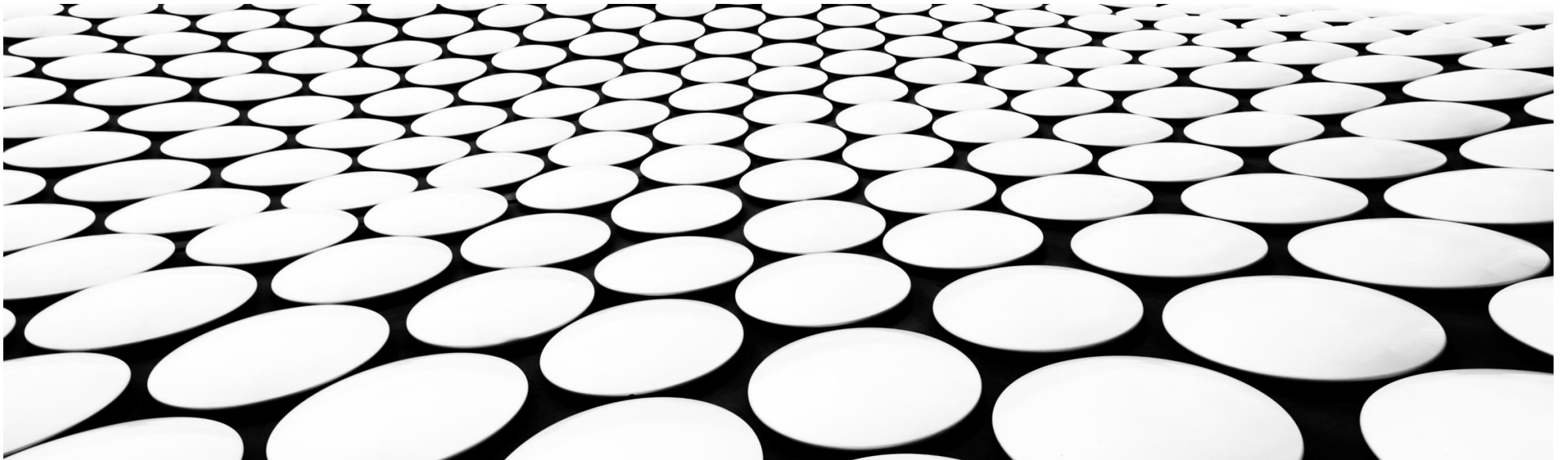

SOFTWARE DESIGN PATTERN (CREATIONAL PATTERN)

LEC3: SINGELTON, PROTOTYPE



SINGLETON DESIGN PATTERN



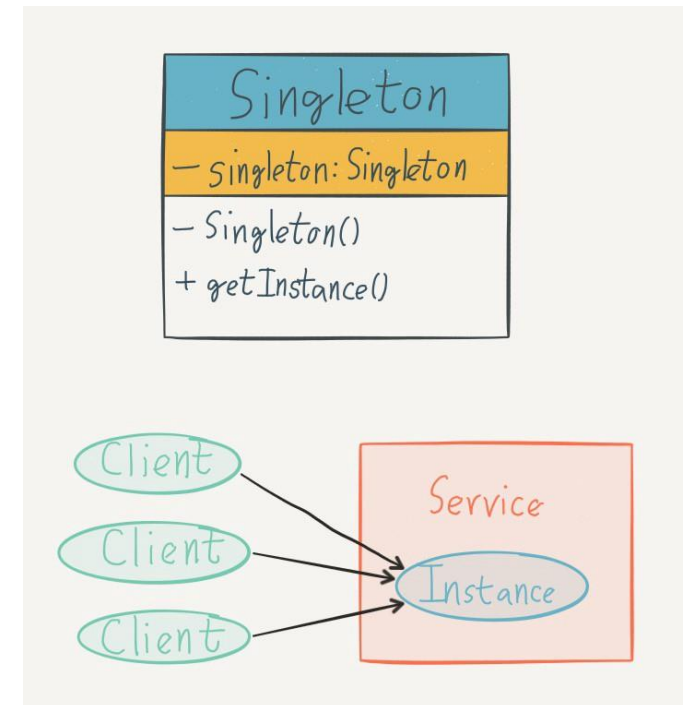
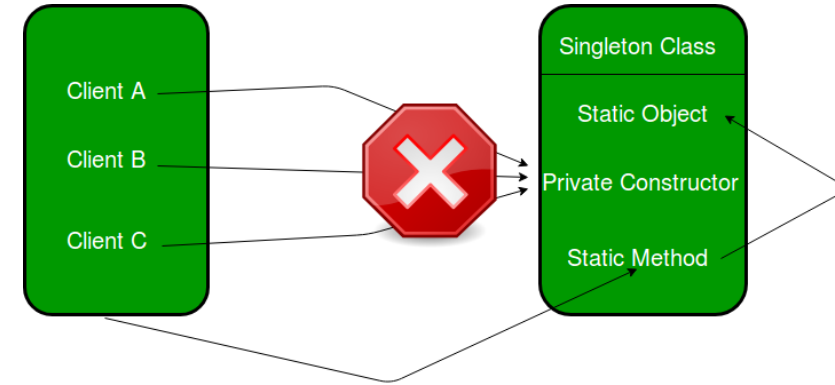
- Sometimes it's important for some classes to have exactly one instance
- Singleton Design Pattern "Ensure a class only has one instance, and provide a global point of access to it."
- A particular class should have only one instance. We will use only that instance whenever we are in need.

WHERE TO USE?

- The perfect example on when to use a singleton class is a logger implementation in which all the resource write in the same log file and is thread safe. Other examples:
- database connections and shared network resources;
- whenever the application needs to read a file from the server. Only in this case the object of the application will be able to access the files stored on the server.
- config files;
- The singleton pattern can be used with Abstract Factory, Builder, and Prototype design patterns to have a unique object
- When only one instance of a class are allowed.

STRUCTURE

- implement the this design pattern in the Java programming language, developers need to have the following:
- Static Member: It will create a single instance in the JVM memory as static are class level variables.
- Private Constructor: It will restrict the instantiation of the Singleton class from the outside world (i.e. Initialization of this class using the new keyword is prevented)
- Static factory method: This provides the global point of access to the Singleton object and returns the instance to the caller
- Participant: Singleton
 - defines an Instance operation that lets clients access its unique instance.



EXAMPLE: LOGGER

- Since there is an external Shared Resource (“log.txt”), we want to closely control how we communicate with it.
- We shouldn’t have to create the Logger class every time we want to access this Shared Resource. Is there any reason to?
- We need ONE.

What is wrong with this code?

```
public class Logger
{
    public Logger() { }

    public void LogMessage()
    {
        //Open File "log.txt"
        //Write Message
        //Close File
    }
}
```

LOGGER – AS A SINGLETON

```
public class Logger
{
    private Logger () {}

    private static Logger uniqueInstance;

    public static Logger getInstance()
    {

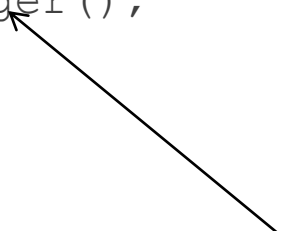
        if (uniqueInstance == null)
            uniqueInstance = new Logger();

        return uniqueInstance;

    }
}
```

- Lazy Instantiation
- Objects are only created when it is needed
- Helps control that we've created the Singleton just once.

Note the
parameterless
constructor



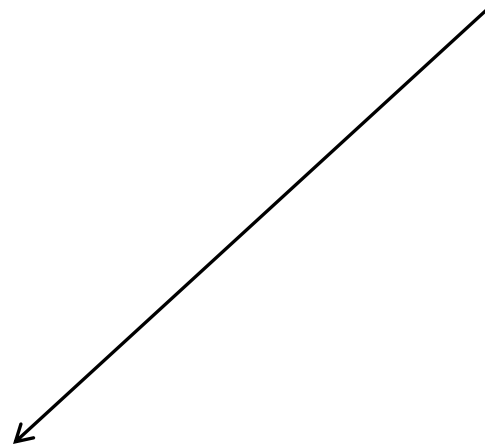
THREADING

```
public class Singleton
{
    private Singleton() {}

    private static Singleton uniqueInstance;
    public static Singleton getInstance()
    {
        if (uniqueInstance == null)
            uniqueInstance = new Singleton();

        return uniqueInstance;
    }
}
```

What would happen if two different threads accessed this line at the same time?



OPTION #1: SIMPLE LOCKING

```
public class Singleton
{
    private Singleton() {}

    private static Singleton uniqueInstance;

    public static Singleton getInstance()
    {
        synchronized(Singleton.class) {
            if (uniqueInstance == null)
                uniqueInstance = new Singleton();
        }

        return uniqueInstance;
    }
}
```


OPTION #2 – DOUBLE-CHECKED LOCKING

```
public class Singleton
{
    private Singleton() {}

    private volatile static Singleton uniqueInstance;

    public static Singleton getInstance()
    {
        if (uniqueInstance == null) {
            synchronized(Singleton.class) {
                if (uniqueInstance == null)
                    uniqueInstance = new Singleton();
            }
        }

        return uniqueInstance;
    }
}
```

Volatile keyword is used to modify the value of a variable by different threads. It is also used to make classes thread safe. It means that multiple threads can use a method and instance of the classes at the same time without any problem. The volatile keyword can be used either with primitive type or objects.

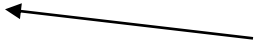
OPTION #3: “EAGER” INITIALIZATION

```
public class Singleton
{
    private Singleton() {}

    private static Singleton uniqueInstance = new Singleton()

    public static Singleton getInstance()
    {
        return uniqueInstance;
    }
}
```

Runtime guarantees
that this is thread-
safe



1. Instance is created the first time any member of the class is referenced.
2. Good to use if the application always creates; and if little overhead to create.

REAL-LIFE EXAMPLE

- Suppose you are a member of a football team, and in a tournament your team is going to play against another team.
- As per the rules of the game, the captain of each side must go for a toss to decide which side will get the ball first.
- So, if your team does not have a captain, you need to elect someone as a captain first. And at the same time, your team cannot have more than one captain.
- In this example, we have `MakeCaptain` class which can be used to get a captain object and it has a `private` constructor `private`, so that we cannot instantiate in normal fashion.
- When we attempt to create an instance of the class, we are checking whether we already have one available copy. If we do not have any such copy, we'll create it; otherwise, we'll simply reuse the existing copy.

```
public class MakeACaptain {

    private static MakeACaptain _captain;
    //We make the constructor private to prevent the use of "new"

    private MakeACaptain() {
    }

    public static MakeACaptain getCaptain() {

        if (_captain == null) {
            _captain = new MakeACaptain();
            System.out.println("New Captain selected for our team");
        } else {
            System.out.print("You already have a Captain for your team.");
            System.out.println("Send him for the toss.");
        }
        return _captain;
    }
}
```

```
public class SingletonPatternEx {  
  
    public static void main(String[] args) {  
        System.out.println("***Singleton Pattern Demo***\n");  
        System.out.println("Trying to make a captain for our team");  
        MakeACaptain c1 = MakeACaptain.getCaptain();  
        System.out.println("Trying to make another captain for our team");  
        MakeACaptain c2 = MakeACaptain.getCaptain();  
        if (c1 == c2) {  
            System.out.println("c1 and c2 are same instance");  
        }  
    }  
}
```

OUTPUT:

```
run:
***Singleton Pattern Demo***

Trying to make a captain for our team
New Captain selected for our team
Trying to make another captain for our team
You already have a Captain for your team.Send him for the toss.
c1 and c2 are same instance
```

IMPLEMENTATION

- In the preceding example, we wrote a class with a method that creates a new instance of the class if one does not exist.
- Do note:
 - The instance attribute in the class is defined private and static
 - The constructor of the class is made private so that there is no other way to instantiate the class
 - The accessor function for obtaining the reference to the singleton object is defined public and static
 - This example is known as Lazy Initialization – which means that it restricts the instance creation until it is requested for the first time.

PORS & CONS

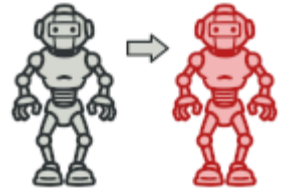
- **Pros:**

- the singleton class is instantiated only once in the life cycle of the app;
- you can use it as many times needed;
- the singleton class cannot be extended and if it is implemented correctly i.e. the get method should be synchronized and static, it is thread safe;

- **Cons:**

- problems during testing. (when the singleton class accesses a shared resource and the execution of the tests is important);

PROTOTYPE DESIGN PATTERN



- **Prototype** is a creational design pattern that allows cloning objects, even complex ones, without coupling to their specific classes.
- All prototype classes should have a common interface that makes it possible to copy objects even if their concrete classes are unknown.
- Prototype objects can produce full copies since objects of the same class can access each other's private fields.

WHERE TO USE

- When a system needs to be independent of how its objects are created, composed, and represented.
- When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.
- When the cost of creating an object is expensive or complicated.
- When you want to keep the number of classes in an application minimum.

STRUCTURE & PARTICIPANTS

■ Prototype

- declares an interface or abstract class for cloning itself.

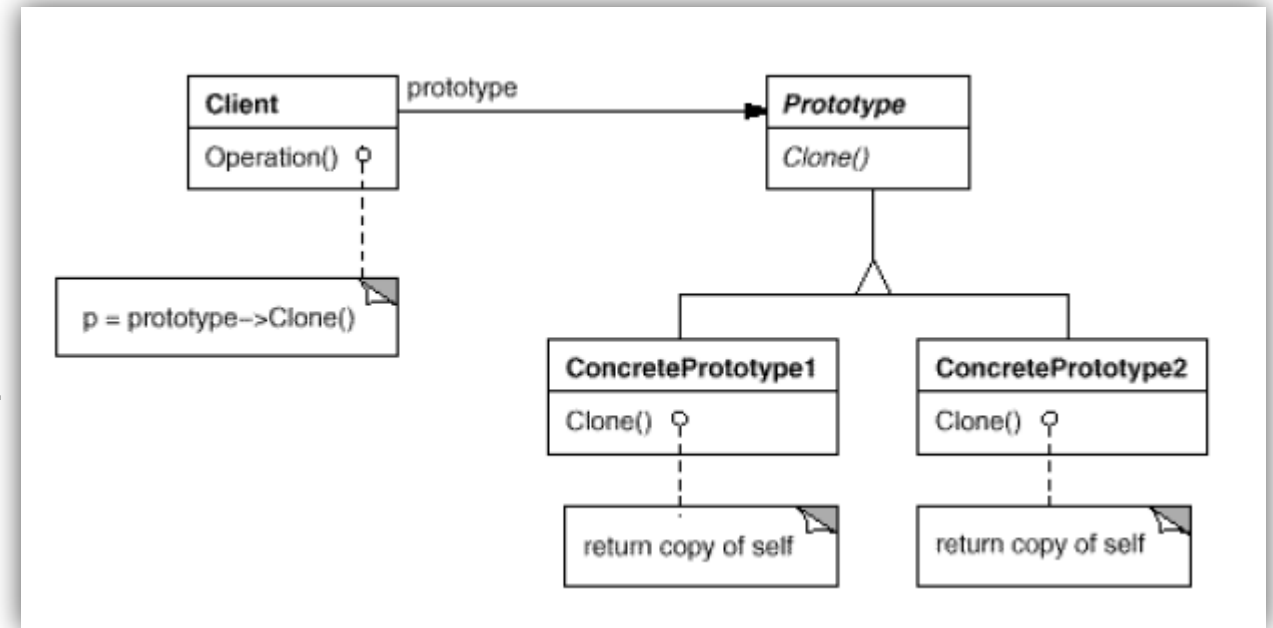
■ ConcretePrototype

- implements an operation for cloning itself.

■ Client

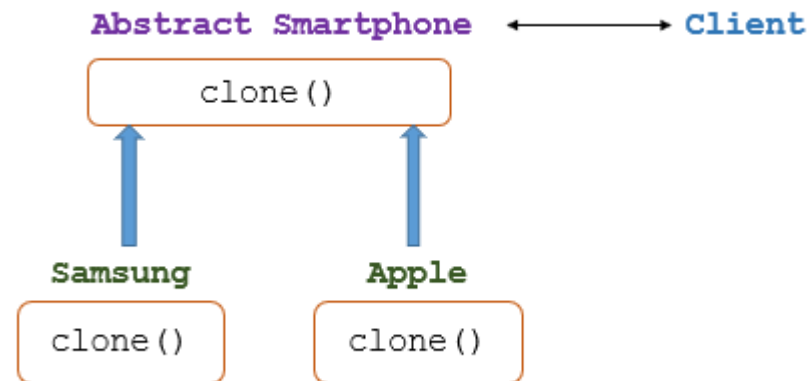
- creates a new object by asking a prototype to clone itself.

- **Cloneable** is an interface that is used to create the exact copy of an object. It exists in **java.lang** package. A class must implement the **Cloneable** interface if we want to create the clone of the class object. The clone() method of the Object class is used to create the clone of the object.



EXAMPLE

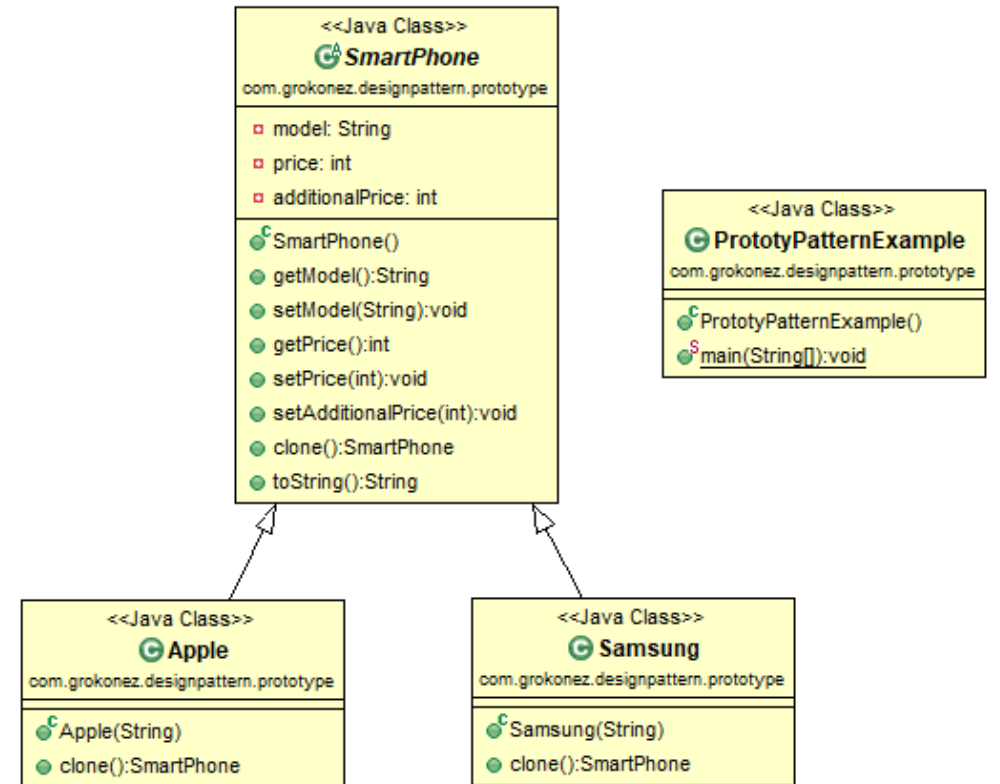
- a smartphone company produces thousands of mobiles with the same hardware and software. But with different model (color), the price could change.



- We have:
 - Basic prototype: SmartPhone abstract class with `clone()` method.
 - Concrete prototypes: Samsung and Apple implement the `clone()` method.

IMPLEMENT JAVA PROTOTYPE PATTERN CLASS DIAGRAM

- We set SmartPhone class with a default additionalPrice = 0. It will be also a field with default value in Samsung and Apple subclasses.
- – Samsung class and Apple class have their own base price.
- – PrototyPatternExample.java is the client.
- We will clone Samsung object and Apple object, then we add additional price for each object.



STEP BY STEP CREATE AN ABSTRACT CLASS THAT IMPLEMENTS CLONEABLE SMARTPHONE.JAVA

Syntax of the clone() method is : protected
Object clone() throws CloneNotSupportedException.
If the object's class doesn't implement Cloneable
interface then it throws
an exception 'CloneNotSupportedException'

```
package com.grokonez.designpattern.prototype;

public abstract class SmartPhone implements Cloneable {
    private String model;
    private int price;
    private int additionalPrice = 0;

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public int getPrice() {
        return price + this.additionalPrice;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    public void setAdditionalPrice(int additionalPrice) {
        this.additionalPrice = additionalPrice;
    }

    public SmartPhone clone() throws CloneNotSupportedException {
        return (SmartPhone) super.clone();
    }

    @Override
    public String toString() {
        return "SmartPhone [model=" + getModel() + ", price=" +
            getPrice() + "];"
    }
}
```

Create subclass with clone() method

Samsung.java

```
public class Samsung extends SmartPhone {  
  
    public Samsung(String model) {  
        this.setPrice(700);  
        this.setModel(model);  
    }  
  
    @Override  
    public SmartPhone clone() throws  
CloneNotSupportedException {  
        return (Samsung) super.clone();  
    }  
}
```

Apple.java

```
public class Apple extends SmartPhone {  
  
    public Apple(String model) {  
        this.setPrice(900);  
        this.setModel(model);  
    }  
  
    @Override  
    public SmartPhone clone() throws  
CloneNotSupportedException {  
        return (Apple) super.clone();  
    }  
}
```

RUN TEST

PROTOTYPATTERNEXAMPLE.JAVA

Result:

```
SmartPhone [model=Note10, price=700]
SmartPhone [model=iPhoneX, price=900]
=== Products for VIPs ===
SmartPhone [model=Note10, price=750]
SmartPhone [model=iPhoneX, price=1000]
```

```
public class PrototyPatternExample {

    public static void main(String[] args) throws CloneNotSupportedException {

        SmartPhone note10 = new Samsung("Note10");
        SmartPhone iphoneX = new Apple("iPhoneX");

        System.out.println(note10);
        System.out.println(iphoneX);

        System.out.println("=== Products for VIPs ===");

        SmartPhone note10Gold = note10.clone();
        note10Gold.setAdditionalPrice(50);
        System.out.println(note10Gold);

        SmartPhone iphoneX128 = iphoneX.clone();
        iphoneX128.setAdditionalPrice(100);
        System.out.println(iphoneX128);

    }
}
```


PROS & CONS OF USING PROTOTYPE PATTERN

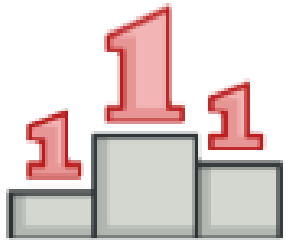
■ Pros

- Reusability: In case we want to create an instance of a class with many default values, or in some complicated processes, Prototype Pattern is useful. We can focus on other activities instead.
- Reducing initialization: We can create new instances at a cheaper cost.
- Simple copy process: We only need to call clone() method, it is simple and easy-reading.

■ Cons

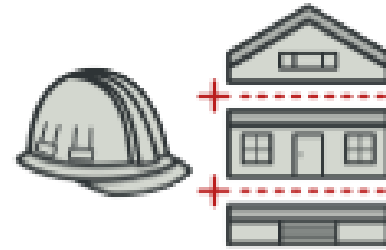
- Each subclass has to implement clone() method or alternative copy methods.
- Building clone for existing class may be complicated. For example, implementing Cloneable interface can constrain all subclasses/implementation to implement clone() method (some class may not need).

QUESTIONS!



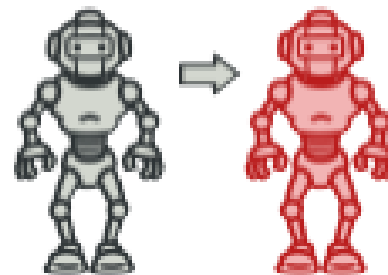
Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance.



Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



Prototype

Lets you copy existing objects without making your code dependent on their classes.