OBJECT-ORIENTED PROGRAMMING (OOP) CONCEPTS WITH EXAMPLES

Object-Oriented Programming (OOP) uses "objects" to model realworld objects. Object-Oriented Programming (OOP) consist of some important concepts namely Encapsulation, Polymorphism, Inheritance and Abstraction. These features are generally referred to as the OOPS concepts. If you are new to object oriented approach for software development,

An object in OOP has some state and behavior. In Java, the state is the set of values of an object's variables at any particular time and the behaviour of an object is implemented as methods. Class can be considered as the blueprint or a template for an object and describes the properties and behavior of that object, but without any actual existence. An object is a particular instance of a class which has actual existence and there can be many objects (or instances) for a class. Static variables and methods are not purely object oriented because they are not specific to instances (objects) but common to all instances.

We will see the OOPS Concepts in a bit more detail.

ENCAPSULATION

Encapsulation is the process of wrapping up of data (properties) and behavior (methods) of an object into a single unit; and the unit here is a Class (or interface). Encapsulate in plain English means *to enclose or be enclosed in or as if in a capsule*. In Java, everything is enclosed within a class or interface, unlike languages such as C and C++ where we can have global variables outside classes.
Encapsulation enables **data hiding,** hiding irrelevant information from the users of a class and exposing only the relevant details required by the user.

- We can expose our operations hiding the details of what is needed to perform that operation.
- We can protect the internal state of an object by hiding its attributes from the outside world (*by making it private)*, and then exposing them through setter and getter methods. Now the modifications to the object internals are only controlled through these methods.
  - Consider the example of a linked list's getsize method. We might be now using a variable named size that is updated on every insert / delete operation. Later we might decide to traverse the list and find size every time someone ask for size. But if some code was directly accessing the size variable, we would have to change all those code for this change. However if we were accessing the size variable through a getsize method, other code can still call that method and we can do our changes in that method.

Setters & Getters

A setter is a method used to change the value of an attribute and a getter is a method used to get the value of an attribute. There is a standard naming convention for getters and setters, but Java compiler won't complain even otherwise.

```
private String name;

public String getName() {

  return name;

}

public void setName(String name) {

  this.name=name;

}
```

Here getName and setName are the getter and setter for the variable 'name' respectively. Since this is a standard naming convention, many IDEs like eclipse will generate it for you in this form.

INHERITANCE

Inheritance describes the parent child relationship between two classes. A class can get some of its characteristics from a parent class and then add more unique features of its own. For example, consider a Vehicle parent class and a child class Car. Vehicle class will have properties and functionalities common for all vehicles. Car will inherit those common properties from the Vehicle class and then add properties which are specific to a car. Vehicle parent class is known as base class or superclass. Car is known as derived class, Child class or subclass.

Inherited fields can be accessed just like other normal fields (even using "this" keyword). Java specification says that "A subclass does not inherit the private members of its parent class.". This means that subclass cannot access private member (using "this" keyword) from a subclass like other members. When you create an object, it will call its super constructor and the super class object is created with all fields including private; however, only inherited fields can be accessed. We can still access the private variables of the parent class using an accessible parent method like a setter or getter.

```
public class Parent {

  private int i;
  public int j;
  Parent(int i, int j) {
    this.i = i;
    this.j = j;
  }
```

```
  public static void main(String[] args) {
    Child c = new Child(5, 10);
    c.printIJ();
  }
  void printIJ() {
    System.out.println(i + " " + j);
  }
}

class Child extends Parent {
  Child(int i, int j) {
    super(i, j);
    // this.i=25;
    this.j = 15;
  }
}
```

I have commented out the line with this.i as that is not valid: i is a private field and hence not inherited and cannot be accessed using "this" keyword. However we can still access it using an accessible method (e.g. printIJ).

In general, Java supports single-parent, multiple-children inheritance and multilevel inheritence (Grandparent-> Parent -> Child) for classes and interfces. Java supports multiple inheritance (multiple parents, single child) only through interfaces. This is done to avoid some confusions and errors such as diamond problem of inheritance.

POLYMORPHISM

The ability to change form is known as polymorphism. Java supports different kinds of polymorphism like oveloading, overriding, parametric etc.

Overloading

The same method name (method overloading) or operator symbol (operator overloading) can be used in different contexts.

In **method overloading**, multiple methods having same name can appear in a class, but with different signature. And based on the number and type of arguments we provide while calling the method, the correct method will be called.
Java doesn't allow operator overloading except that "+" is overloaded for class String. The "+" operator can be used for addition as well as string concatenation.

Overriding (or subtype polymorphism)

We can override an instance method of parent class in the child class. When you refer to a child class object using a Parent reference (e.g. Parent p = new Child()) and invoke a method, the overriden child class method will be invoked. Here, the actual method called will depend on the object at runtime, not the reference type. Overriding is not applicable for static methods or variables (static and non-static). In case of variables (static and non-static) and static methods, when you invoke a method using a reference type variable, the method or variable that belong to the reference type is invoked.

Example: Consider a class Shape with a draw() method. It can have subclasses Circle and Square. An object of Circle or Square can be assigned to a Shape reference type variable at runtime (e.g. Shape s = new Circle();). While executing draw() on the Shape reference, it will draw a Circle or Square based on the actual object assigned to it at runtime.

You need to follow some rules while overriding:

1. The argument list must be same as that of the overridden method. If they don't match, you might be doing an overload rather than override.
2. The return type must be the same as, or a subtype of the return type declared in the parent class method (also called covariant return type). In case of primitives, return types should be same; even int and short, or double and float, will give compilation error that: "The return type is incompatible with...".
3. Instance methods can be overridden only if they are inherited by the subclass. Private methods are not inheritted and hence cannot be overriden. Protected methods can be overriden by a subclass within the same package.
4. Final classes cannot be inheritted and final methods cannot be overriden.
5. The overriding method can throw any new unchecked (runtime) exception.
6. The overriding method must not throw checked exceptions that are new or broader than those declared by the overridden method.
7. You cannot make the access modifier more restrictive, but you can make less restrictive. Public cannot be made default, but a default can be made public.

   In java 5 and above, we can confirm whether we are doing a valid override by using the @Override annotation above the subclass's method. Compiler will throw error when @Override is applied on static methods, static variables or instance variables.

Parametric polymorphism through generics

Within a class declaration, a field name can associate with different types and a method name can associate with different parameter and return types. Java supports parametric polymorphism via generics.

An example is a list which can accept the type of data it contains through generics.

```
List<String> list = new ArrayList<String>();
```

ABSTRACTION

In plain English, abstract is a concept or idea not associated with any specific instance and does not have a concrete existence. Abstraction in Object Oriented Programming refers to the ability to make a class abstract. Abstraction captures only those details about an object that are relevant to the current perspective, so that the programmer can focus on a few concepts at a time.

Java provides interfaces and abstract classes for describing abstract types. An **interface** is a contract or specification without any implementation. An interface can't have behavior or state. An **abstract class** is a class that cannot be instantiated, but has all the properties of a class including constructors. Constructor of an abstract class is invoked by a subclass from its constructor using super keyword (e.g. super()). Abstract classes can have state and can be used to provide a skeletal implementation.

Source : http://javajee.com/object-oriented-programming-oop-concepts-with-examples