Comp 248
Introduction to Programming
Chapter 4 & 5 *Defining Classes*

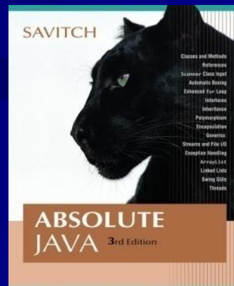
Part B

Dr. Aiman Hanna

Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides has been extracted, modified and updated from original slides of Absolute Java 3rd Edition by Savitch; which has originally been prepared by Rose Williams of Binghamton University. Absolute Java is published by Pearson Education / Addison-Wesley.

Copyright © 2007 Pearson Addison-Wesley
Copyright © 2007-2016 Aiman Hanna
All rights reserved



Overloading

- *Overloading* is when two or more methods *in the same class* have the same method name

- To be valid, any two definitions of the method name must have different *signatures*
  - A signature consists of the name of a method together with its parameter list
  - Differing signatures must have different numbers and/or types of parameters

- Overloading1.java (MS-Word file)

Overloading and Automatic Type Conversion

- If Java cannot find a method signature that exactly matches a method invocation, it will try to use automatic type conversion
- The interaction of overloading and automatic type conversion can have unintended results
- In some cases of overloading, because of automatic type conversion, a single method invocation can be resolved in multiple ways
  - Ambiguous method invocations will produce an error in Java

You Can Not Overload Operators in Java

- Although many programming languages, such as C++, allow you to overload operators (+, -, etc.), Java does not permit this
  - You may only use a method name and ordinary method syntax to carry out the operations you desire

Default Variable Initializations

- Instance class variables are automatically initialized in Java
  - `boolean` types are initialized to false
  - Other primitives are initialized to the *zero* of their type
  - Class types are initialized to `null`
- However, it is a better practice to explicitly initialize instance variables in a constructor
- Note: Local variables are not automatically initialized

The `this` Parameter

- All instance variables are understood to have **`<the calling object>.`** in front of them

- If an explicit name for the calling object is needed, the keyword **`this`** can be used
  - **`myInstanceVariable`** always means and is always interchangeable with **`this.myInstanceVariable`**

- VehicleCompare5.java (MS-Word file)

The `this` Parameter

- **`this`** ***must*** be used if a parameter or other local variable with the same name is used in the method
  - Otherwise, all instances of the variable name will be interpreted as local

```
int someVariable = this.someVariable
```

local (points to the first someVariable) **instance** (points to this.someVariable)

Information Hiding and Encapsulation

- *Information hiding* is the practice of separating how to use a class from the details of its implementation
  - *Abstraction* is another term used to express the concept of discarding details in order to avoid information overload
- *Encapsulation* means that the data and methods of a class are combined into a single unit (i.e., a class object), which hides the implementation details
  - Knowing the details is unnecessary because interaction with the object occurs via a well-defined and simple interface
  - In Java, hiding details is done by marking them **`private`**

Static Methods

- Sometimes, it is desired to use a function of a class without creating objects from this class. In such case, the method can be created as *static*
- When a static method is defined, the keyword **`static`** is placed in the method header

```
public static returnedType myMethod(parameters)
{ . . . }
```

- Static methods are invoked using the class name in place of a calling object

```
maxMiles = MetricConverter.kmToMile(maxSpeed);
```

- VehicleSearch5.java (MS-Word file)

Pitfall: Invoking a Non-static Method Within a Static Method

- A static method cannot refer to an instance variable of the class, and it cannot invoke a non-static 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

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`**

```
private static double bestPrice;
```

- VehicleSearch8.java (MS-Word file)

Overloading Constructors

- Constructors can also be overloaded to provide different object creation options

- VehicleSearch6.java (MS-Word file)

Copy Constructors

- A *copy constructor* is a constructor with a single argument of the same type as the class

- The copy constructor should create an object that is a separate, independent object, but with the instance variables set so that it is an exact copy of the argument object

- VehicleSearch7.java (MS-Word file)

- **Revisit** VehicleSearch8.java (MS-Word file)

The `StringTokenizer` Class

- The **`StringTokenizer`** class is used to recover the words or *tokens* in a multi-word **`String`**
  - You can use whitespace characters to separate each token, or you can specify the characters you wish to use as separators
  - In order to use the **`StringTokenizer`** class, be sure to include the following at the start of the file:

```
import java.util.StringTokenizer;
```

- Strings2.java (MS-Word file)

Some Methods in the `StringTokenizer` Class (Part 1 of 2)

Display 4.17 Some Methods in the Class `StringTokenizer`

The class `StringTokenizer` is in the `java.util` package.

```
public StringTokenizer(String theString)
```

Constructor for a tokenizer that will use whitespace characters as separators when finding tokens in `theString`.

```
public StringTokenizer(String theString, String delimiters)
```

Constructor for a tokenizer that will use the characters in the string `delimiters` as separators when finding tokens in `theString`.

```
public boolean hasMoreTokens()
```

Tests whether there are more tokens available from this tokenizer's string. When used in conjunction with `nextToken`, it returns `true` as long as `nextToken` has not yet returned all the tokens in the string; returns `false` otherwise.

(continued)

Some Methods in the `StringTokenizer` Class (Part 2 of 2)

Display 4.17 **Some Methods in the Class `StringTokenizer`**

```
public String nextToken()
```

Returns the next token from this tokenizer's string. (Throws `NoSuchElementException` if there are no more tokens to return.)[5]

```
public String nextToken(String delimiters)
```

First changes the delimiter characters to those in the string `delimiters`. Then returns the next token from this tokenizer's string. After the invocation is completed, the delimiter characters are those in the string `delimiters`.
(Throws `NoSuchElementException` if there are no more tokens to return. Throws `NullPointerException` if `delimiters` is `null`.)[5]

```
public int countTokens()
```

Returns the number of tokens remaining to be returned by `nextToken`.

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)

Display 5.6 Some Methods in the Class Math

```
public static double abs(double argument)
public static float abs(float argument)
public static long abs(long argument)
public static int abs(int argument)
```

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.

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

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

```
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.

```
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`**

```
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`

```
public static double floor(double argument)
```

Returns the largest whole number less than or equal to the argument.

EXAMPLE

`Math.floor(3.2)` and `Math.floor(3.9)` both return `3.0`.

```
public static double sqrt(double argument)
```

Returns the square root of its argument.

EXAMPLE

`Math.sqrt(4)` returns `2.0`.