Advanced Web Development In Java

Java/J2EE

By Bhuwan Prasad Upadhyay

Advance Java Web Development

Week 2

Part 1 – Advanced Features

- Java Generics and type safety
- Auto-boxing /un-boxing
- Java Enumerations
- Introduction to Enhanced 'For Loop' (for each loop) in java
- Introduction to enhanced if/else
- Java mutable objects – Introduction to StringBuffer and StringBuilder object
- String Tokenizer
- Comparing and identifying objects
- Class/Type casting in java
- Static methods and variables

Java Generics and type safety

- In a nutshell, generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.
- type safety is the extent to which a programming language discourages or prevents type errors. A type error is erroneous or undesirable program behavior caused by a discrepancy between differing data types for the program's constants, variables, and methods (functions), e.g., treating an integer (int) as a floating-point number (float).

Benefits of Generic Code over Non-generic Code

- Code that uses generics has many benefits over non-generic code:
  - Stronger type checks at compile time.
  - Elimination of casts.

```
//without generics requires casting
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0); //Need Cast
```

```
//with generics not requires casting
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

  - Enabling programmers to implement generic algorithms.

Keep On Mind About Java Generics

- Generics are implemented using Type Erasure
- Generics does not support sub-typing
- You can't create Generic Arrays
- Use of wildcards with extends or super to increase API flexibility
- Use of Multiple Bounds
- Java Generic not support for primitive data types.

Auto-boxing /un-boxing

- Conversion of a primitive type to the corresponding reference type(wrapper class) is called *auto-boxing*.

```
Character ch = 'a';
```

- Conversion of the reference type(wrapper class) to the corresponding primitive type is called *unboxing*.

```
Integer i = new Integer(-8);
//Unboxing through method invocation
int absVal = absoluteValue(i);

public static int absoluteValue(int i) {
    return (i < 0) ? -i : i;
}
```

Java Enumerations

- An *enum type* is a special data type that enables for a variable to be a set of predefined constants.
- In the Java programming language, you define an enum type by using the *enum* keyword. For example, you would specify a days-of-the-week enum type as:

```
public enum Day {

    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

}
```

- You should use enum types any time you need to represent a fixed set of constants where you know all possible values at compile time.

Introduction to Enhanced 'For Loop' (for each loop) in java

```
List ints = Arrays.asList(1,2,3);
int s = 0;
for (int n : ints) {
    s += n;
}
```

EQUIVALENT

```
for (Iterator it = ints.iterator(); it.hasNext();) {
    int n = it.next();
    s += n;
}
```

Introduction to enhanced if/else

- **"? :" Ternary operator**

```
int x = true ? 10 : 120;
```

x=10

```
int x = false ? 10 : 120;
```

x=120

Java mutable objects – Introduction to StringBuffer and StringBuilder object

- **StringBuilder/StringBuffer** objects are like String objects, except that they can be modified. Internally, these objects are treated like variable-length arrays that contain a sequence of characters. At any point, the length and content of the sequence can be changed through method invocations.
- StringBuilder is not thread safe but StringBuffer is thread safe.
- Both are mutable but String is Immutable class

String Tokenizer

- **StringTokenizer** class allows an application to break a string into tokens.
- This class is a legacy class that is retained for compatibility reasons although its use is discouraged in new code.

Comparing and identifying objects

- Comparing Objects By:
  1. `==` Operator (Reference Comparison)
  2. `equals(Object o)` Method (Value Comparison)
- Identifying Objects :
  - `instanceof` Operator (To identify class type of object)

Class/Type casting in java

- Implicit Casting
  - An implicit cast means you don't have to write code for the cast

```
int i = 10000;
float f = i;
```

- Explicit Casting
  - An explicit cast means you need to write code for the cast

```
float a = 100.001f;
int b = (int)a;
```

Static methods, variables and block

- static variable [*<class-name>.a*]
  - "static" Keyword = Class Variables
  - no "static" Keyword = Instance Variables

```
static int a = 10;
```

- static method [*<class-name>.add(10,20)*]

```
static int add(int a, int b) {
    return a + b;
}
```

- static block [executed when a class is first loaded in to the JVM]

```
static {
    System.out.println("Hello Static Block.");
}
```

Part 2 – Object Collaborations

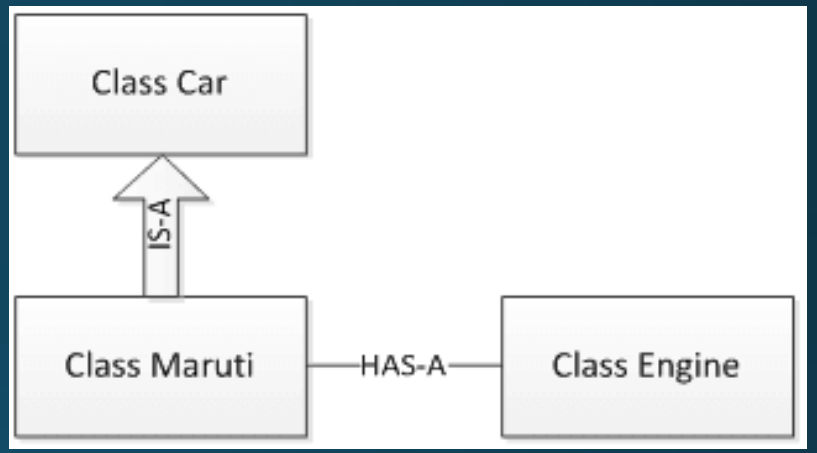
- Introduction to Object Composition and Class Inheritance
- 'Has a' and 'Is a' relationship
- Java Class Inheritance
- Java Interface and Abstract Classes
- Inner classes
- Java Object Cloning – Shallow and Deep Copy
- Java Serialization
- Polymorphism – Method Overriding and Overloading

Introduction to Object Composition and Class Inheritance

- Inheritance
  - Inheritance is convenient way to classify similar behaving objects as having entirely common root structure and behavior.
  - But what about objects that aren't the same ????
- Composition
  - Common behavior can be provided through interfaces and a behavioral composite.

‘Has a’ and ‘Is a’ relationship

- **IS-A Relationship :**
  - This refers to inheritance or implementation.
  - Expressed using keyword “extends”.
  - Main advantage is code reusability.



- **HAS-A Relationship :**
  - Has-A means an instance of one class “has a” reference to an instance of another class or another instance of same class.
  - It is also known as “composition” or “aggregation”.
  - There is no specific keyword to implement HAS-A relationship but mostly we are depended upon “new” keyword.

Java Interface and Abstract Classes

- Interface – Expose method outside the world

```
interface Play {
    void start();
    void run();
}
```

- Abstract Classes
  - An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be sub classed.
  - An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

Java Interface vs Abstract Classes

- Abstract classes are similar to interfaces. You cannot instantiate them.
- Abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods.
- With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public.

Nested classes (static nested/inner classes)

- Why ?
  - It is a way of logically grouping classes that are only used in one place
  - It increases encapsulation
  - It can lead to more readable and maintainable code

```
class OuterClass {

    static class StaticNestedClass {

    }
    class InnerClass {

    }
}
```

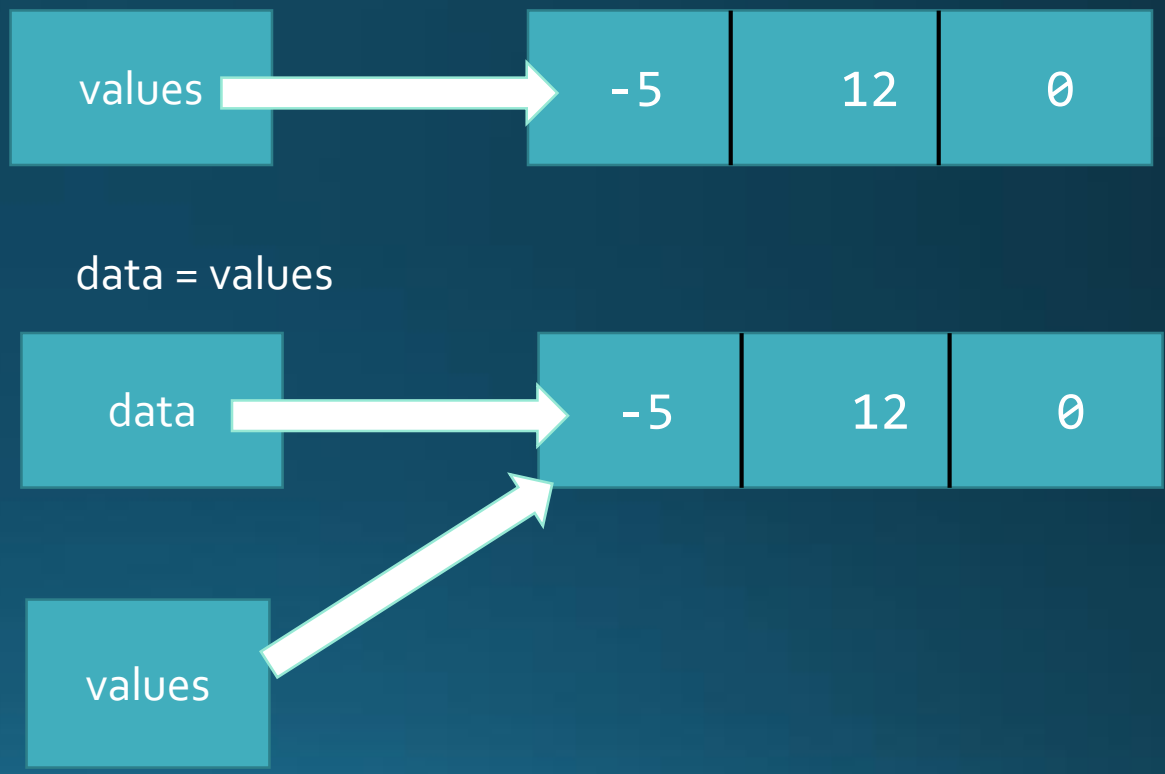
```
Static inner class
----------------------
#Access
OuterClass.StaticNestedClass

#create an object
OuterClass.StaticNestedClass nestedObject =
     new OuterClass.StaticNestedClass();

# Inner class cannot define any static members
itself
```

Java Object Cloning – Shallow and Deep Copy

- The **object cloning** is a way to create exact copy of an object.
- Shallow Copy
  - A shallow copy can be made by simply copying the reference.
- Deep Copy
  - A deep copy means actually creating a new array/object and copying over the values.

Shallow Copy

```
public class ShallowCopy {

    private int[] data;

    // makes a shallow copy of values
    public ShallowCopy(int[] values) {
          data = values;
    }

     public void showData() {
          System.out.println(Arrays.toString(data));
     }

}

public class ShallowCopyDemo {

     public static void main(String[] args) {
          int[] vals = {-5, 12, 0};
          ShallowCopy copy = new ShallowCopy(vals);
          copy.showData(); // prints out [-5, 12, 0]
          vals[1] = 25;
          copy.showData(); // prints out [-5, 25, 0]
     }
}
```



Deep Copy

```
public class DeepCopy {
     private int[] data;

     // makes a deep copy of values
     public DeepCopy(int[] values) {
          data = new int[values.length];
          for (int i = 0; i < data.length; i++) {
            data[i] = values[i];
          }
     }
     public void showData() {
          System.out.println(Arrays.toString(data));
     }

}
public class DeepCopyDemo {

     public static void main(String[] args) {
          int[] vals = {-5, 12, 0};
          DeepCopy copy = new DeepCopy(vals);
          copy.showData(); // prints out [-5, 12, 0]
          vals[1] = 25;
          copy.showData(); // prints out [-5, 12, 0]
     }
}
```


```
data = new int[values.length];
```

Object#clone()

- Class must implements `Cloneable` interface and override clone() methods.

```
public class CloneDemo {

    public static void main(String[] args)
         throws CloneNotSupportedException {
        Student student = new Student(12);
        Student object = (Student) student.clone();
        object.setA(20);
        System.out.println(student + " " + object);
    }
}
```

```
class Student implements Cloneable {

    private int a;
    public Student(int a) { this.a = a; }
    public int getA() { return a; }
    public void setA(int a) { this.a = a; }
    @Override
    public String toString() {
        return "Student{" + "a=" + a + '}';
    }

    @Override
    protected Object clone() throws
     CloneNotSupportedException {
        return super.clone();
    }

}
```

Java Serialization

- To serialize an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object. A Java object is serializable if its class or any of its superclasses implements either the *`java.io.Serializable`* interface or its subinterface, *`java.io.Externalizable`*.
- Deserialization is the process of converting the serialized form of an object back into a copy of the object.
- Serialization is used for lightweight persistence and for communication via sockets or Java Remote Method Invocation (Java RMI)

Polymorphism – Method Overriding and Overloading

- **Method overriding** is when a child class redefines the same method as a parent class, with the same parameters.
- **Method overloading** is defining several methods in the same class, that accept different numbers and types of parameters. In this case, the actual method called is decided at compile-time, based on the number and types of arguments.

You Can find me:

Twitter : @DeveloperBhuwan

Portfolio : https://devbhuwan.github.io/

My Blog : https://developerbhuwan.blogspot.com/

End Week-2 : Happy Coding.....