math.PI * radius * radius;
Some Methods in the Class **Math**

(Part 1 of 5)

Display 5.6  **Some Methods in the Class Math**

The Math class is in the `java.lang` package, so it requires no import statement.

```
public static double pow(double base, double exponent)
```

Returns base to the power exponent.

**EXAMPLE**

Math.pow(2.0, 3.0) returns 8.0.

(continued)
Some Methods in the Class **Math**  
(Part 2 of 5)

<table>
<thead>
<tr>
<th>Method Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>public static double abs(double argument)</strong></td>
</tr>
<tr>
<td><strong>public static float abs(float argument)</strong></td>
</tr>
<tr>
<td><strong>public static long abs(long argument)</strong></td>
</tr>
<tr>
<td><strong>public static int abs(int argument)</strong></td>
</tr>
</tbody>
</table>

Returns the absolute value of the argument. (The method name abs is overloaded to produce four similar methods.)

**EXAMPLE**

Math.abs(-6) and Math.abs(6) both return 6. Math.abs(-5.5) and Math.abs(5.5) both return 5.5.

<table>
<thead>
<tr>
<th>Method Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>public static double min(double n1, double n2)</strong></td>
</tr>
<tr>
<td><strong>public static float min(float n1, float n2)</strong></td>
</tr>
<tr>
<td><strong>public static long min(long n1, long n2)</strong></td>
</tr>
<tr>
<td><strong>public static int min(int n1, int n2)</strong></td>
</tr>
</tbody>
</table>

Returns the minimum of the arguments n1 and n2. (The method name min is overloaded to produce four similar methods.)

**EXAMPLE**

Math.min(3, 2) returns 2.

(continued)
Some Methods in the Class **Math**
(Part 3 of 5)

**Display 5.6 Some Methods in the Class Math**

```java
public static double max(double n1, double n2)
public static float max(float n1, float n2)
public static long max(long n1, long n2)
public static int max(int n1, int n2)
```

Returns the maximum of the arguments n1 and n2. (The method name max is overloaded to produce four similar methods.)

**EXAMPLE**

Math.max(3, 2) returns 3.

```java
public static long round(double argument)
public static int round(float argument)
```

Rounds its argument.

**EXAMPLE**

Math.round(3.2) returns 3; Math.round(3.6) returns 4.

(continued)
Some Methods in the Class `Math`  

(Part 4 of 5)

Display 5.6  **Some Methods in the Class Math**

```java
public static double ceil(double argument)
```

Returns the smallest whole number greater than or equal to the argument.

**EXAMPLE**

Math.ceil(3.2) and Math.ceil(3.9) both return 4.0.

(continued)
Some Methods in the Class Math
(Part 5 of 5)

Display 5.6  **Some Methods in the Class Math**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public static double floor(double argument)</code></td>
<td>Returns the largest whole number less than or equal to the argument.</td>
<td><code>Math.floor(3.2)</code> and <code>Math.floor(3.9)</code> both return 3.0.</td>
</tr>
<tr>
<td><code>public static double sqrt(double argument)</code></td>
<td>Returns the square root of its argument.</td>
<td><code>Math.sqrt(4)</code> returns 2.0.</td>
</tr>
</tbody>
</table>
Random Numbers

• The **Math** class also provides a facility to generate pseudo-random numbers

  ```java
  public static double random()
  ```

  – A pseudo-random number appears random but is really generated by a deterministic function

  • There is also a more flexible class named **Random**

• Sample use:  `double num = Math.random();`

  Returns a pseudo-random number greater than or equal to 0.0 and less than 1.0
Wrapper Classes

• *Wrapper classes* provide a class type corresponding to each of the primitive types
  – This makes it possible to have class types that behave somewhat like primitive types
  – The wrapper classes for the primitive types `byte, short, long, float, double, and char` are (in order) `Byte, Short, Long, Float, Double, and Character`

• Wrapper classes also contain a number of useful predefined constants and static methods
Wrapper Classes

- **Boxing**: the process of going from a value of a primitive type to an object of its wrapper class
  - To convert a primitive value to an "equivalent" class type value, create an object of the corresponding wrapper class using the primitive value as an argument
  - The new object will contain an instance variable that stores a copy of the primitive value
  - Unlike most other classes, a wrapper class does not have a no-argument constructor
    
    ```java
    Integer integerObject = new Integer(42);
    ```
Wrapper Classes

• *Unboxing*: the process of going from an object of a wrapper class to the corresponding value of a primitive type
  
  – The methods for converting an object from the wrapper classes `Byte, Short, Integer, Long, Float, Double,` and `Character` to their corresponding primitive type are (in order) `byteValue, shortValue, intValue, longValue, floatValue, doubleValue,` and `charValue`
  
  – None of these methods take an argument

```java
int i = integerObject.intValue();
```
Automatic Boxing and Unboxing

• Starting with version 5.0, Java can automatically do boxing and unboxing

• Instead of creating a wrapper class object using the `new` operation (as shown before), it can be done as an automatic type cast:
  ```java
  Integer integerObject = 42;
  ```

• Instead of having to invoke the appropriate method (such as `intValue`, `doubleValue`, `charValue`, etc.) in order to convert from an object of a wrapper class to a value of its associated primitive type, the primitive value can be recovered automatically
  ```java
  int i = integerObject;
  ```
Constants and Static Methods in Wrapper Classes

• Wrapper classes include useful constants that provide the largest and smallest values for any of the primitive number types
  – For example, `Integer.MAX_VALUE`, `Integer.MIN_VALUE`, `Double.MAX_VALUE`, `Double.MIN_VALUE`, etc.

• The `Boolean` class has names for two constants of type `Boolean`
  – `Boolean.TRUE` and `Boolean.FALSE` are the Boolean objects that correspond to the values `true` and `false` of the primitive type `boolean`
Constants and Static Methods in Wrapper Classes

• Wrapper classes have static methods that convert a correctly formed string representation of a number to the number of a given type
  – The methods `Integer.parseInt`, `Long.parseLong`, `Float.parseFloat`, and `Double.parseDouble` do this for the primitive types (in order) `int`, `long`, `float`, and `double`
  – `Double.parseDouble("144.75")` returns the double value `144.75`

• Wrapper classes also have static methods that convert from a numeric value to a string representation of the value
  – For example, the expression `Double.toString(123.99);` returns the string value `"123.99"`

• The `Character` class contains a number of static methods that are useful for string processing
Some Methods in the Class **Character**

(Part 1 of 3)

The class Character is in the `java.lang` package, so it requires no import statement.

```java
public static char toUpperCase(char argument)
```

Returns the uppercase version of its argument. If the argument is not a letter, it is returned unchanged.

**EXAMPLE**
Character.toUpperCase('a') and Character.toUpperCase('A') both return 'A'.

```java
public static char toLowerCase(char argument)
```

Returns the lowercase version of its argument. If the argument is not a letter, it is returned unchanged.

**EXAMPLE**
Character.toLowerCase('a') and Character.toLowerCase('A') both return 'a'.

```java
public static boolean isUpperCase(char argument)
```

Returns true if its argument is an uppercase letter; otherwise returns false.

**EXAMPLE**
Character.isUpperCase('A') returns true. Character.isUpperCase('a') and Character.isUpperCase('%') both return false.
Some Methods in the Class **Character**
(Part 2 of 3)

### Display 5.8  Some Methods in the Class Character

**public static boolean isLowerCase(char argument)**

Returns true if its argument is a lowercase letter; otherwise returns false.

**EXAMPLE**

Character.isLowerCase('a') returns true. Character.isLowerCase('A') and Character.isLowerCase('%') both return false.

**public static boolean isWhitespace(char argument)**

Returns true if its argument is a whitespace character; otherwise returns false. Whitespace characters are those that print as white space, such as the space character (blank character), the tab character ('\t'), and the line break character ('\n').

**EXAMPLE**

Character.isWhitespace(' ') returns true. Character.isWhitespace('A') returns false.

(continued)
### Some Methods in the Class Character

(Part 3 of 3)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static boolean isLetter(char argument)</td>
<td>Returns true if its argument is a letter; otherwise returns false.</td>
</tr>
<tr>
<td>EXAMPLE</td>
<td>Character.isLetter('A') returns true. Character.isLetter('%') and Character.isLetter('5') both return false.</td>
</tr>
<tr>
<td>public static boolean isDigit(char argument)</td>
<td>Returns true if its argument is a digit; otherwise returns false.</td>
</tr>
<tr>
<td>EXAMPLE</td>
<td>Character.isDigit('5') returns true. Character.isDigit('A') and Character.isDigit('%') both return false.</td>
</tr>
<tr>
<td>public static boolean isLetterOrDigit(char argument)</td>
<td>Returns true if its argument is a letter or a digit; otherwise returns false.</td>
</tr>
<tr>
<td>EXAMPLE</td>
<td>Character.isLetterOrDigit('A') and Character.isLetterOrDigit('5') both return true. Character.isLetterOrDigit('&amp;') returns false.</td>
</tr>
</tbody>
</table>
Variables and Memory

• A computer has two forms of memory
• *Secondary memory* is used to hold files for "permanent" storage
• *Main memory* is used by a computer when it is running a program
  – Values stored in a program's variables are kept in main memory
Variables and Memory

• Main memory consists of a long list of numbered locations called *bytes*
  – Each byte contains eight *bits*: eight 0/1 digits

• The number that identifies a byte is called its *address*
  – A data item can be stored in one (or more) of these bytes
  – The address of the byte is used to find the data item when needed
Variables and Memory

• Values of most data types require more than one byte of storage
  – Several adjacent bytes are then used to hold the data item
  – The entire chunk of memory that holds the data is called its memory location
  – The address of the first byte of this memory location is used as the address for the data item

• A computer's main memory can be thought of as a long list of memory locations of varying sizes
Variables in Memory

Display 5.10  Variables in Memory

Main Memory

- **byte 0**
- **byte 1**
- **byte 2**
- **byte 3**
- **byte 4**
- **byte 5**
- **byte 6**
- **byte 7**
- **byte 8**

- **variable1** (3-byte location with address 0)
- **variable2** (2-byte location with address 3)
- **variable3** (1-byte location with address 5)
- **variable4** (3-byte location with address 6)
References

• Every variable is implemented as a location in computer memory
• When the variable is a primitive type, the value of the variable is stored in the memory location assigned to the variable
  – Each primitive type always require the same amount of memory to store its values
References

• When the variable is a class type, only the memory address (or reference) where its object is located is stored in the memory location assigned to the variable
  – The object named by the variable is stored in some other location in memory
  – Like primitives, the value of a class variable is a fixed size
  – Unlike primitives, the value of a class variable is a memory address or reference
  – The object, whose address is stored in the variable, can be of any size
References

• Two reference variables can contain the same reference, and therefore name the same object
  – The assignment operator sets the reference (memory address) of one class type variable equal to that of another
  – Any change to the object named by one of these variables will produce a change to the object named by the other variable, since they are the same object

```java
ToyClass variable1 = new ToyClass("Joe", 42);
ToyClass variable2;
variable2 = variable1; //Now both variables name the same object.
variable2.set("Josephine", 1);
System.out.println(variable1);
```

The output is

Josephine 1
Class Type Variables Store a Reference
(Part 1 of 2)

Display 5.12  Class Type Variables Store a Reference

```java
public class ToyClass
{
    private String name;
    private int number;
}
```

ToyClass sampleVariable;

*Creates the variable sampleVariable in memory but assigns it no value.*

The complete definition of the class ToyClass is given in Display 5.11.

```java
sampleVariable = new ToyClass("Josephine Student", 42);
```

*Creates an object, places the object someplace in memory, and then places the address of the object in the variable sampleVariable. We do not know what the address of the object is, but let's assume it is 2056. The exact number does not matter.*

(continued)
Class Type Variables Store a Reference

(Part 2 of 2)

For emphasis, we made the arrow point to the memory location referenced.
Assignment Operator with Class Type Variables

(Part 1 of 3)

Display 5.13  Assignment Operator with Class Type Variables

ToyClass variable1 = new ToyClass("Joe", 42);
ToyClass variable2;

We do not know what memory address (reference) is stored in the variable variable1. Let’s say it is 4068. The exact number does not matter.

Someplace else in memory:

new ToyClass("Joe", 42)

as returning a reference.

(continued)
Assignment Operator with Class Type Variables (Part 2 of 3)

Display 5.13  Assignment Operator with Class Type Variables

```
variable2 = variable1;

variable1
  4068
  ...

variable2
  4068
  ...

Someplace else in memory:

4068
  ...

Joe
  42
  ...

(continued)```
Assignment Operator with Class Type Variables (Part 3 of 3)

Display 5.13  Assignment Operator with Class Type Variables

```java
variable2.set("Josephine", 1);
```

variable1

```
4068
...
```

variable2

```
4068
...
```

Someplace else in memory:

```
...
```

4068

```
Josephine
1
...
```
Material Covered

• Chapter 5: pages 286 – 317
  – Absolute Java, 4th edition, Savitch