STUPID Software Design:
STUPID is an acronym (and not so much of an acronym) that describes flawed practices in Object-Oriented design.

- Singleton: Avoid using static/global entities in your software; it'll make it a b*th to test and debug! It also helps with managing low coupling in your code.
- Tight Coupling: Making a change to a part in your application should not affect another part of your application.
- Untestability: Testing should not be hard!
- Premature Optimization: Avoid obsessing about optimization especially at the cost of writing convoluted code.
- Indescriptive Naming: Avoid abbreviating your names, and be descriptive with the names, especially the finer it goes (e.g. module-level → class-level →function level)
- Duplication: If you're finding yourself copying and pasting code, that is a direct sign that you can do way better than that.

SOLID Software Design:
SOLID is an acronym that describes good software practices around Object-Oriented Programming:

- Single Responsibility Principle: A class should have one and only one reason to change.
- Open/Closed Principle: Open for extension, closed for modification
- Liskov Substitution Principle: Your code should not break if you were to replace all objects of type A with its subtype, B.
- Interface Segregation Principle: Split larger interfaces with smaller ones so the implementing classes are concerned with the methods that are of interest to them.
- Dependency Inversion Principle: Instead of a high-level module depending on low-level modules, both should depend on abstractions.

Multiple inheritance is not supported by Java using classes , handling the complexity that causes due to multiple inheritance is very complex. It creates problem during various operations like casting, constructor chaining etc and the above all reason is that there are very few scenarios on which we actually need multiple inheritance, so better to omit it for keeping the things simple and straightforward.

- An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

- An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon).

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

Abstract class vs Interface

1. **Type of methods:** Interface can have only abstract methods. Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.
2. **Final Variables:** Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
3. **Type of variables:** Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.
4. **Implementation:** Abstract class can provide the implementation of interface. Interface can't provide the implementation of abstract class.
5. **Inheritance vs Abstraction:** A Java interface can be implemented using keyword "implements" and abstract class can be extended using keyword "extends".
6. **Multiple implementation:** An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
7. **Accessibility of Data Members:** Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

The **enumerated**, or **enum**, data type in Java is used to describe a specific set of values for a variable. They are **static** or **final** and **type-safe**. They can not

change or be modified. Enum types are **reference type**, meaning they act like a class and can have their own constructors and methods. With enums, there is no risk of bad data getting into the values because they are predefined and unchanging.

You can declare some or all of a class's methods *final*. You use the `final` keyword in a method declaration to indicate that the method cannot be overridden by subclasses. The `Object` class does this—a number of its methods are `final`.

A `final` class is simply a class that can't be extended.

```
1. class A{
2. A get(){return this;}
3. }
4.
5. class B1 extends A{
6. B1 get(){return this;}
7. void message(){System.out.println("welcome to covariant return type");}
8.
9. public static void main(String args[]){
10. new B1().get().message();
11. }
```

```
12. }
```

Covariant return, means that when one overrides a method, the return type of the overriding method is allowed to be a subtype of the overridden method's return type.

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

```
import java.util.ArrayList;


public class Main {

  public static void main(String[] args) {

    ArrayList<Integer> numbers = new ArrayList<Integer>();

    numbers.add(5);

    numbers.add(9);

    numbers.add(8);

    numbers.add(1);

    numbers.forEach( (n) -> { System.out.println(n); } );

  }

}
```

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. ***Runnable***, ***ActionListener***, ***Comparable*** are some of the examples of functional interfaces.

Functional Programming based on different concepts is

1. High Order Functions (HOF).

2. Pure functions.

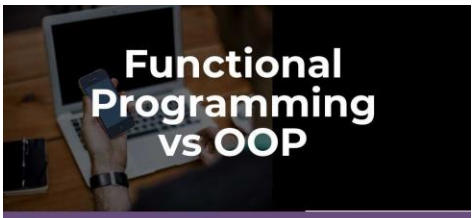
3. Recursion.

4. Strict and Non-strict Evaluation.

5. Type systems.

6. Referential Transparency.

It is a declarative style of programming rather than imperative. The basic objective of this style of programming is to make code more concise, less complex, more predictable, and easier to test compared to the legacy style of coding. Functional programming deals with certain key concepts such as pure function, immutable state, assignment-less programming etc.



#1. Definition

Functional Programming	OOP
Functional programming emphasizes on evaluation of functions.	Object oriented programming based on concept of objects.

#2. Data

Functional Programming	OOP
Functional programming uses immutable data.	Object oriented uses the mutable data.

#3. Model

Functional Programming	OOP
Functional programming does follow declarative programming model.	Object oriented programming does follow imperative programming model.

#4. Support

Functional Programming	OOP
Parallel programming supported by Functional Programming.	Object oriented programming does not support parallel programming.

#5. Execution

Functional Programming	OOP
In Functional programming, the statements can be executed in any order.	In OOPs, the statements should be executed in particular order.

#6. Iteration

Functional Programming	OOP
In Functional programming, recursion is used for iterative data.	In OOPs, loops are used for iterative data.

#7. Element

Functional Programming	OOP
The basic elements of functional programming are Variables and Functions.	The basic elements of object oriented programming are objects and methods.

#8. Use

Functional Programming	OOP
The functional programming is used only when there are few things with more operations.	Object oriented programming is used when there are many things with few operations.

www.educba.com