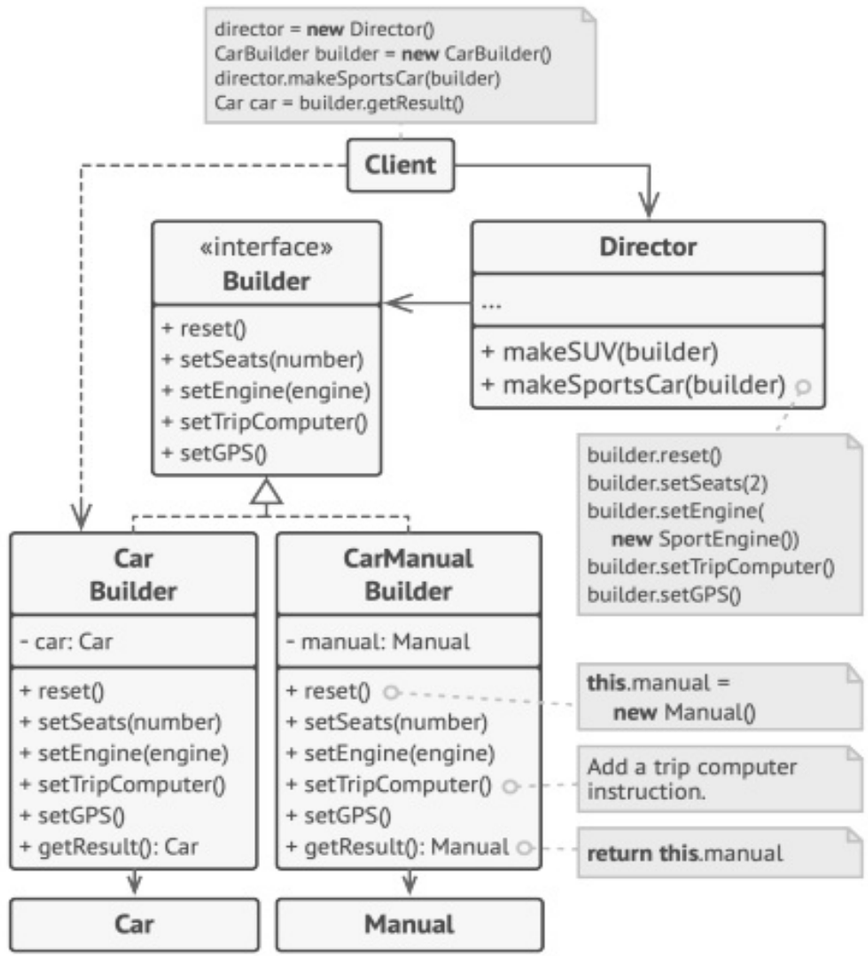
This example of the **Builder** pattern illustrates how you can reuse the same object construction code when building different types of products, such as cars, and create the corresponding manuals for them.



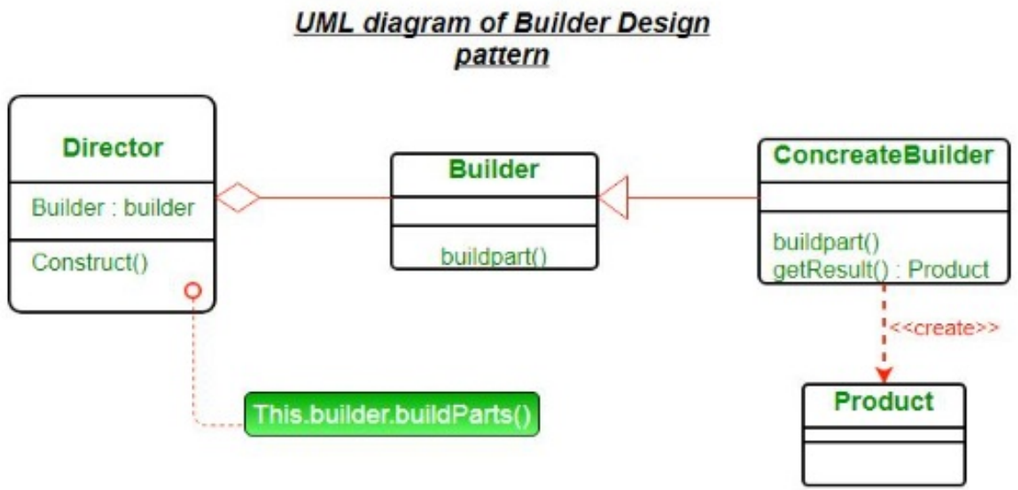
The example of step-by-step construction of cars and the user guides that fit those car models.

A car is a complex object that can be constructed in a hundred different ways. Instead of bloating the `Car` class with a huge constructor, we extracted the car assembly code into a separate car builder class. This class has a set of methods for configuring various parts of a car.

If the client code needs to assemble a special, fine-tuned model of a car, it can work with the builder directly. On the other hand, the client can delegate the assembly to the director class, which knows how to use a builder to construct several of the most popular models of cars.

Client can work on its own builder if has special requirement else use director

Builder pattern aims to "Separate the construction of a complex object from its representation so that the same construction process can create different representations." It is used to construct a complex object step by step and the final step will return the object. The process of constructing an object should be generic so that it can be used to create different representations of the same object.



Here in this Example we have used director

- **Product –** The product class defines the type of the complex object that is to be generated by the builder pattern.
- **Builder –** This abstract base class defines all of the steps that must be taken in order to correctly create a product. Each step is generally abstract as the actual functionality of the builder is carried out in the concrete subclasses. The GetProduct method is used to return the final product. The builder class is often replaced with a simple interface.
- **ConcreteBuilder –** There may be any number of concrete builder classes inheriting from Builder. These classes contain the functionality to create a particular complex product.
- **Director –** The director-class controls the algorithm that generates the final product object. A director object is instantiated and its Construct method is called. The method includes a parameter to capture the specific concrete builder object that is to be used to generate the product. The director then calls methods of the concrete builder in the correct order to generate the product object. On completion of the process, the GetProduct method of the builder object can be used to return the product.

```java
interface HousePlan
{
    public void setBasement(String basement);

    public void setStructure(String structure);

    public void setRoof(String roof);

    public void setInterior(String interior);
}
```

Why Example

```java
class House implements HousePlan
{

    private String basement;
    private String structure;
    private String roof;
    private String interior;

    public void setBasement(String basement)
    {
        this.basement = basement;
    }

    public void setStructure(String structure)
    {
        this.structure = structure;
    }

    public void setRoof(String roof)
    {
        this.roof = roof;
    }

    public void setInterior(String interior)
    {
        this.interior = interior;
    }

}
```

```java
interface HouseBuilder
{

    public void buildBasement();

    public void buildStructure();

    public void buildRoof();

    public void buildInterior();

    public House getHouse();
}
```

```java
class IglooHouseBuilder implements HouseBuilder
{
    private House house;

    public IglooHouseBuilder()
    {
        this.house = new House();
    }

    public void buildBasement()
    {
        house.setBasement("Ice Bars");
    }

    public void buildStructure()
    {
        house.setStructure("Ice Blocks");
    }

    public void buildInterior()
    {
        house.setInterior("Ice Carvings");
    }

    public void buildRoof()
    {
        house.setRoof("Ice Dome");
    }
    public House getHouse()
    {
        return this.house;
    }
}
```

```java
class TipiHouseBuilder implements HouseBuilder
{
    private House house;

    public TipiHouseBuilder()
    {
        this.house = new House();
    }

    public void buildBasement()
    {
        house.setBasement("Wooden Poles");
    }

    public void buildStructure()
    {
        house.setStructure("Wood and Ice");
    }

    public void buildInterior()
    {
        house.setInterior("Fire Wood");
    }

    public void buildRoof()
    {
        house.setRoof("Wood, caribou and seal skins");
    }

    public House getHouse()
    {
        return this.house;
    }

}
```

Director here

```java
class CivilEngineer
{

    private HouseBuilder houseBuilder;

    public CivilEngineer(HouseBuilder houseBuilder)
    {
        this.houseBuilder = houseBuilder;
    }

    public House getHouse()
    {
        return this.houseBuilder.getHouse();
    }

    public void constructHouse()
    {
        this.houseBuilder.buildBasement();
        this.houseBuilder.buildStructure();
        this.houseBuilder.buildRoof();
        this.houseBuilder.buildInterior();
    }
}
```

Just see it

```java
class Builder
{
    public static void main(String[] args)
    {
        HouseBuilder iglooBuilder = new IglooHouseBuilder();
        CivilEngineer engineer = new CivilEngineer(iglooBuilder);

        engineer.constructHouse();

        House house = engineer.getHouse();

        System.out.println("Builder constructed: "+ house);
    }
}
```

Gfg Example lets modify it a bit

use of gfg example is very common so in interview first use it

another e.g is in MacD we go to eat food either eat what they offer as a combo or make your combo what you want to eat

Advantages of Builder Design Pattern

- The parameters to the constructor are reduced and are provided in highly readable method calls.
- Builder design pattern also helps in minimizing the number of parameters in the constructor and thus there is no need to pass in null for optional parameters to the constructor.
- Object is always instantiated in a complete state
- Immutable objects can be built without much complex logic in the object building process.

Disadvantages of Builder Design Pattern

- The number of lines of code increases at least to double in builder pattern, but the effort pays off in terms of design flexibility and much more readable code.
- Requires creating a separate ConcreteBuilder for each different type of Product.

```java
public interface HousePlan
{
    public void setBasement(String basement);

    public void setStructure(String structure);

    public void setRoof(String roof);

    public void setInterior(String interior);
}
```

Here See why we taken this in set method of Builder class

```java
class House implements HousePlan
{

    private String basement;
    private String structure;
    private String roof;
    private String interior;

    public void setBasement(String basement)
    {
        this.basement = basement;
    }

    public void setStructure(String structure)
    {
        this.structure = structure;
    }
    public void setRoof(String roof)
    {
        this.roof = roof;
    }

    public void setInterior(String interior)
    {
        this.interior = interior;
    }

    @Override
    public String toString() {
        return this.basement+" "+this.interior+" "+this.roof
        +" "+this.structure;
    }

}
```

```java
interface HouseBuilder
{

    public HouseBuilder buildBasement();

    public HouseBuilder buildStructure();

    public HouseBuilder buildRoof();

    public HouseBuilder buildInterior();

    public House getHouse();
}
```

```java
class IglooHouseBuilder implements HouseBuilder
{
    private House house;

    public IglooHouseBuilder()
    {
        this.house = new House();
    }

    public HouseBuilder buildBasement()
    {
        house.setBasement(basement: "Ice Bars");
        return this;
    }

    public HouseBuilder buildStructure()
    {
        house.setStructure(structure: "Ice Blocks");
        return this;
    }

    public HouseBuilder buildInterior()
    {
        house.setInterior(interior: "Ice Carvings");
        return this;
    }
    public HouseBuilder buildRoof()
    {
        house.setRoof(roof: "Ice Dome");
        return this;
    }
    public House getHouse()
    {
        return this.house;
    }

}
```

This is returned

```java
class TipiHouseBuilder implements HouseBuilder
{
    private House house;

    public TipiHouseBuilder()
    {
        this.house = new House();
    }

    public HouseBuilder buildBasement()
    {
        house.setBasement(basement: "Wooden Poles");
        return this;
    }

    public HouseBuilder buildStructure()
    {
        house.setStructure(structure: "Wood and Ice");
        return this;
    }

    public HouseBuilder buildInterior()
    {
        house.setInterior(interior: "Fire Wood");
        return this;
    }

    public HouseBuilder buildRoof()
    {
        house.setRoof(roof: "Wood, caribou and seal skins");
        return this;
    }
    public House getHouse()
    {
        return this.house;
    }

}
```

```java
class CivilEngineer
{

    private HouseBuilder houseBuilder;

    public CivilEngineer(HouseBuilder houseBuilder)
    {
        this.houseBuilder = houseBuilder;
    }

    public House constructHouse()
    {
       return this.houseBuilder.buildBasement()
        .buildStructure()
        .buildRoof().buildInterior().getHouse();
    }
}
```

This was refered to set this structure & Simple code !! ☺

```java
public class Build {
    Run | Debug
    public static void main(String[] args)
    {
        HouseBuilder iglooBuilder = new IglooHouseBuilder();
        CivilEngineer engineer = new CivilEngineer(iglooBuilder);

        House h1=engineer.constructHouse();
         System.out.println("Builder constructed: "+ h1);

        HouseBuilder tb=new TipiHouseBuilder();
        CivilEngineer engineer2 = new CivilEngineer(tb);
        House h2=engineer2.constructHouse();
        System.out.println(h2);
    }

}
```

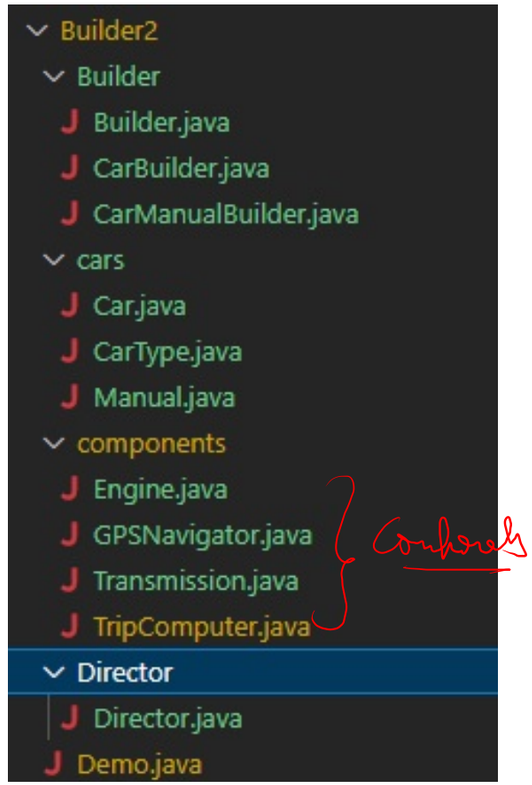
O/P

```
/data/User/workspaceStorage/4ba69438/redhat.java/jdt_ws/workspace_4
Builder constructed: Ice Bars Ice Carvings Ice Dome Ice Blocks
Wooden Poles Fire Wood Wood, caribou and seal skins Wood and Ice
```

Lets see another Example on Builder

Here we are going to build Car & Manual using Director & w/o director

Lets see code on next page



```java
public class Engine {
    private final double volume;
    private double mileage;

    public Engine(double volume, double mileage) {
        this.volume = volume;
        this.mileage = mileage;
    }

    public double getVolume() {
        return volume;
    }

    public double getMileage() {
        return mileage;
    }
}
```

```java
package Patterns.Builder2.components;

public enum Transmission {
    SINGLE_SPEED, MANUAL, AUTOMATIC, SEMI_AUTOMATIC
}
```

```java
public class TripComputer {

    private String s=" gio";

    public String getString() {
        return s;
    }

}
```

```java
public class GPSNavigator {
    private String route;

    public GPSNavigator() {
        this.route = "j_2978,sector-49,sainik colony";
    }

    public GPSNavigator(String manualRoute) {
        this.route = manualRoute;
    }

    public String getRoute() {
        return route;
    }
}
```

See all Contents at one place

```java
public enum CarType {
    CITY_CAR, SPORTS_CAR, SUV
}
```

```java
public class Car {
    //Mandatory
    private final CarType carType;
    private final int seats;
    private final Engine engine;
    private final Transmission transmission;
    //Optional
    private final TripComputer tripComputer;
    private final GPSNavigator gpsNavigator;
    private double fuel ;

    public Car(CarBuilder cb) {
        this.carType = cb.carType;
        this.seats = cb.seats;
        this.engine = cb.engine;
        this.transmission = cb.transmission;
        this.tripComputer = cb.tripComputer;
        this.gpsNavigator = cb.gpsNavigator;
        this.fuel=cb.fuel;
    }

    @Override
    public String toString() {
        StringBuilder sb=new StringBuilder();
        sb.append(this.carType+" "+this.seats+" "+this.engine.getMileage()+" "
        +this.engine.getVolume()+" "+this.transmission+" ");
        if(this.tripComputer!=null) sb.append(this.tripComputer.getString()+" ");
        if(this.gpsNavigator!=null)sb.append(this.gpsNavigator.getRoute()+" ");
        sb.append(this.fuel);
        return sb.toString();
    }
}
```

```java
public class Manual {
    //Mandatory
    private final CarType carType;
    private final int seats;
    private final Engine engine;
    private final Transmission transmission;
    //Optional
    private final TripComputer tripComputer;
    private final GPSNavigator gpsNavigator;
    private double fuel ;

    public Manual(CarManualBuilder cb) {
        this.carType = cb.carType;
        this.seats = cb.seats;
        this.engine = cb.engine;
        this.transmission = cb.transmission;
        this.tripComputer = cb.tripComputer;
        this.gpsNavigator = cb.gpsNavigator;
        this.fuel=cb.fuel;
    }

    @Override
    public String toString() {
        return this.carType+" "+this.seats+" "+this.engine+" "+this.transmission+" "+
        this.tripComputer+" "+this.gpsNavigator+" "+this.fuel;
    }
}
```

```java
public interface Builder {

    Builder setTripComputer(TripComputer tripComputer);
    Builder setGPSNavigator(GPSNavigator gpsNavigator);
    Car buildCar();
    Manual buildManual();

}
```

```java
public class CarBuilder implements Builder {
        //Mandatory
        public CarType carType;
        public int seats;
        public  Engine engine;
        public Transmission transmission;
        //Optional
        public TripComputer tripComputer=null;
        public GPSNavigator gpsNavigator=null;
        public double fuel ;

    public CarBuilder(CarType ct,int seats,Engine eg,Transmission t){
            this.carType=ct;
            this.seats=seats;
            this.engine=eg;
            this.transmission=t;
        }


@Override
public Builder setTripComputer(TripComputer tripComputer) {
    this.tripComputer = tripComputer;
    return this;
}
```

```java

    @Override
    public Builder setGPSNavigator(GPSNavigator gpsNavigator) {
        this.gpsNavigator = gpsNavigator;
        return this;

    }
    @Override
    public Car buildCar() {
        return new Car(this);
    }


    @Override
    public Manual buildManual() {
        return null;
    }
}

```

```java
public class CarManualBuilder implements Builder{
        //Mandatory
        public CarType carType;
        public int seats;
        public  Engine engine;
        public Transmission transmission;
        //Optional
        public  TripComputer tripComputer=null;
        public  GPSNavigator gpsNavigator=null;
        public double fuel ;

        CarManualBuilder(CarType ct,int seats,Engine eg,Transmission t){
            this.carType=ct;
            this.seats=seats;
            this.engine=eg;
            this.transmission=t;
        }


    @Override
    public Builder setTripComputer(TripComputer tripComputer) {
        this.tripComputer = tripComputer;
        return this;
    }
```

```java

    @Override
    public Builder setGPSNavigator(GPSNavigator gpsNavigator) {
        this.gpsNavigator = gpsNavigator;
        return this;

    }
    @Override
    public Manual buildManual() {
        return new Manual(this);
    }
    @Override
    public Car buildCar() {
        return null;
    }
}

```

```java
public class Director {

    public Car constructSportsCar() {
        Builder cb=new CarBuilder(CarType.SPORTS_CAR,seats: 2,
            new Engine(volume: 3.0, mileage: 0),Transmission.SEMI_AUTOMATIC);
         return cb.setTripComputer(new TripComputer()).
        setGPSNavigator(new GPSNavigator()).
        buildCar();
    }

    public Car constructCityCar() {
        Builder cb=new CarBuilder(CarType.CITY_CAR,seats: 4,
        new Engine(volume: 1.2, mileage: 4),Transmission.AUTOMATIC);
        cb.setTripComputer(new TripComputer());
        cb.setGPSNavigator(new GPSNavigator());
        return cb.buildCar();

    }
}
```

Originally it was Director thoda phle wale Builder class kia constructor mai jisse Director ko pta lgta ki kis type ke hum chaie cephto

```java
public class Demo {

    Run | Debug
    public static void main(String[] args) {
        Director director = new Director();

        Car c1=director.constructSportsCar();
        System.out.println(c1);

        Car c2= director.constructCityCar();
        System.out.println(c2);


        Car c3 =new CarBuilder(CarType.CITY_CAR,seats: 3,new Engine(volume: 4 ,mileage: 3),
        Transmission.MANUAL ).buildCar();
        System.out.println(c3);

    }

}
```

This is I have modified ki yeh phle jaisa hi kuch karna hai chahe to director ko bata do warna direct se diya & builder chain lega

↓ If something new you need to make then contact Builder

Prototype design pattern

↓

means copy | clone

used when we want to copy heavy object

Object → very heavy object

↓

Instead of creating the same object from scratch, go and copy it.

↓

Helps us in saving our time and resources.

According to Gang of four –

Prototype design pattern specify the kind of objects to create using a prototypical instance and create new objects by copying this prototype.

To simplify, instead of creating objects from the scratch every time, you can make copies of an original instance and modify it as required.

Prototype is unique among the other creational patterns as it does not require a class but only an end object.

An object that supports cloning is called a *prototype*. When your objects have dozens of fields and hundreds of possible configurations, cloning them might serve as an alternative to subclassing.

Here's how it works: you create a set of objects, configured in various ways. When you need an object like the one you've configured, you just clone a prototype instead of constructing a new object from scratch.

Implementation guidelines -

we need to choose the prototype design pattern when -

1. Creating an object is an expensive op^n and it would be more efficient to copy an object.
2. we need objects that are similar to existing objects.
3. We need to hide the complexity of creating the new instances from the client.
4. When we want our systems to be independent of how its products are created, produced and represented.

```java
// Cloneable interface allows the implementing class to
// have its objects to be cloned instead of using a new operator.
// It is available in Java.lang.Cloneable.

abstract class Color implements Cloneable
{

    protected String colorName;

    abstract void addColor();

    public Object clone()
    {
        Object clone = null;
        try
        {
            clone = super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            e.printStackTrace();
        }
        return clone;
    }
}
```

```java
class blueColor extends Color
{
    public blueColor()
    {
        this.colorName = "blue";
    }

    @Override
    void addColor()
    {
        System.out.println("Blue color added");
    }

}

class blackColor extends Color{

    public blackColor()
    {
        this.colorName = "black";
    }

    @Override
    void addColor()
    {
        System.out.println("Black color added");
    }
}
```

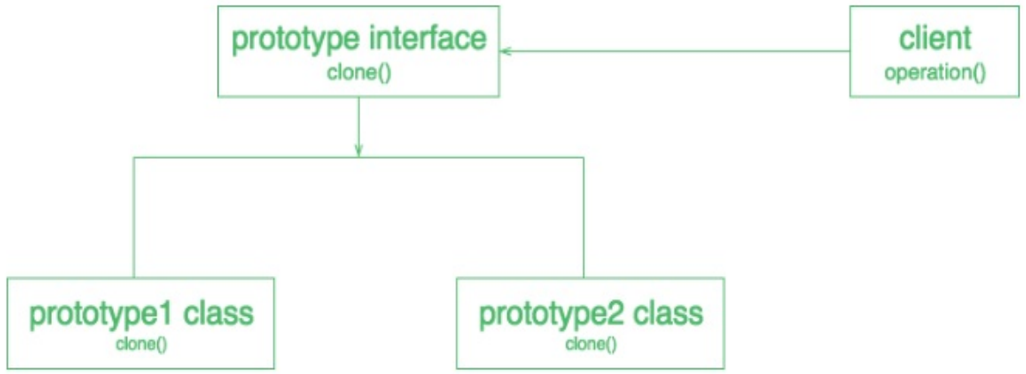
```
class ColorStore {

    private static Map<String, Color> colorMap = new HashMap<String, Color>();

    static
    {
        colorMap.put("blue", new blueColor());
        colorMap.put("black", new blackColor());
    }

    public static Color getColor(String colorName)
    {
        return (Color) colorMap.get(colorName).clone();
    }
}
```

```
// Driver class
class Prototype
{
    public static void main (String[] args)
    {
        ColorStore.getColor("blue").addColor();
        ColorStore.getColor("black").addColor();
        ColorStore.getColor("black").addColor();
        ColorStore.getColor("blue").addColor();
    }
}
```

we know differnce between shallow copy and deep copy
like tres is an arraylist and res is arraylist of arraylist
and we do res.add(tres);
it's a shallow copy now if we remove elements from tres it will be removed from res arraylist too,it's like sharing link of drive excel file and he can change too now both of you can change the other person has shallow copy.shallow==>(not complete),both have access

now we do res.add(new ArrayList<>(tres));
here we created new arraylist of same of tres
then change in tres will not affect res it's a deep copy
so it;s like you download the excel file and now you have your own copy of excel file so owner has no access of it so it;s deep copy

yha pe deep copy hi bna hai



Advantages of prototype pattern

1. It hides the complexities of creating objects.

2. It lets you add or remove objects at runtime.

3. It reduces the need of sub-classing.

4. The client can get new objects without knowing which type of object it will be.

ICloneable interface

It provides us the customized implementation that creates copy of an existing object.

ICloneable Interface → method

↓

clone()

↓

Provides support for Memberwise clone method.

Read Discuss Courses Practice Video

The **Java.lang.Cloneable** interface is a marker interface. It was introduced in JDK 1.0. There is a method clone() in the Object class. **Cloneable** interface is implemented by a class to make Object.clone() method valid thereby making field-for-field copy. This interface allows the implementing class to have its objects to be cloned instead of using a **new** operator.

```java
class Student {

    // attributes of Student class
    String name = null;
    int id = 0;

    // default constructor
    Student() {}

    // parameterized constructor
    Student(String name, int id)
    {
        this.name = name;
        this.id = id;
    }

    public static void main(String[] args)
    {
        // create an instance of Student
        Student s1 = new Student("Ashish", 121);

        // Try to clone s1 and assign
        // the new object to s2
        Student s2 = s1.clone();
    }
}
```

```
prog.java:28: error: incompatible types: Object cannot be converted
to Student
        Student s2 = s1.clone();
                              ^
1 error
```

Example 1: Below program explains that If you will try to Clone an object which doesn't implement the Cloneable interface, it will **CloneNotSupportedException**, which you may want to handle.

without cloneable if you try to clone the object then it will show the error see below

Example 2: Below code explains the proper usage of the Cloneable interface to make the Object.clone() method legal. Classes that implement this interface should override the Object.clone() method (which is protected) so that it can be invoked.

```
class A implements Cloneable {
    int i;
    String s;

    // A class constructor
    public A(int i, String s)
    {
        this.i = i;
        this.s = s;
    }

    // Overriding clone() method
    // by simply calling Object class
    // clone() method.
    @Override
    protected Object clone()
        throws CloneNotSupportedException
    {
        return super.clone();
    }
}
```

```
public class Test {
    public static void main(String[] args)
        throws CloneNotSupportedException
    {
        A a = new A(20, "GeeksForGeeks");

        // cloning 'a' and holding
        // new cloned object reference in b

        // down-casting as clone() return type is Object
        A b = (A)a.clone();

        System.out.println(b.i);
        System.out.println(b.s);
    }
}
```

Output

```
20
GeeksForGeeks
```

prototype vha pe use ho skta like 1st object ke lie DB mai hit kia but ab nii chahte ki phir se DB pe jaya jae nye object ke lie so you just copy previously made object or we can call that object as prototype

Prototype tum kisiko btaoge toh voh shallow copy aur deep copy poochega

```
class A{
   int a,b;
   ArrayList<Integer,Integer>al;
   A(int a ,int b,ArrayList<Integer,Integer> l){
    this.a=a;
    this.b=b;
   this.al=l;    //shallow copy
 // deep copy al=new ArrayList<>(l);
}
```

in shallow copy if you do change in l ,al will be changed too. in deep copy both are independent primitive ki hmesha deep copy milegi but Object data type mai dikkat aati