Object-Oriented Techniques

- Java is an object-oriented (OO) language in the tradition of Simula-67, SmallTalk, and C++.
- It borrows syntax from C++ and ideas from SmallTalk.
- The Java API has been designed and built on the OO model.

Formatting Objects for Printing with toString()

- If you print any object, java compiler internally invokes the toString() method on the object. Java "knows" that every object has a toString() method because java.lang.Object has one and all classes are ultimately subclasses of Object.
- The default implementation, in java.lang.Object, is just prints the class name, an @sign, and the object's hashCode() value.

Example:

```
public class ToStringWithout {

    int x, y;

    /** Simple constructor */
    public ToStringWithout(int anX, int aY) {
        x = anX; y = aY;
    }

    /** Main just creates and prints an object */
    public static void main(String[] args) {
```

```
        ToStringWithout  S=new ToStringWithout(42,  86);

            System.out.println(S);
    }
}
```

Output:

ToStringWithout@4617c264

toString()

- If you want to represent any object as a string, **toString() method** comes into existence.
- The toString() method returns the string representation of the object.
- The toString() method inherited from java.lang.Object.
- So overriding the toString() method, returns the desired output.

Example:

```
public  class ToStringWithout  {
    int x, y;

    /** Simple constructor  */
    public  ToStringWithout(int  anX, int aY) {
        x = anX; y = aY;
    }

    //override the toString()
    public  String toString() {
        return "ToStringWith    [" + x + "," + y + "]";
    }
```

```
    /** Main just creates and prints an object */
    public static void main(String[] args) {

        ToStringWithout S=new ToStringWithout(42, 86);

            System.out.println(S);
    }
}
```

Output:

ToStringWith [42,86]

Overriding the equals() and hashCode() Methods

- Java.lang.object has two very important methods :
    - public boolean equals(Object obj) and
    - public int hashCode().

equals() method

- In java equals() method is used to compare equality of two Objects.
- Simply checks if two Object references (say x and y) refer to the same Object. i.e. It checks if x == y.
- Some principles of equals() method of Object class :

- **reflexive**
  - x.equals(x) must be true.

- **symmetrical**
  - x.equals(y) must be true if and only if y.equals(x) is also true.

- **transitive**
  - If x.equals(y) is true and y.equals(z) is true, then x.equals(z) must also be true.

- **repeatable**
  - Multiple calls on x.equals(y) return the same value (unless state values used in the comparison are changed, as by calling a set method).

- cautious
  - x.equals(null) must return false rather than accidentally throwing a NullPointerException.

Example:

```
public class EqualsDemo {
      static int a = 10, b=20;
   int c;
   // Constructor
   EqualsDemo()
   {
      System.out.println("Addition of 10 and 20 : ");
      c=a+b;
      System.out.println("Answer : "+c);
   }
```

```
    // Driver code
    public  static void main(String args[])
    {
        System.out.println("1st  object created...");
        EqualsDemo obj1 = new EqualsDemo();
        System.out.println("2nd  object created...");
        EqualsDemo obj2 = new EqualsDemo();
        EqualsDemo obj3 = obj1;
        System.out.println("obj1  == obj2 :" + obj1.equals(obj2));
        System.out.println("obj1  == obj3 :" + obj1.equals(obj3));
    }
}
```

Output:

```
1st object created...
Addition of 10 and 20 :
Answer : 30
2nd object created...
Addition of 10 and 20 :
Answer : 30
obj1 == obj2 :false
obj1 == obj3 :true
```

hashCode() method

- The hashCode() method is supposed to return an int that should uniquely identify different objects.

- Hashcode value is mostly used in hashing based collections like HashMap, HashSet, HashTable….etc.

- A properly written hashCode() method will follow these rules:
    - It is repeatable.
        - hashCode(x) must return the same int when called repeatedly, unless set methods have been called.
    - It is consistent with equality.
        - If x.equals(y), then x.hashCode() must == y.hashCode().
    - Distinct objects should produce distinct hashCodes
        - If !x.equals(y), it is not required that x.hashCode() != y.hashCode(), but doing so may improve performance of hash tables (i.e., hashes may call hashCode() before equals()).

Example:

```
class SomeClass{

    }

public  class PrintHashCodes {

    /** Some objects to hashCode() on */
    protected static Object[] data = {
                new PrintHashCodes(),
                new java.awt.Color(0x44, 0x88, 0xcc),
                new SomeClass()
        };
```

```
    public  static void main(String[] args) {

        System.out.println("About  to hashCode " + data.length + "
                                                        objects.");
        for (int i=0; i<data.length; i++) {
            System.out.println(data[i].toString()  + " --> " +
                                                data[i].hashCode());
        }
        System.out.println("All  done.");
    }
}
```

Output:

```
About to hashCode 3 objects.
PrintHashCodes@27d6c5e0 --> 668386784
java.awt.Color[r=68,g=136,b=204] --> -12285748
SomeClass@12edcd21 --> 317574433
All done.
```

Example: Java program to illustrate how hashCode() and equals() methods work.

```
import java.io.*;

    class Geek
    {

       public String name;
       public int id;
```

```
Geek(String name, int id)
{

    this.name = name;
    this.id = id;
}

@Override
public boolean equals(Object obj)
{

// checking if both the object references are
// referring to the same object.
if(this == obj)
        return true;

    // it checks if the argument is of the
    // type Geek by comparing the classes
    // of the passed argument and this object.
    // if(!(obj instanceof Geek)) return false; ---> avoid.
    if(obj == null || obj.getClass()!= this.getClass())
        return false;

    // type casting of the argument.
    Geek geek = (Geek) obj;

    // comparing the state of argument with
    // the state of 'this' Object.
    return (geek.name == this.name && geek.id == this.id);
}

@Override
public int hashCode()
{
```

```
        // We are returning the Geek_id
        // as a hashcode value.
        // we can also return some
        // other calculated value or may
        // be memory address of the
        // Object on which it is invoked.
        // it depends on how you implement
        // hashCode() method.
        return this.id;
    }

}

//Driver code
public class HashCodeEqualsTo {


    public static void main (String[] args)
    {

        // creating the Objects of Geek class.
        Geek g1 = new Geek("aa", 1);
        Geek g2 = new Geek("aa", 1);

        // comparing above created Objects.
        if(g1.hashCode() == g2.hashCode())
        {

            if(g1.equals(g2))
                System.out.println("Both Objects are equal. ");
            else
                System.out.println("Both Objects are not equal. ");

        }
        else
```

```
            System.out.println("Both  Objects are not equal. ");
        }
    }
```

Output:

Both Objects are equal.

Using Inner Classes

- Java inner class or nested class (a class within a class).
- Java inner class is a class which is declared inside the class or interface.
- The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.
- There are basically four types of inner classes in java.
    - Nested Inner class
    - Static nested classes
    - Method Local inner classes
    - Anonymous inner classes

1) Nested Inner class

- To access the inner class, create an object of the outer class, and then create an object of the inner class.

Example:

```
class OuterClass {
    int x = 10;

    class InnerClass {
        int y = 5;
    }

    //    //If you don't want outside objects to access
    //    //the inner class, declare the class as private
    //    private class InnerClass {
    //        int y = 5;
    //      }


}

public  class InnerClassExample  {

    public  static void main(String[] args) {

        //create an object of the outer class
        OuterClass myOuter  = new OuterClass();

        //create an object of the inner class by using
        //an object of the outer class
        OuterClass.InnerClass myInner  = myOuter.new  InnerClass();

        System.out.println(myInner.y  + myOuter.x);
```

```
    }
}
```

Output:

```
15
```

2) Static nested classes

- An inner class can also be static, which means that you can access it without creating an object of the outer class.

Example:

```
class OuterClass {
    int x = 10;

    static class InnerClass {
      int y = 5;
    }
}

public  class InnerClassExample  {

    public  static void main(String[]  args) {

        //create an object of the inner class without creating
        // an object of the outer class
        OuterClass.InnerClass  myInner  = new
                                          OuterClass.InnerClass();
        System.out.println(myInner.y);
    }
}
```

Output:

```
5
```

3) Method Local inner classes

- In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

- A method-local inner class can be instantiated only within the method where the inner class is defined.

Example:

```
public class Outerclass {

    //instance method of the outer class
    void outermethod() {
        int num = 23;

        //inner class define inside the method of outer class.
        class InnerClass {

                //instance method of the inner class
                public void innermethod() {
                    System.out.println("This is method inner
                                        class " + num);
                }
        } // end of inner class

        // Accessing the inner class
        InnerClass inner = new InnerClass();
        inner.innermethod();
    }
```

```
    public static void main(String args[]) {

        // Accessing the outer class
        Outerclass outer = new Outerclass();
        outer.outermethod();
    }
}
```

Output:

```
This is method inner class 23
```

4) Anonymous inner classes

- An inner class declared without a class name is known as an **anonymous inner class**.

- In case of anonymous inner classes, we declare and instantiate them at the same time.

- Generally, they are used whenever you need to override the method of a class or an interface.

- The syntax of an anonymous inner class is as follows –

```
AnonymousInner an_inner = new AnonymousInner() {
        public void my_method() {
        ........
        ........
        }
};
```

- The anonymous inner class is created in two ways.
    - *As subclass of specified type*
    - *As implementer of the specified interface*

Example: The anonymous inner class is created *as subclass of specified type*

```
class Demo {
	void show() {
		System.out.println("i am in show method of super class");
	}
}

public class InnerClassExample {

	// An anonymous class with Demo as base class
	static Demo d = new Demo() {

		//method of anonymous class
		void show() {

			//method of Demo class
			super.show();
			System.out.println("i am in anonymous class");
		}
	};

	public static void main(String[] args){
		d.show();
	}
}
```

Output:

i am in show method of super class
i am in anonymous class

Note: In the above code, we have two class Demo and InnerClassExample. Here demo act as super class and anonymous class acts as a subclass, both classes have a method show(). In anonymous class show() method is overridden.

Example: The anonymous inner class is created *as implementer of the specified interface*

```
public class InnerClassExample {

    // An anonymous class that implements Hello interface
    static Hello h = new Hello() {
        public void show() {
            System.out.println("i am in anonymous class");
        }
    };

    public static void main(String[] args) {
            h.show();
        }
    }

interface Hello {
    void show();
}
```

Output:

i am in anonymous class

Note: In above code we create an object of anonymous inner class but this anonymous inner class is an implementer of the interface Hello. Any anonymous inner class can implement only one interface at one time. It can either extend a class or implement interface at a time.

Polymorphism/Abstract Methods

Problem

- You want each of a number of subclasses to provide its own version of one or more methods.

Solution

- Make the method abstract in the parent class; this makes the compiler ensure that each subclass implements it.

Abstract class in Java

- A class which is declared as abstract is known as an abstract class.
- It can have abstract and non-abstract methods.
- It needs to be extended and its method implemented.
- It cannot be instantiated.

Abstract Method in Java

- A method which is declared as abstract and does not have implementation is known as an abstract method.

Polymorphism in Java

- Polymorphism is the ability of an object to take on many forms.
- Java makes software more reliable and maintainable with the use of polymorphism.
- Polymorphism is a great boon for software maintenance: if a new subclass is added, the code in the main program does not change.

Example:

```
//parent class
public abstract class Shape {
	protected int x, y;
	public abstract double computeArea();
}


//
public class Rectangle extends Shape {
	double width=5, height=7;
	public double computeArea() {
		return width * height;
	}
}

//
public class Circle extends Shape {
	double radius=5;
	public double computeArea() {
		return Math.PI * radius * radius;
	}
}
```

```
//
 public  class Triangle extends Shape {
        double  a=3.0, b=4, c=5;
        public  double  computeArea() {
                double  s = (a + b + c) / 2;
                return  Math.sqrt(s * (s - a) * (s - b) * (s - c));
        }
}

import  java.util.ArrayList;
import  java.util.Collection;

public  class PolymorphismAbstract  {

         public  static void main(String[] args) {

                 // created in a Constructor,  not shown
                Collection<Shape> allShapes;
                allShapes = new ArrayList<>();

                allShapes.add(new Circle());
                allShapes.add(new Rectangle());
                allShapes.add(new Triangle());

                // Iterate over all the Shapes, getting their areas;
                double  totalAreas = 0.0;
                int i = 1;

                for (Shape s : allShapes) {
                                System.out.println("The  " + i + " Area : " +
                                                                s.computeArea());
                        totalAreas += s.computeArea();
                        i++;
```

```
            }

            System.out.println("Total  Area = " + totalAreas);
        }
}
```

Output:

```
The 1 Area: 78.53981633974483
The 2 Area: 35.0
The 3 Area: 6.0
Total Area = 119.53981633974483
```

Passing Values

Problem

- You need to pass a number like an int into a routine and get back the routine's updated version of that value in addition to the routine's return value.

Solution

- Use a specialized class such as the MutableInteger class presented here.

- MutableInteger, that is like an Integer but specialized by omitting the overhead of Number and providing only the set, get, and incr operations.

- The latter is overloaded to provide a no-argument version that performs the increment (++) operator on its value, and also a one-integer version that adds that increment into the value (analogous to the += operator).

- Because Java doesn't support operator overloading, the calling class has to call these methods instead of invoking the operations syntactically, as you would on an int.

Example:

```
public class StringParse {

    /** This is the function that has a return value of true but
    * also "passes back" the offset into the String where a
    * value was found. Contrived example!
    */
    public static boolean parse(String in, char lookFor,
                                MutableInteger whereFound) {

        int i = in.indexOf(lookFor);

        if (i == -1)
            return false; // not found
        whereFound.setValue(i); // say where found
        return true; // say that it was found
    }

    public static void main(String[] args) {

        // Create object of MutableInteger class
        MutableInteger mi = new MutableInteger();
        String text = "Hello, World";
        char c = 'W';
        if (parse(text, c, mi)) {
            System.out.println("Character " + c + " found at offset "
                                + mi + " in " + text);
        }
```

```
            else {
                System.out.println("Not found");
            }
        }
}


/** A MutableInteger is like an Integer but mutable, to avoid the
* excess object creation involved in
* c = new Integer(c.getInt()+1)
* which can get expensive if done a lot.
* Not subclassed from Integer, since Integer is final (for performance :-
))
*/
public class MutableInteger {

            private int value = 0;

            public MutableInteger(int i) {
                value = i;
            }

            public MutableInteger() {
                this(0);
            }

            public int incr() {
                value++;
                return value;
            }

            public int incr(int amt) {
                value += amt;
                return value;
```

```
    }

    public int decr() {
        value--;
        return value;
    }

    public int setValue(int i) {
        value = i;
        return value;
    }

    public int getValue() {
        return value;
    }

    public String toString() {
        return Integer.toString(value);
    }

    public static String toString(int val) {
        return Integer.toString(val);
    }

    public static int parseInt(String str) {
        return Integer.parseInt(str);
    }
}
```

Output:

Character W found at offset 7 in Hello, World

Using Typesafe Enumerations

Problem

- You need to manage a small list of discrete values within a program.

Solution

- Use the Java enum mechanism.

Java Enums

- The **Enum in Java** is a data type which contains a fixed set of constants.

- Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change).

- The Java enum constants are static and final implicitly. It is available since JDK 1.5.

- The **enum** data type (also known as Enumerated Data Type) is used to define an enum in Java.

- Java Enum internally inherits the *Enum class*, so it cannot inherit any other class, but it can implement many interfaces.

- Java enum can have fields, constructors, methods, and main methods.

- Enums can be used in switch statements.

Example: A code sample showing the definition of a typesafe enum. Simply, iterate over the array returned by the enum class's inherited values() method.

```
public class EnumExample {

        //defining the enum inside the class
        public enum Media {
            BOOK, MUSIC_CD, MUSIC_VINYL, MOVIE_VHS,
            MOVIE_DVD;
        }

        //main method
        public static void main(String[] args) {

            //traversing the enum
            for (Media m : Media.values())
                System.out.println(m);
    }
}
```

Output:

```
BOOK
MUSIC_CD
MUSIC_VINYL
MOVIE_VHS
MOVIE_DVD
```

Example: Simple program of applying Enum on a switch statement.

```
class EnumExample{

   enum Day{SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
                                THURSDAY, FRIDAY, SATURDAY}

   public  static void main(String args[]) {

   Day day=Day.MONDAY;

   switch(day) {

       case SUNDAY:
              System.out.println("Sunday");
              break;

       case MONDAY:
              System.out.println("Monday");
              break;

       case TUESDAY:
              System.out.println("Tuesday");
              break;

       case WEDNESDAY:
              System.out.println("Tuesday");
              break;

       case THURSDAY:
              System.out.println("Tuesday");
              break;

       case FRIDAY:
              System.out.println("Tuesday");
```

```
            break;

        case SATURDAY:
            System.out.println("Tuesday");
            break;

        default:
            System.out.println("other  day");
        }
    }
}
```

Output:

Monday

Example: An example of an enum with method overriding.

```
public enum MediaFancy {

    /** The enum constant for a book, with a method override */
    BOOK {
        public String toString() { return "Book"; }
    },

    /** The enum constant for a Music CD */
    MUSIC_CD,
    /** ... */
    MUSIC_VINYL,
    MOVIE_VHS,
    MOVIE_DVD;
```

```
    public  static void main(String[] args) {

        /** It is generally disparaged to have a main() in an enum;
        MediaFancy[] data = { BOOK, MOVIE_DVD,
                                        MUSIC_VINYL };

        for (MediaFancy mf : data) {
                System.out.println(mf);
        }
    }
}
```

Output:

```
Book
MOVIE_DVD
MUSIC_VINYL
```

Enforcing the Singleton Pattern

Problem

- You want to be sure there is only one instance of your class in a given Java Virtual Machine.

Solution

- Make your class enforce the Singleton Pattern

Singleton Pattern

- The Singleton's purpose is to control object creation, limiting the number of objects to only one.

- Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields.

- Singletons often control access to resources, such as database connections or sockets.

- For example, if you have a license for only one connection for your database or your JDBC driver has trouble with multithreading, the Singleton makes sure that only one connection is made or that only one thread can access the connection at a time.

Example:

```
public class Singleton {

    private static Singleton singleton = new Singleton( );

      /* A private Constructor prevents any other
       * class from instantiating.
       */
      private Singleton() { }

      //Static 'instance (i.e., getInstance( ))' method
      //which must be public) then simply returns this instance
      public static Singleton getInstance( ) {
        return singleton;
      }

      /* Other methods protected by singleton-ness */
      protected static void demoMethod( ) {
        System.out.println("demoMethod for singleton");
      }
}
```

```
public class SingletonDemo {

    public static void main(String[] args) {

        // create a singleton object
        Singleton tmp = Singleton.getInstance( );
        tmp.demoMethod( );
    }
}
```

Output:

demoMethod for singleton

Roll Your Own Exceptions

- You can create your own exceptions in Java.
- need to extend the predefined **Exception** class to create your own Exception, which are considered to be checked exceptions.
- When subclassing either of these, it is customary to provide at least these constructors:
    - A no-argument constructor
    - A one-string argument constructor
    - A two argument constructor—a string message and a Throwable "cause"

.

Example: Exception using in banking problem

- In this example we have used 3 classes:

    - **InsufficientFundsException Class:**
        - The InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.
    - CheckingAccount **Class:**
        - The CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.
    - BankDemo **Class:**
        - The BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

```
import java.io.*;

public class InsufficientFundsException extends Exception {

        private double amount;

        public InsufficientFundsException(double amount) {
                this.amount = amount;
        }

        public double getAmount() {
                return amount;
        }
}
```

```
import java.io.*;

public class CheckingAccount {

	private double balance;
	private int number;

	public CheckingAccount(int number) {
		this.number = number;
	}

	public void deposit(double amount) {
		balance += amount;
	}

	public void withdraw(double amount) throws
					InsufficientFundsException {
		if(amount <= balance) {
			balance -= amount;
		}
		else {
			double needs = amount - balance;
			throw new InsufficientFundsException(needs);
		}
	}

	public double getBalance() {
		return balance;
	}

	public int getNumber() {
		return number;
	}
}
```

```
public class BankDemo {

    public static void main(String [] args) {

        CheckingAccount c = new CheckingAccount(101);
        System.out.println("Depositing $500...");
        c.deposit(500.00);
        System.out.println("The Curreent Balance: " +
                                                    c.getBalance());

        try {
            System.out.println("\nWithdrawing $100...");
            c.withdraw(100.00);
            System.out.println("The Curreent Balance: " +
                                                    c.getBalance());

            System.out.println("\nWithdrawing $600...");
            c.withdraw(600.00);
        }
        catch (InsufficientFundsException e) {

            System.out.println("Sorry, but you are short $" +
                                                    e.getAmount());
            e.printStackTrace();
        }
    }
}
```

Output:

```
Depositing $500...
The Curreent Balance: 500.0

Withdrawing $100...
The Curreent Balance: 400.0
```

```
Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
        at CheckingAccount.withdraw(CheckingAccount.java:22)
        at BankDemo.main(BankDemo.java:16)
```