CSE-2102
Object Oriented Programming

Dr. Muhammad Ibrahim
Assistant Professor
Dept. of Computer Science and Engineering
University of Dhaka

Topics

- Enumerations
  - Syntax
  - Data and methods
  - Motivation
- Type wrappers
  - Numeric and non-numeric wrappers
  - Built-in functions
- Autoboxing

Enumerations

- A list of named constants
- Common to many languages
  - However, Java didn’t have this feature until recently (from JDK 5).
- A major difference between Java’s enumeration and that of many other languages: in Java enumerations are objects.
  - Thus the capabilities of enumerations are expanded because enumerations can have instance variables, constructors and methods

Enumerations

- An enumeration is declared by `enum` keyword
  - They must not be declared inside a method.

```
enum Student {
  very_good, good, moderate, poor
}
```

- The above segment creates in the background a class named `Student`
  - `very_good, good, moderate, poor` are called enumeration constants.
  - They are public, static and final member of `Student` class.
  - Their type is the “Student”, for this reason these are called “self-typed”.
    - `public static final Student good = good;`

Enumerations

- Once an enumeration is declared, we can declare variable of this type
  - However, we cannot use `new` operator here.
    - `Student s1;`
- The enumeration variables can only take the named constants
  - `s1 = Student.very_good;`
    - Now you can see why the type of the members of enumeration class are the type of the class itself.
- Enumeration variables can be checked for equality (using ==)
  - So they can be put in switch statement
- Printing named constants: simply displays the value.
- All enumerations automatically inherit `java.lang.Enum` class.
  - Some commonly used methods of Enum class are discussed next.

Commonly Used Methods

- public static *enum-type*[ ] values()
  - Returns an array of enum-type
- public static *enum-type* valueOf(String str)
  - Just changes the type from String to enum-type
- final int ordinal()
  - Returns the “index” of the named constant in question

Adding Constructors, Data and Methods

- As mentioned earlier, enumerations can have constructors, data members and methods because they are implemented in the background as classes.
  - Example: next slide
- Constructors can also be overloaded.

```
public class enum_advanced {
    enum Student{
        very_good(5.0), good(4.0), moderate(3.0), poor(2.0);
        private double gradepoint;
        Student(double g) {
            gradepoint = g;
        }
        double get_gradepoint(){
            return gradepoint;
        }
    }    //enum ends
    public static void main(String args[]){
        Student s = Student.good;
        System.out.println("Gradepoint of good student is: " +
s.get_gradepoint());
    }
}
```

Enumerations

- Now lets see a more realistic example…

Why Do We Need Enumerations Anyway?

- True, we can work without enumerations
  - We can use string constants instead

  In our previous example, we could write:

```
String ask () { … return “NO”; …. }
static void answer (String result) {
switch(result):
case “NO”: ...
```

- However, using enumerations reduces the error due to typos of programmers.
  - In the above, we could mistakenly write `case “NOO”:` which produces no compile-time error.
    - But in enumerations we get a compile time error
- In essence, enumerations increase readability and correctness of programs.

Type Wrappers

- Recall that primitive types such as int, float, double, char are faster to be processed than objects
  - They are not part of Java's (huge) class hierarchy, i.e., they don't inherit Object class
- Despite this, there will be situations when we do need to use "object versions" of this primitive types
  - We can simply put an integer inside an object to create an "object version"
  - A very common such situation is when we want to pass an integer to some method using call-by-reference scheme
    - Recall from our previous module that in a thread we need to extensively use call-by-reference scheme to pass parameters and to get returned results.
- Also, many built-in data structures in Java are implemented as objects, so to use these we must use "object versions" of primitive types
- For these two reasons Java provides type wrapping mechanism that wraps primitive types with classes.

List of Type Wrappers

- Available type wrappers in Java:
  - Numeric: Double, Float, Long, Integer, Short, Byte
  - Non-numeric: Character and Boolean.
- These classes provide a good number of useful methods

Non-Numeric Type Wrappers

- Character class:
  - Constructor: `Character(char ch)`
  - `char charValue();`
- Boolean class:
  - Constructors: `Boolean(boolean b);` and `Boolean(String b);`
  - `boolean booleanValue();`

Numeric Type Wrappers

- Byte, Short, Integer, Long, Float, and Double.
- All of these inherit abstract class called Number
  - Declares a number of methods which are overridden by these classes
- Methods
  - `byte byteValue( )`
  - `double doubleValue( )`
  - `float floatValue( )`
  - `int intValue( )`
  - `long longValue( )`
  - `short shortValue( )`
- Each of the six classes mentioned above implement all of these methods
  - The methods return the corresponding values

Numeric Type Wrappers

- Each of these classes offer two constructors
  - For numbers
  - For strings
  - E.g.: `Integer(int num)` and `Integer(String str)`
    - NumberFormatException might be thrown if str is not a valid integer
- All of the type wrappers (both numeric and non-numeric) override toString() method to display the value of the stored in the object in question
  - So we can print the value simply by passing the object into `System.out.println`

Two Definitions Regarding Type Wrappers

- Boxing: the process of wrapping a primitive type
  - `Integer i_ob = new Integer(10);`
- Unboxing: the process of unwrapping a primitive type
  - `int i = i_ob.intValue();`

Autoboxing

- Beginning with JDK 5, autoboxing and auto-unboxing features have been added.
- Autoboxing: `Integer i_Ob = 100; // autobox an int`
  - Java automatically creates an object of type Integer and puts 100 inside it.
- Auto-unboxing: `int i = i_ob; // auto-unbox`
  - No need to call intValue()
- Can be applied to both numeric and non-numeric wrappers
- Can occur when passing parameters to methods or returning results from methods
- Can occur in expressions

Final Comment on Autoboxing

- Autoboxing and auto-unboxing
  - relieves the programmer greatly from writing otherwise tedious code and
  - reduces chance of some errors

```
Integer iOb = 1000; // autobox the value 1000
int i = iOb.byteValue(); // manually unbox as byte !!!
System.out.println(i); // does not display 1000 !
```

- However, should be used with care and wisdom, i.e., when necessary, otherwise overhead may ensue

```
// A bad use of autoboxing/unboxing because it increases overhead!
Double a, b, c;
a = 10.0;
b = 4.0;
c = Math.sqrt(a*a + b*b);
System.out.println("Hypotenuse is " + c);
```

Autoboxing

- Beginning with JDK 5, autoboxing and auto-unboxing features have been added.
- Autoboxing: `Integer i_Ob = 100; // autobox an int`
  - Java automatically creates an object of type Integer and puts 100 inside it.
- Auto-unboxing: `int i = i_ob; // auto-unbox`
  - No need to call intValue()
- Can be applied to both numeric and non-numeric wrappers
- Can occur when passing parameters to methods or returning results from methods
- Can occur in expressions

Topics

Annotations

Annotation: Motivation

```
public class myclass {
  void f () { }
}
public class yourclass extends myclass {
  void f() { }
}
```

- Here yourclass is overriding the f() method of myclass.
- Suppose for our code it is mandatory to override f() in yourclass. But if we mistakenly write `void ff() { }` instead of `void f() { }`, will there be any compile time error? No!

Annotation: Motivation (Contd.)

```
public class myclass {
  void f () { }
}
public class yourclass extends myclass {
  @Override
  void f() { }
}
```

- Annotations help us in this regard
  - This mechanism provides a way to enforce that f() method is overridden in yourclass
  - Type @Override before `void f () { }` in yourclass.
- Other uses of annotations include giving the compiler some additional information about the code, for example, suppressing warning message etc.

Annotations (or Metadata)

- Annotation is a feature of Java that facilitates putting up supplementary information in a code for compiler or JVM and/or later use (by humans).
  - The code's logic remains unchanged
- Why so important you wonder? Because for a large program correct documentation becomes of utmost importance
  - For the programmer himself
  - For future (re)use of programs
- Annotation provides a systematic way to put information into a code so that they can be maintained, manipulated and efficiently used later on.
- Also called metadata, that is, data about data (code).

Some Commonly Used Built-in Annotations

- @Override
  - To ensure at compile time that a method of superclass is overridden in a subclass
  - Used only on methods
- @SuppressWarnings(“...”)//the name of the warning should be inside “”
  - To stop displaying some specific warning message or messages.
- @FunctionalInterface
  - To make sure an interface is a functional interface, i.e., contains one and only one abstract method.
- @Depcrecated
  - To issue a warning that a method should no longer be used.

End of Lecture 22.