**Four fundamental concepts of OOP**

Four fundamental concepts of Object Oriented Programming are:

* Abstraction
* Encapsulation
* Inheritance
* Polymorphism

**Abstraction**

Abstraction is a model that includes the most important, essential quality of something, rather than including specific example or details of it.

It automatically discards the unimportant and irrelevant details.

* Abstraction is an external view of the object
* Abstraction allows us to manage complexity by concentrating on the essential features /characteristics of an entity that distinguishes it from all other kind of entities.
* Abstraction is domain depended and perspective dependent. That is, what is important in one context may not be important in another.
* In the context of OOP, abstraction of an object means it outside view provided by the interface. Note that, the interface only tells us what the object can do; it does not say how it performs it work.

**Example**: suppose, you own a car. Now, you don’t think car as a set of ten thousands (say) of individual parts. Instead you think of it as a well-defined object with its own unique behavior. That allows you to use your own car to go to grocery store without thinking of the complexity of ten thousands of parts of the car. While you are driving your car, you don’t even consider how the engine, transmission and breaking system actually work so that your car moves from a place to another place. Because you skip the complexities of the car by means of abstraction.

**Encapsulation**

“The physical localization of features (e.g. properties, behaviours) into a single black box abstraction that hides their implementation (and associates design decisions) behind a public interface”

* Encapsulation means hiding implementation details from external world.
* **Points to remember about encapsulation**
* It is approach of putting related things together.
* Distinguishes between interface and implementation. Exposes interface, hides implementation.
* Encapsulation is often referred to as “information hiding or data hiding” making it possible for the clients to operate without knowing how the implementation fulfills the interface.
* It is possible to change the implementation without updating the clients as long as the interface is unchanged.

**Inheritance**

Inheritance is the process of involving building classes upon existing classes. So that additional functionality can be added.

* “Inheritance is a way by which a newly defined class inherits attributes and behaviours of an existing class along with its own properties”
* Using inheritance the hierarchical relationships are established. Inheritance allows the reusability of the code of the existing class in derived class of the existing classes.
* But reusability should not be the only focus in inheritance. We must check first, there if there a “is a” relationship between two classes or not.
* 2 wheeler “is an” automobile. Ok. A **“is a”** relationship exists.
* 4 wheeler “is an” automobile. Ok. A **“is a”** relationship exists.
* Steering wheel “is an “automobile. Wrong. No, “is a” relationship exists.

**Inheritance example:**

Suppose, a class exists named “Person”. Now, we want to create another class named “Employee”. Now, an employee is surely a person. (i.e. a “is a “relationship exists between class “Employee” and class “Person”) So, now we have two choices:

1. We can make the “Employee” class from the scratch (i.e. making it from the very beginning)
2. We can reuse the code of class “Person” in the class “Employee” and adds the needed functionality in it.

To do in the second way, we have to use the **“inheritance”** concept.

**Polymorphism**

**Polymorphism** is sharing a common interface for multiple types but having different implementation for different types.

**Different types of Polymorphism (from this, not needed for the 10th standard)**

* **Coercion (Though the details of the different types of polymorphism is not in the syllabus, this coercion section need to be read to understand the implicit conversion or implicit casting between datatypes)**Coercion indicates a single abstraction serving several type through implicit type conversion. Only a finitely many different types can be coerced to a given parameter type. Sometimes, when we call a method m1 with one parameter of type X (say) and the method m2 to accept a parameter of type Y (say). Then the implicit type conversion of X to Y is done and it is a type of coercion polymorphism.

class Coercion4

{

public static double add(double x,double y)

{

return x+y;

}

public static void main(String args[])

{

System.out.println("\u000c");

System.out.println("The addition of two nos(one double and one int):"+add(10.5,5));

System.out.println("The addition of two nos(one double and one char):"+add(10.5,'A'));

System.out.println("The addition of two nos(two integers):"+add(10,5));

System.out.println("The addition of two nos(two characters):"+add('A','C'));

}

}

Now, notice that the function public static void add (Why static? Because we want to use the function add without creating an object of the class **“Coercion4”** ) expects two parameters of double type.

**In first call,** we send a double type parameter and an integer type parameter. The integer type parameter is implicitly converted to double type. Then the addition operation is performed in the function add.

**In second call,** we send a double type parameter and a character type parameter. The character type’s ASCII value is implicitly converted to double then the addition operation is performed in the function.

**In third call,** we send two integer type variables. Both are converted to double type first, then those nos. are added.

**In fourth call,** we send two character type parameters to the function. First their ASCII values are converted to double. Then addition operation is performed upon them.

* Type conversion rules (for implicit conversion)
* short and char datatypes can be converted into integer datatype.
* short, char and int datatypes can be converted into float datatype.
* short, char, int and float datatypes can be converted into double datatype.
* short, char, int, float, double and Boolean can be converted into String datatype.

**See the following examples: First example**

String S1="The value of the integer is:"+5;

String S2="The value of the float is:"+5.0;

String S3="The value of the double is:"+5.10;

String S4="The ASCII value of the "+'A'+" is: "+65;

boolean b=true;

String S5="The value of the boolean is:"+true;

System.out.println(S1);

System.out.println(S2);

System.out.println(S3);

System.out.println(S4);

System.out.println(S5);

The output of the program is:

The value of the integer is: 5

The value of the float is: 5.0

The value of the double is: 5.10

The ASCII value of the A is: 65

The value of the boolean is: true

**This is output box. So, I colored it as green.**

Now, this is all due to implicit type conversion of other types to String.

Note

The coloring is based on the coloring in BlueJ. As you can see, all primitive in build types of java come into the color red. That makes the clear concept that though String is an in-build datatype, it is not primitive.

**Second example**

int X=65+'A';

System.out.println("The value of the integer is:"+X);

Output of the program is 130 (since ASCII value of the ‘A’ is 65. Here, A is implicitly type casted as integer.

However, coercion does not only mean implicit type conversions of in-build types.

Suppose, class” Y” extends class “X”, and class “Other” has a method with signature aMethod(X). (the method’s name is aMethod). For the method invocation in code below, the compiler implicitly converts the subclass reference variable of type “Y” to the base class reference of type “X” as required by the method signature. The implicit conversion allows the aMethod’s implementation to use only the type operator defined by “X”.

* Method overloading

Method overloading is to provide same interface (In a simple language, providing same name) for functions with different **signatures**.

Now, some points on function signature:

* Methods with same signature have same name, same parameter types and same order of the parameters.
* But the signature of a method is not comprised of its return type or its visibility or its thrown exceptions (**visibility: is the method public, protected or private?**)
* So, methods with same name but different signatures have different no. of arguments, or they have different orders of the arguments.
* **And to satisfy the method overloading, methods must exist in the same scope. (in java, they must exist in the same class)**

public void Add(int a,int b)

{

return a+b;

}

public void Add(double a,double b)

{

return a+b;

}

If these two functions exist in the same class, then this is an example of method overloading.

public void logIn(String param, String err)

{

……………………………………………

}

public void logIn(String err, String param)

{

……………………………………………

}

This is not method overloading. It may seem that these two functions differ in the order of arguments, but they actually do not differ in argument order. Consider the function calling, how the compiler will detect which one to pass as param (parameter) and which one to pass as err (error message if parameters do not satisfy the log in condition)?

* **Parametric polymorphism**

Parametric polymorphism indicates a single name working as an abstraction (or interface) operating uniformly across different data types. For example, a list data type can be used as an abstraction to represent a generic list of homogeneous data. The list will be defined in terms of generic parameterized data types, as if we are passing data type as a parameter to the list container.

When the list is instantiated for a parametric data type, the list contains data members each of which are of particular data type. Since the parameterized data type can be any data type, we could virtually support infinite no. of used data types.

* **Inclusion polymorphism**

**Inclusion polymorphism** allows polymorphism through an inclusion relationship between types or sets of values (this is also called as **subtyping** in java). A subclass or derived class is a subtype of the superclass or the base class. When the base class has a method and derived class also has the method (with same signature… i.e. name of methods, argument order of methods, and datatypes of arguments are same of the base class and the derived class) with same signature but different definition (i.e. though two functions have same signature, they differ in operations), that is called inclusion polymorphism or method overriding (common name of this type of polymorphism)

class Animal

{

public void stinky()

{

System.out.println("Stinky animal!");

}

}

class Dog extends Animal

{

public void Stinky()

System.out.println("Stinky dog!");

}

public void Bark()

{

System.out.println("vow vow");

}

}

Now, class **Animal** has a method named “Stinky”. And class **Dog** has also a method named “Stinky”. Now, signatures of both functions are same. But, the derived class redefines the method. So, this is method overriding.