OOPS Principles:

NOTE: Remember that multiple inheritance is not allowed in java. Means that a child class cannot have more than one parent.

Encapsulation: is the binding of data(variables) and code(methods).

Example is capsule and class.

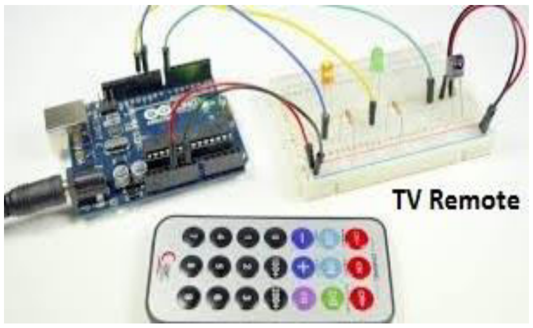
Example: In a company, there are various sections, say finance and sales section- both contains their own data without affecting others.

Advantage arises in security. It can be achieved by creating private variables and creating getter and setter methods.

Abstraction: Advantage also arises in security. ATM machine: in which only service is visible and not the implementation. Also known as data hiding.

When we go to certain websites, then we don't know how the information has been fetched.

NOTE: ABSTRACTION IS ABOUT EXPRESSING EXTERNAL SIMPLICITY AND ENCAPSULATION IS ABOUT HIDING INTERNAL COMPLEXITY.



Example: TV remote, everything is hidden for simplicity means that it is an abstraction. But if connections are seen then encapsulation will be there (upper image). Hiding the details of working is encapsulated.

Interface is the source of pure abstraction while abstract class of partial Abstraction.

Interfaces: are like a set of contracts between client and service provider.

Interfaces does not contain constructors, as they have public static and final variables so no need to instantiate them. But abstract class have constructors because it may contain instance variables to instantiate.

Inheritance increases the code readability and reusability of code.

NOTE: Classes will not occupy memory at the runtime. Just like a datatype defined by the user.

NOTE: We cannot override static methods in Java, as they are associated with the class. Method hiding will then occur.

NOTE: If the parent class throws checked exceptions, then the child class must throw that exception or its subtype, but no need when the parent class throws unchecked exceptions as both are independent classes then.

Marker Interfaces:

The interfaces which do not contain any variables and functions are called marker interfaces. This is also called an empty interface, as they are empty, and do not contain anything.

The three important marker interfaces in the Java language are Cloneable, Serializable, and Remote interface.

(a) Cloneable: Present in java.lang package. For cloning any class object that class must implement the cloneable interface. If not, then CloneNotSupportedException is thrown. By convention, classes that implement this interface should override Object. clone() method.

(b) Serializable: Serializable interface is present in java.io package. The class which must save the present state of an object of that class to the file, must implement the serializable interface.

(c) Remote: Remote interface is present in java.rmi package. A remote object is an object which is stored in one machine and accessed from another machine. So, to make an object a remote object, we need to flag it with a Remote interface. Here, the Remote interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine.

Ques: Why are Marker interfaces created or used?

Ans: They are just like metadata for the compiler. Providing more information about what a class does.

Co-variant method overriding:

It helps to remove the client-side typecasting by enabling you to return a subtype of the overridden method's actual return type.

It means that it is (after Java 5) not necessary to have the same return type for the overriding and the overridden function. The return type of the overriding function must be the subtype of the return type of overridden function. Before Java 5 it was not allowed to override any function if the return type is changed in the child class.

It helps to eliminate the burden of typecasting from the programmer. This method is often used when an object is returned by the overriding method.

```java
// Covariant Method Overriding of Java
import java.util.ArrayList;
// Student class
class Student implements Cloneable {
    int rollNo;
    String className;
    String name;

    // Getters and setters
    public int getRollNo() { return rollNo; }
    public void setRollNo(int rollNo)
    {
        this.rollNo = rollNo;
    }
    public String getClassName() { return className; }
    public void setClassName(String className)
    {
        this.className = className;
    }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    // Class constructor
    public Student(int rollNo, String className,
                   String name)
    {
        this.rollNo = rollNo;
        this.className = className;
        this.name = name;
    }

    // Print method
    public void printData()
    {
        System.out.println("Name : " + name
                           + ", RollNo: " + rollNo
                           + ", Class Name : " + className);
    }

    // Override the clone method
    @Override
    public Student clone() throws CloneNotSupportedException
    {
        return (Student) super.clone(); // Here covariant method overriding
is used. Here the child class method returns the subclass object type.
    }
}

// Driver code
public class GFG {
    public static void main(String arg[])
        throws CloneNotSupportedException
    {
        // new student object created
        Student student1 = new Student(1, "MCA", "Kapil");
        student1.printData();

        // Student object created using clone method
        Student student2 = student1.clone();
        student2.setName("Sachin");
        student2.setRollNo(2);
        student2.printData();
    }
}
```

Constructor Chaining:

The sequence of invoking constructors of the same class upon initializing an object. This means that a constructor will call another constructor of the same class and then that constructor will call the next one.

NOTE: Destructor cannot be overloaded.