Object-Oriented Programming Concepts

Encapsulation

Encapsulation is the practice of wrapping the data and the methods acting on it within the same unit, e.g. class. This allows variables in the local scope to be hidden from other classes and only accessible through their methods. Encapsulation is achieved by declaring private variables within the class and providing public methods to get and set the data.

The advantages of encapsulation are:

- Ability to make fields read-only or write-only.
- Control over what is stored in the variables of a class.
- The users of a class don't know how it stores its data.

Example:

```
1.  public class Encapsulate {
2.    private int total;
3.
4.    public void setTotal(int newTotal) {
5.      total = newTotal;
6.    }
7.
8.    public int getTotal() {
9.      return total;
10.   }
```

Inheritance

Inheritance is the process through which a class gets the attributes and methods of another. It is mainly used for code reusability, avoiding redundancy and enabling programmers to reduce the amount of code by making use of what is already written. Inheritance also allows for data flow between parents and children.

Example:

```
1.  1.  public class Inheritance extends Encapsulate {
2.  2.    private String fistName;
3.  3.    public String lastName;
4.  4.
5.  5.    public void getFullName() {
6.  6.      //some code
7.  7.    }
8.  8.  }
```

Polymorphism

Polymorphism refers to a language's ability to interpret the same word or symbol correctly in different situations based on the context. There are two types of polymorphism in Java:

- **Static Polymorphism** (aka Method Overloading): ability to execute different method implementations by altering the argument/s passed to it.
- **Dynamic Polymorphism** (aka Method Overriding): ability of a subclass to re-declare the methods it inherits from the superclass as appropriate.

Examples:

```
1.  public class StaticPolymorphism {
2.    void sum(int a,int b){System.out.println(a+b);}
3.    void sum(int a,int b,int c){System.out.println(a+b+c);}
4.
5.    public static void main(String args[]) {
6.      Calculation obj=new Calculation();
7.      obj.sum(10,10,10);  // 30
8.      obj.sum(20,20);     //40
9.    }
10. }
```

```
1.  public class SuperAnimal {
2.     public void move(){
3.        System.out.println("Animals can move");
4.     }
5.  }
6.
7.  public class SubAnimal extends SuperAnimal {
8.
9.     public void move() {
10.       System.out.println("SubAnimal can walk and run");
11.    }
12. }
13.
14. public class TestAnimal {
15.
16.    public static void main(String args[]) {
17.       Animal a = new SuperAnimal(); // Animal reference and object
18.       Animal b = new SubAnimal(); // Animal reference but SubAnimal object
19.
20.       a.move();//output: Animals can move
21.
22.       b.move();//output: SubAnimal can walk and run
23.    }
24. }
```

Abstraction

Abstraction is the process of hiding the implementation from the user so that only functionality will be seen. In Java, this is achieved using Abstract classes and methods. An abstract class is incomplete and cannot be instantiated, and if you want to use it you must make it concrete by extending it and implementing all abstract methods that the abstract class has implemented or extended.

Example:

```
1.  public abstract class Dinosaur {
2.    public abstract void eat();
3.    public abstract void move();
4.  }
5.
6.  public class TRex extends Dinosaur {
7.    public void eat() {
8.      System.out.println("The T-Rex eats meat");
9.    }
10.   public void move() {
11.     System.out.println("The T-Rex is fast");
12.   }
13. }
14.
15. public class Brachiosaurus extends Dinosaur {
16.   public void eat() {
17.     System.out.println("The Brachiosaurus eats plants");
18.   }
19.   public void move() {
20.     System.out.println("The Brachiosaurus moves slowly");
21.   }
22. }
```