OOP Assignment II

1. What is an Object class? With appropriate syntax, explain the use of `final` keyword for different purposes.

Java defines a special **Object Class**, which is the default parent class of all the other classes. In other words, every other class inherits from the **Object Class**. The methods defined by the object class are thus inherited by every other class.

Few methods of the object class include:

```
public String toString()
public boolean equals(Object obj)
protected Object clone() throws CloneNotSupportedException
protected void finalize() throws Throwable
```

The `final` keyword in java is used to restrict the user. It is used in many contexts :

- Variable
- Method
- Class

Final Variable

If you make any variable **final**, you cannot change it's value later. i.e. A constant. A final variable that is not initialized during declaration is to be initialized only inside the constructor, not inside an method.

e.g. : `final int SIZE = 1000;`

Final Method

By making a method **final,** we prevent it from being overridden. **Final** methods can still be inherited. A constructor cannot be declared as final, as they are never inherited.

e.g. :
```
public final void finalMethod () { // Do Stuff...
}
```

Final Class

A **final** class cannot be subclassed/extended by any other class.

e.g. :
```
public final class FClass { // Do Stuff...
}
```

2. What is dynamic method dispatch? Explain with a complete example program Java's way to achieve run time polymorphism.

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.

For example:

```
class Demon {
        public void type () {
              System.out.println("Demon.");
        }
}
class Vampire extends Demon {
        public void type () {
              System.out.println("Vampire.");
        }
}
class VampireWithSoul extends Vampire {
        public void type () {
              System.out.println ("Vampire with soul.");
        }
}
class DemonTest {
 public static void main(String[] args) {

   Demon demon = new Demon ();
   Vampire dracula = new Vampire ();
   VampireWithSoul angel = new VampireWithSoul ();

   demon.type ();  // Calls Demon's version of type
   demon = dracula;
   demon.type ();  // Calls Vampire's version of type
   demon = angel;
   demon.type ();  // Calls VampireWithSoul's version of type
 }
}


/**
 *  Output:
 *  Demon.
 *  Vampire.
 *  Vampire with soul.
 */
```

3. Distinguish between the following

a) Method overriding and Method overloading.

METHOD OVERRIDING	METHOD OVERLOADING
Used *to provide the specific implementation* of the method that is already provided by its super class.	Used *to increase the readability* of the program.
Occurs *in two classes* that have inheritance relationship.	Performed *within class*.
Parameters must be same.	*Parameters must be different*.
Run time polymorphism.	*Compile time polymorphism*.
Return type must be same or covariant.	*Return type can be same or different* in method overloading.

b) Interfaces and Abstract classes

INTERFACES	ABSTRACT CLASSES
Can have **only abstract** methods.	Can **have abstract and non-abstract** methods.
Supports multiple inheritance.	**Doesn't support multiple inheritance**.
Only static and final variables.	**Can have final, non-final, static and non-static variables**.
Can't have **static methods, main method or constructor**.	Can have **static methods, main method and constructor**.
Can't **provide the implementation of abstract class**.	Can **provide the implementation of interface**.
The **interface keyword** is used to declare interface.	The **abstract keyword** is used to declare abstract class.

Example:
`public interface Drawable {`
`  void draw ();`
`}`
	Example:
	`public abstract class Shape {`
	`  public abstract void draw ();`
	`}`

4. Explain the use of keyword `super` in all possible ways with respect to inheritance with examples.

The `super` keyword in java is a reference variable that is used to refer immediate parent class object.

An instance of the parent (super) class from it's child is referred by the `super` keyword.

Uses of `super`:

- `super` is used to refer immediate parent class instance variable.
- `super()` is used to invoke immediate parent class constructor.
- `super` is used to invoke immediate parent class method.

Example:

```
class Demon {
     String demonType = "General Demon";

     Demon() {
       System.out.println("[Demon init]");
     }

     void weakness() {
       System.out.println("Arrogance.");
     }
}

public class Vampire extends Demon {
     String demonType = "Vampire";

     Vampire() {
       super ();   // Invoking the super class' constructor
       System.out.println("[Vampire init]");
     }

     void display() {
       System.out.println(this.demonType);
// Prints the current demonType ("Vampire")
       System.out.println(super.demonType);
// Prints the demonType of the super class ("General Demon")
     }

     void weakness() {
       System.out.println("Sunlight, Stakes, Fire.");
       super.weakness(); // invoking the super class' method
     }
```

```
    public static void main (String args[]) {
      Vampire vamp = new Vampire();

      vamp.display();

      vamp.weakness();
    }
}

/**
 *   Output:
 *   [Demon init]
 *   [Vampire init]
 *   Vampire
 *   General Demon
 *   Sunlight, Stakes, Fire.
 *   Arrogance.
 */
```

5. Explain the following with appropriate examples.

a) What is the order of invocation of constructors in multilevel hierarchy?

The order of invocation of constructors in multilevel hierarchy is from top to bottom i.e. the top most parent class' constructor is executed first followed by it's immediate children and so on till the last child's constructor.

For example:

```
class Demon {
    Demon() {
      System.out.println("[Demon init]");
    }
}

class Vampire extends Demon {
    Vampire() {
      System.out.println("[Vampire init]");
    }
}



class VampireWithSoul extends Vampire {
    VampireWithSoul() {
      System.out.println("[VampireWithSoul init]");
    }
}
```

```
public class test {
    public static void main (String args[]) {
      VampireWithSoul spike = new VampireWithSoul();
    }
}

/**
 *  Output:
 *  [Demon init]
 *  [Vampire init]
 *  [VampireWithSoul init]
 */
```

Here on creation of new `VampireWithSoul` object, the first call goes to the `Demon` class, followed by `Vampire` class and finally `VampireWithSoul` class.

b) What is the use of interface reference?

We can declare variables as object references of an interface rather than a class type.

Any instance of any class that implements the declared interface can be referred to by such a variable.

When we call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.

An interface reference variable only has knowledge of the methods declared by its interface declaration.

The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them.

The calling code can dispatch through an interface without having to know anything about the “callee.” This process is similar to using a superclass reference to access a subclass object.

For Example:

```
interface Eatable {
    public void eat ();
    //...
}
```

```
class Pizza implements Eatable {
    public void eat () {
      System.out.println ("Mmmmm... Cheese!");
    }
}

class Nugget implements Eatable {
    public void eat () {
      System.out.println ("Mmmmm... Crunchy!");
    }
}

class Stone implements Eatable {
    public void eat () {
      System.out.println ("Wait, WTF am I eating?");
    }
}

public class Test2 {
    public static void main (String [] args)  {

      Eatable eatable = new Pizza ();  // Interface reference to Pizza.
      eatable.eat ();

      eatable = new Nugget ();  // Now pointing to Nugget Class.
      eatable.eat ();

      eatable = new Stone ();   // Now pointing to Stone Class.
      eatable.eat ();
    }
}
```

Output:

```
Mmmmm... Cheese!
Mmmmm... Crunchy!
Wait, WTF am I eating?
```