

Comp 248

Introduction to Programming

Chapter 4 & 5 *Defining Classes*

Part C

Dr. Aiman Hanna

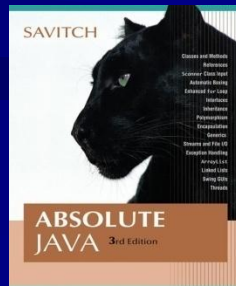
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

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



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

```
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
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:

```
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

```
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**
- 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)

Display 5.8 Some Methods in the Class Character

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

```
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'`.

```
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'`.

```
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.

(continued)

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)

Display 5.8 Some Methods in the Class Character

```
public static boolean isLetter(char argument)
```

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

EXAMPLE

`Character.isLetter('A')` returns true. `Character.isLetter('%')` and `Character.isLetter('5')` both return false.

```
public static boolean isDigit(char argument)
```

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

EXAMPLE

`Character.isDigit('5')` returns true. `Character.isDigit('A')` and `Character.isDigit('%')` both return false.

```
public static boolean isLetterOrDigit(char argument)
```

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

EXAMPLE

`Character.isLetterOrDigit('A')` and `Character.isLetterOrDigit('5')` both return true. `Character.isLetterOrDigit('&')` returns false.

Class Parameters

- All parameters in Java are *call-by-value* parameters
 - A parameter is a *local variable* that is set equal to the value of its argument
 - Therefore, any change to the value of the parameter cannot change the value of its argument
- Class type parameters appear to behave differently from primitive type parameters
 - They appear to behave in a way similar to parameters in languages that have the *call-by-reference* parameter passing mechanism

Class Parameters

- The value plugged into a class type parameter is a reference (memory address)
 - Therefore, the parameter becomes another name for the argument
 - Any change made to the object named by the parameter (i.e., changes made to the values of its instance variables) will be made to the object named by the argument, because they are the same object
 - Note that, because it still is a call-by-value parameter, any change made to the class type parameter itself (i.e., its address) will not change its argument (the reference or memory address)


Parameters of a Class Type

Display 5.14 Parameters of a Class Type

```
1 public class ClassParameterDemo
2 {
3     public static void main(String[] args)
4     {
5         ToyClass anObject = new ToyClass("Mr. Cellophane", 0);
6         System.out.println(anObject);
7         System.out.println(
8             "Now we call changer with anObject as argument.");
9         ToyClass.changer(anObject);
10        System.out.println(anObject);
11    }
12 }
```

ToyClass is defined in Display 5.11.

Notice that the method changer changed the instance variables in the object anObject.



SAMPLE DIALOGUE

```
Mr. Cellophane 0
Now we call changer with anObject as argument.
Hot Shot 42
```

The Constant `null`

- **`null`** is a special constant that may be assigned to a variable of any class type
 - `YourClass yourObject = null;`
- It is used to indicate that the variable has no "real value"
 - It is often used in constructors to initialize class type instance variables when there is no obvious object to use
- **`null`** is not an object: It is, rather, a kind of "placeholder" for a reference that does not name any memory location
 - Because it is like a memory address, use `==` or `!=` (instead of **`equals`**) to test if a class variable contains null
 - `if (yourObject == null) . . .`

Pitfall: Null Pointer Exception

- Even though a class variable can be initialized to **null**, this does not mean that **null** is an object
 - **null** is only a placeholder for an object
- A method cannot be invoked using a variable that is initialized to **null**
 - The calling object that must invoke a method does not exist
- Any attempt to do this will result in a "Null Pointer Exception" error message
 - For example, if the class variable has not been initialized at all (and is not assigned to **null**), the results will be the same

Using and Misusing References

- When writing a program, it is very important to insure that private instance variables remain truly private
- For a primitive type instance variable, just adding the **private** modifier to its declaration should insure that there will be no *privacy leaks*
- For a class type instance variable, however, adding the **private** modifier alone is not sufficient

Copy Constructor for a Class with Primitive Type Instance Variables

```
public Date(Date aDate)
{
    if (aDate == null) //Not a real date.
    {
        System.out.println("Fatal Error.");
        System.exit(0);
    }

    month = aDate.month;
    day = aDate.day;
    year = aDate.year;
}
```

Copy Constructor for a Class with Class Type Instance Variables

```
public Person(Person original)
{
    if (original == null)
    {
        System.out.println("Fatal error.");
        System.exit(0);
    }
    name = original.name;
    born = new Date(original.born);
    if (original.died == null)
        died = null;
    else
        died = new Date(original.died);
}
```

Copy Constructor for a Class with Class Type Instance Variables

- Unlike the **Date** class, the **Person** class contains three class type instance variables
- If the **born** and **died** class type instance variables for the new **Person** object were merely copied, then they would simply rename the **born** and **died** variables from the original **Person** object

```
born = original.born //dangerous
```

```
died = original.died //dangerous
```

- This would not create an independent copy of the original object

Pitfall: Privacy Leaks

- The previously illustrated examples from the **Person** class show how an incorrect definition of a constructor can result in a *privacy leak*
- A similar problem can occur with incorrectly defined mutator or accessor methods

- For example:

```
public Date getBirthDate()  
{  
    return born; //dangerous  
}
```

- Instead of:

```
public Date getBirthDate()  
{  
    return new Date(born); //correct  
}
```

Mutable and Immutable Classes

- The accessor method **getName** from the **Person** class appears to contradict the rules for avoiding privacy leaks:

```
public String getName()  
{  
    return name; //Isn't this dangerous?  
}
```

Mutable and Immutable Classes

- A class that contains no methods (other than constructors) that change any of the data in an object of the class is called an *immutable class*
 - Objects of such a class are called *immutable objects*
 - It is perfectly safe to return a reference to an immutable object because the object cannot be changed in any way
 - The **String** class is an immutable class

Mutable and Immutable Classes

- A class that contains public mutator methods or other public methods that can change the data in its objects is called a *mutable class*, and its objects are called *mutable objects*
 - Never write a method that returns a mutable object
 - Instead, use a copy constructor to return a reference to a completely independent copy of the mutable object