**###Prototype design pattern:**

The prototype design pattern comes under creational design pattern and uses object re-use principle. This pattern is most suitable when object creation is very complex and time consuming.

Prototype interface enables the object cloning via the concrete implementation. A class has to implement the prototype interface to achieve the cloning ability. Instead of creating the object from the scratch every time, the pattern provides a way to create exact copies of the original object instance to modify and use it as required.

**####Prototype working:**

This pattern uses cloning mechanism to clone objects from already created objects. Most appropriate way to create a copy of the existing object is to extend the ‘Cloneable’ interface in your own prototype interface. Then we can override the clone() method to create the copies of the object as required using the concrete class.

**####Approaches to copy objects using Cloneable interface:**

1. Directly use the cloning functionality of the object from its’ super object class. Create the raw copy of the object and modify it as per your requirement. This is the most independent approach where you can use the clone object at any level and do anything to take the required output.
2. Another approach is to tweak the overriding clone method itself to get the updated object right at creation time. No post modification required for the cloned object. This is a more restricted approach and can use in very specific requirements

**####What is Cloning:**

Cloning to create exact copy of an object however it’s up to your requirement to create your object because you can change the behavior of your object while cloning your object.

A Java class can achieve cloning ability by implementing the marker interface – Cloneable. The application should create a custom interface which implements the ‘Cloneable’ interface. Then, we can use the clone() method to get the copy of the required object.

**####Shallow Cloning:**

Suppose we have one object of Student and it contains further two more objects of class Address and Name.

Student

Address

Name

Fig: Student class with Address and Name

![Student Class]( https://github.com/PiyushMittl/java/blob/master/java\_designpattern\_prototype\_im1\_studentclass.png)

If we make shallow copy of an object previously discussed, we can make copy of “Student” only not its inner objects like “Address” and “Name”.

For example, we have one Student and we create one shallow copy of that already created object then what would happen is we will have two separate objects but the both of the object would be sharing same Address and Name object.

See below image for reference.

Student

Address

Name

Student

Fig: Shallow copy

![Student Class]( https://github.com/PiyushMittl/java/blob/master/java\_designpattern\_prototype\_im2\_shallowcopy.png)

**####Deep Cloning:**

Unlike the shallow copy, a deep copy is fully independent copy of existing object. If we copy one Student using deep cloning, then we will get entire separate copy of existing object.

Student

Address

Name

Student

Address

Name

Fig: Deep copy

![Student Class]( https://github.com/PiyushMittl/java/blob/master/java\_designpattern\_prototype\_im3\_deepcopy.png)

If we make deep copy of an object previously discussed, we can make copy of “Student” as well as its inner objects like “Address” and “Name”.

For example, we have one Student and we create one deep copy of that already created object then what would happen is we will have two separate objects and the both of the object would be having separate/independent copy of Address and Name object.

**####Cloneable interface and Clone():**

This is a protected method in java.lang.Object class. If we need to get any clones from class instances, we should implement the java.lang.Cloneable interface in that class. The clone() method will throw CloneNotSupportedException exception if the Cloneable interface is not implemented inside the required class. The clone() method reduces the expensive and heavy processing effort to create the exact copy of an object. That is it saves the cost of instantiating objects using the ‘new’ operator.

**####Prototype Pattern implementation steps:**

**\*\*Step1:\*\*** create one interface extending the cloneable interface.

Name it as the ‘Prototype interface’

**\*\*Step2:\*\*** include abstract method available in interface or define getClone() method.

**\*\*Step3:\*\*** Create concrete class which implements the prototype method.

Instance of this class will be cloned when required.

**\*\*Step4:\*\*** The client will use the concrete class to instantiate the object once and getClone() method to reuse the copy of the created object

**####Key term in Prototype Design Pattern:**

**\*\*Prototype:\*\*** this is the base interface which extends the Cloneable marker interface and contains the abstract method clone().

**\*\*ConcretePrototype:\*\*** this is the class which will implement Prototype interface and implement clone().

**\*\*Client:\*\*** This is the program which creates a new object by contacting the prototype to clone itself.

**####Example1:**

**(C)CakeStore**

**(I)Cake**

cakeStore()

main(String []):void

prepareCake():Cake

CheeseCake()

prepareCase():Cake

addSugar(String):void

addButter(String):void

addCheese(String):void

setName(String):void

toString():String

sugar: String

butter: String

cheese: String

name: String

**(C)CheeseCake**

![Student Class]( https://github.com/PiyushMittl/java/blob/master/java\_designpattern\_prototype\_im4\_classdiagarm.png)

**\*\*Cake.java\*\***

``` java

package com.piyush.designpattern.prototype;

public interface Cake extends Cloneable {

public Cake prepareCake();

}

```

**\*\*CheeseCake.java\*\***

``` java

package com.piyush.designpattern.prototype;

public class CheeseCake implements Cake {

private String sugar;

private String butter;

private String cheese;

private String name;

public Cake prepareCake() {

Cake cake = null;

try {

cake = (Cake) super.clone();

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

return cake;

}

void addSugar(String sugar) {

this.sugar = sugar;

}

void addButter(String butter) {

this.butter = butter;

}

public void addCheese(String cheese) {

this.cheese = cheese;

}

public void setName(String name) {

this.name = name;

}

@Override

public String toString() {

return "Sugar:" + sugar + ", Butter:" + butter + ", Name:" + name

+ ", Cheese:" + cheese;

}

}

```

**\*\*CakeStore.java\*\***

``` java

package com.piyush.designpattern.prototype;

public class CakeStore {

public static void main(String[] args) {

/\* Cheese cake preparation \*/

CheeseCake cake = new CheeseCake();

cake.addSugar("100g");

cake.addButter("200g");

cake.addCheese("Acapella");

System.out.println("Cake = " + cake.toString());

/\* Order with custom name \*/

CheeseCake cake1 = (CheeseCake) cake.prepareCake();

cake1.setName("Bastien");

System.out.println("Cake1 = " + cake1.toString());

/\* Order with custom name and customized cheese \*/

CheeseCake cake2 = (CheeseCake) cake.prepareCake();

cake2.addCheese("Extra Cheese");

cake2.setName("Adams");

System.out.println("Cake3 = " + cake2.toString());

}

}

```

```

Output

Cake = Sugar:100g, Butter:200g, Name:null, Cheese:Acapella

Cake1 = Sugar:100g, Butter:200g, Name:Bastien, Cheese:Acapella

Cake3 = Sugar:100g, Butter:200g, Name:Adams, Cheese:Extra Cheese

```

**####Example2:**

**\*\*Competitor.java\*\***

``` java

public interface Competitor extends Cloneable{

protected String skill;

public void getSkill();

public Object clone() {

Object clone = null;

try {

clone = super.clone();

}

catch (CloneNotSupportedException e) {

e.printStackTrace();

}

return clone;

}

}

```

**\*\*Singer.java\*\***

``` java

public Singer implements Competitor{

public Singer(){

this.skill = “Singing”;

}

//Override

public void getSkill(){

System.out.println(“Singing is my skill”)

}

}

**```**

**\*\*Dancer.java\*\***

``` java

public Dancer implements Competitor{

public Dancer(){

  this.skill = “Dancing”;

}

//Override

public void getSkill(){

  System.out.println(“Dancing is my skill”)

}

}

```

**\*\*Actor.java\*\***

``` java

public Actor implements Competitor{

public Actor(){

this.skill = “Acting”;

}

//Override

public void getSkill(){

System.out.println(“Acting is my skill”)

}

}

**```**

**\*\*Client.java\*\***

``` java

public Singer implements Competitor{

public Competitor competitor = new Competitor();

//Introduce Singer

Singer singer = (Singer)competitor.clone();

singer.getSkill();

//Introduce Dancer

Dancer dancer = (Dancer)competitor.clone();

dancer.getSkill();

//Introduce Actor

Actor actor = (Actor)competitor.clone();

actor.getSkill();

}

**```**