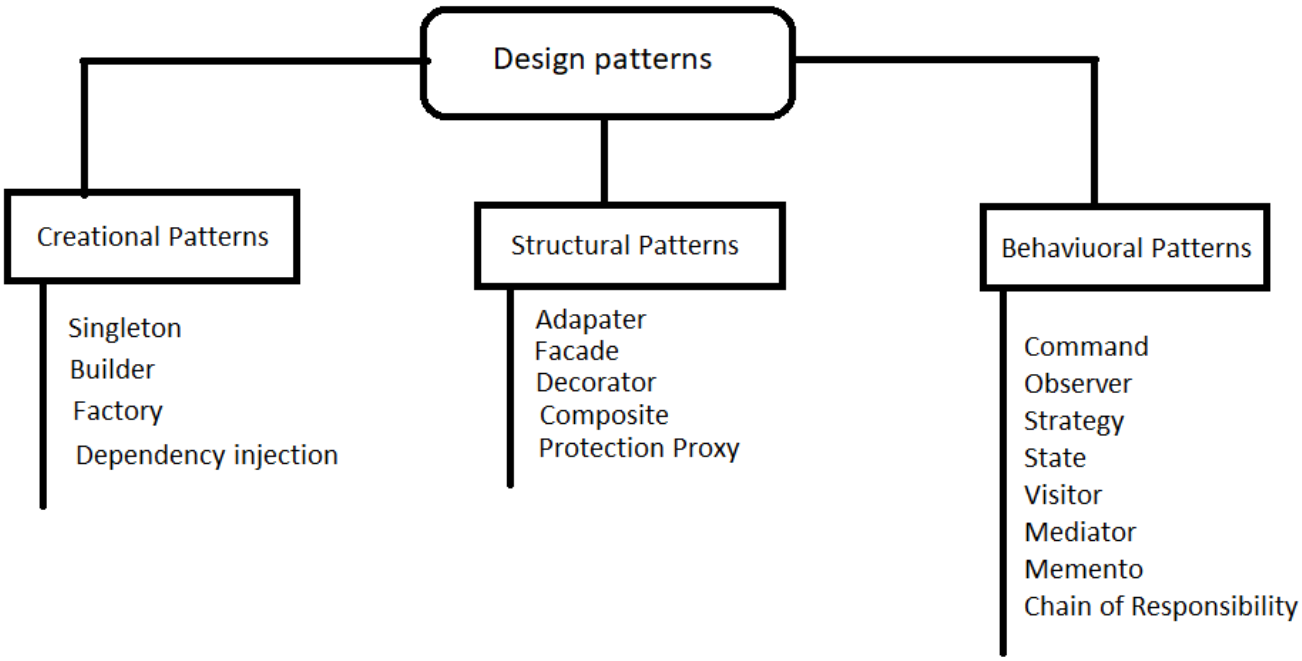
Design Patterns

- **Introduction**
- **Creational pattern**
- **Structural Pattern**
- **Behavioral Pattern**

Introduction

1. Design are blueprint for the solution in programming
2. There are 3 design patterns
   a. Creational pattern
   b. Structural pattern
   c. Behavioral pattern



➢ Singleton

1. Single ton is creational design pattern which restrict instantiation of class to only one object
2. Singleton objects are used in costly resources like database only one instance of the database should be created throughout the app
3. In singleton we can create classic singleton, thread safe singleton, eager singleton.

```kotlin
object KotlinSingleTon {

}
```

Kotlin Singleton

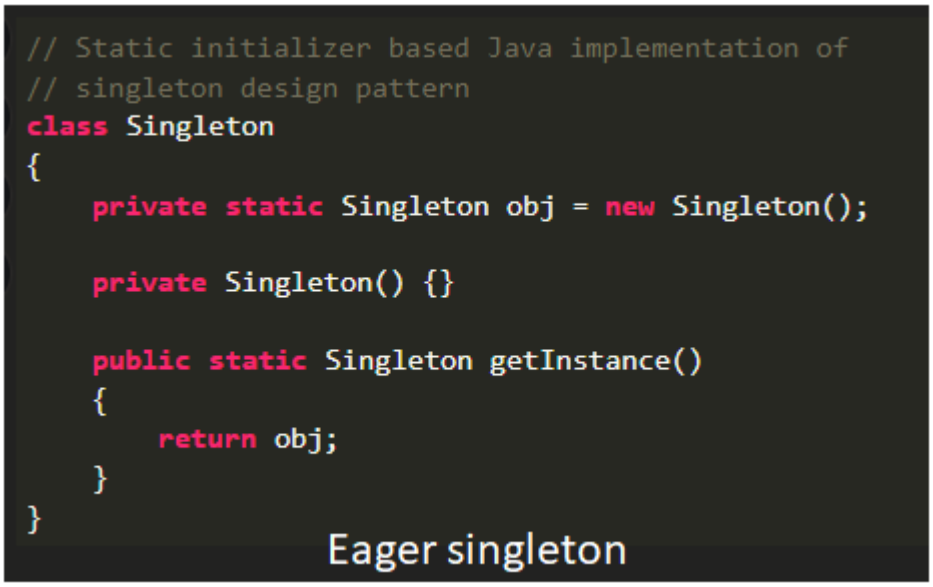
4. Classic singleton are not thread safe if we start two threads at same time different objects may get created for singleton
5. Using CountDownLatch we can demonstrate classic singleton are not safe

```java
/** Classic singleton are not thread safe*/
public class ClassicSingleTon {
    private static ClassicSingleTon obj =null;

    private ClassicSingleTon(){}

    public static ClassicSingleTon getInstance() {
        if(obj == null){
            obj = new ClassicSingleTon();
        }
        return obj;
    }
    public static void destroyObject(){
        obj = null;
    }
}
```

Classic singleton

6. Thread safe singleton

7. Eager singleton will create object in static initializer, these are thread safe as JVM creates the objects



1. Builder Pattern

2. Builder pattern is used to create complex objects step by step and the final step will return the object of product class
3. Advantage of using builder pattern are
   a. readability
   b. reduces parameters in the constructors
   c. Objects will always have instantiated in complete state
4. Disadvantage of using builder pattern is
   a. More number of lines while building objects
   b. Need separate concrete builder class for each product
5. In java builder class will be static and in kotlin builder class can be companion object.
6. **Java Example**

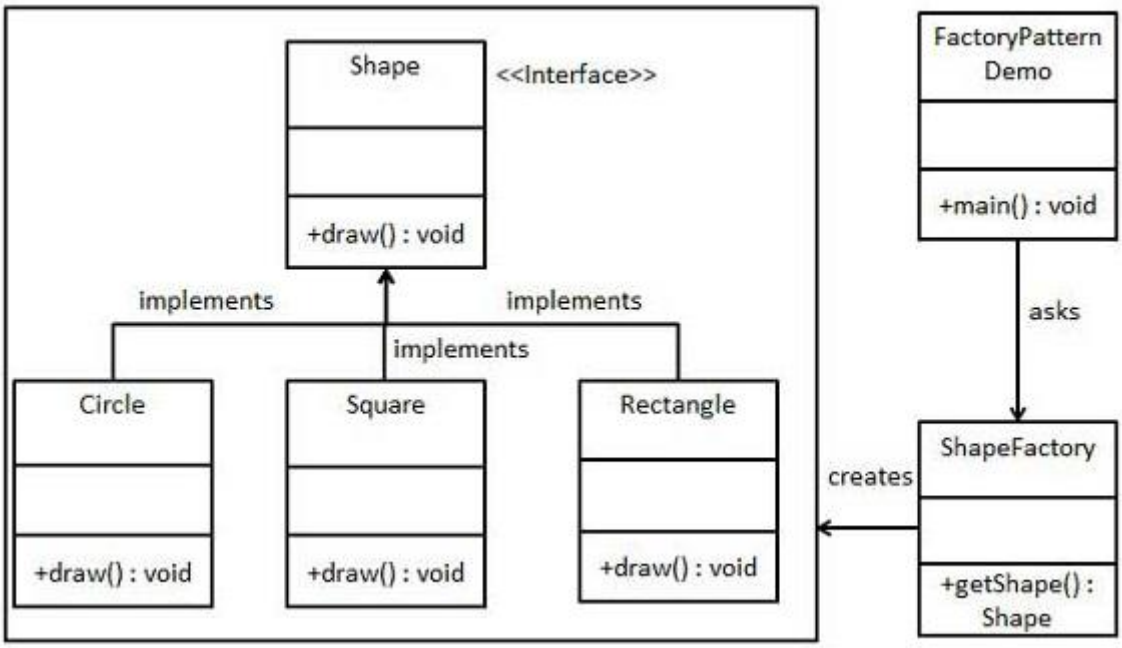
```
public class JavaCarBuilder {
    private int carModel;
    private String brand;
    private String color;

    JavaCarBuilder(Builder builder){
        this.carModel = builder.carModel;
        this.brand = builder.brand;
        this.color = builder.color;
    }
    public static class Builder{
        private int carModel;
        private String brand;
        private String color;
        public Builder(){}

        public Builder setCarModel(int carModel){
            this.carModel = carModel;
            return this;
        }

        public Builder setBrand(String brand){
            this.brand = brand;
            return this;
        }

        public Builder setColor(String color){
            this.color = color;
            return this;
        }

        public JavaCarBuilder build(){
            return new JavaCarBuilder(this);
        }
    }

    @Override
    public String toString() {
        return "JavaCarBuilder{" +
                "Hashcode='"+this.hashCode() + '\''+
                ", carModel='" + carModel+ '\'' +
                ", brand='" + brand + '\'' +
                ", color='" + color + '\'' +
                '}';
    }
}
```

7. Kotlin Example

```
class KotlinCarBuilder(private var carModel: Int, private  var carBrand: String,
private var carColor: String) {

    /**Using kotlin nested class*/
    class Builder {
        private var carModel = 0
        private var carBrand = ""
        private var carColor = ""
        public fun setCarModel(carModel: Int): Builder {
            this.carModel = carModel
            return this
        }

        public fun setCarBrand(carBrand: String): Builder {
            this.carBrand = carBrand
            return this
        }

        public fun setCarColor(color: String): Builder {
            this.carColor = color
            return this
        }

        public fun build(): KotlinCarBuilder {
            return KotlinCarBuilder(this.carModel, this.carBrand, this.carColor)
        }
    }
    /**Using Kotlin companion object*/
    companion object Builder2 {
        private var carModel = 0
        private var carBrand = ""
        private var carColor = ""
        public fun setCarModel(carModel: Int): Builder2 {
            this.carModel = carModel
            return this
        }

        public fun setCarBrand(carBrand: String): Builder2 {
            this.carBrand = carBrand
            return this
        }

        public fun setCarColor(color: String): Builder2 {
            this.carColor = color
            return this
        }

        public fun build(): KotlinCarBuilder {
            return KotlinCarBuilder(this.carModel, this.carBrand, this.carColor)
        }
    }

    override fun toString(): String {
        super.toString()
        return "KotlinCarBuilder : {Hashcode = '${this.hashCode()}', " +
                "CarModel = '$carModel', " +
                "CarBrand = '$carBrand', " +
                "CarColor = '$carColor'}"
    }
}
```

➢ Factory Pattern

1. Factory pattern is object creational design pattern where object creation is not exposed to user
2. Factory class handles all the object creation and give the object back to user
3. Create a vendor class which ask Factory to create object of need
4. Create a Factory class which is responsible of creating object based on vendor needs and return the product
5. Create an interface which hides Product creation



6. **Example**

```
interface Shape {
    fun draw() : String
}
```

```
class Circle : Shape {
    override fun draw() : String{
        return "Circle Drawing"
    }
}
```

```
class Rectangle : Shape{
    override fun draw() : String{
        return "Rectangle Drawing"
    }
}
```

```
class ShapeFactory() {
    public fun getShape(shape : String) : Shape? {
        when(shape){
            "RECTANGLE" -> { return Rectangle() }
            "CIRCLE" -> { return Circle() }
        }
        return null
    }
}
```

```
val shapeFactory = ShapeFactory()
val shape1 : Shape? = shapeFactory.getShape( shape: "CIRCLE")
val shape2 : Shape? = shapeFactory.getShape( shape: "RECTANGLE")
Log.d(FACTORY_PATTERN,  msg: "${shape1?.draw()}")
Log.d(FACTORY_PATTERN,  msg: "${shape2?.draw()}")
```