**Java Codes for Creational Patterns**

1. **Abstract factory Pattern Example 1**

// AddressFactory.java

public interface AddressFactory{

public Address createAddress();

public PhoneNumber createPhoneNumber();

}

//Address.java

public abstract class Address{

private String street;

private String city;

private String region;

private String postalCode;

public static final String EOL\_STRING = System.getProperty("line.separator");

public static final String SPACE = " ";

public String getStreet(){ return street; }

public String getCity(){ return city; }

public String getPostalCode(){ return postalCode; }

public String getRegion(){ return region; }

public abstract String getCountry();

public String getFullAddress(){

return street + EOL\_STRING +

city + SPACE + postalCode + EOL\_STRING;

}

public void setStreet(String newStreet){ street = newStreet; }

public void setCity(String newCity){ city = newCity; }

public void setRegion(String newRegion){ region = newRegion; }

public void setPostalCode(String newPostalCode){ postalCode = newPostalCode; }

}

//PhoneNimber.java

public abstract class PhoneNumber{

private String phoneNumber;

public abstract String getCountryCode();

public String getPhoneNumber(){ return phoneNumber; }

public void setPhoneNumber(String newNumber){

try{

Long.parseLong(newNumber);

phoneNumber = newNumber;

}catch (NumberFormatException exc){

}

}

}

**//USAddressFactory**

public class USAddressFactory implements AddressFactory{

public Address createAddress(){

return new USAddress();

}

public PhoneNumber createPhoneNumber(){

return new USPhoneNumber();

}

}

**//USAddress.java**

public class USAddress extends Address{

private static final String COUNTRY = "UNITED STATES";

private static final String COMMA = ",";

public String getCountry(){ return COUNTRY; }

public String getFullAddress(){

return getStreet() + EOL\_STRING +

getCity() + COMMA + SPACE + getRegion() +

SPACE + getPostalCode() + EOL\_STRING +

COUNTRY + EOL\_STRING;

}

}

//**USPhoneNumber.java**

public class USPhoneNumber extends PhoneNumber{

private static final String COUNTRY\_CODE = "01";

private static final int NUMBER\_LENGTH = 10;

public String getCountryCode(){ return COUNTRY\_CODE; }

public void setPhoneNumber(String newNumber){

if (newNumber.length() == NUMBER\_LENGTH){

super.setPhoneNumber(newNumber);

}

}

}

**//FrenchAddressFactory.java**

public class FrenchAddressFactory implements AddressFactory{

public Address createAddress(){

return new FrenchAddress();

}

public PhoneNumber createPhoneNumber(){

return new FrenchPhoneNumber();

}

}

**// FrenchAddress.java**

public class FrenchAddress extends Address{

private static final String COUNTRY = "FRANCE";

public String getCountry(){ return COUNTRY; }

public String getFullAddress(){

return getStreet() + EOL\_STRING +

getPostalCode() + SPACE + getCity() +

EOL\_STRING + COUNTRY + EOL\_STRING;

}

}

**//FrenchPhoneNumber.java**

public class FrenchPhoneNumber extends PhoneNumber{

private static final String COUNTRY\_CODE = "33";

private static final int NUMBER\_LENGTH = 9;

public String getCountryCode(){ return COUNTRY\_CODE; }

public void setPhoneNumber(String newNumber){

if (newNumber.length() == NUMBER\_LENGTH){

super.setPhoneNumber(newNumber);

}

}

}

//client application …….**RunPattern.java**

The RunPattern class provides an example of the AbstractFactory in use. It uses the USAddressFactory and the FrenchAddressFactory to create two different sets of address/phone number combinations. It is significant that once the factory objects have been loaded, we can deal with their products by using the Address and PhoneNumber interfaces. There are no method calls which depend on the distinction between a USAddress and a FrenchAddress.

public class RunPattern{

public static void main(String [ ] arguments){

System.out.println("Example for the AbstractFactory pattern");

System.out.println();

System.out.println(" (take a look in the RunPattern code. Notice that you

System.out.println(" use the Address and PhoneNumber classes when writing");

System.out.println(" almost all of the code. This allows you to write a

System.out.println(" generic framework, and plug in Concrete Factories");

System.out.println(" and Products to specialize the behavior of your code)");

System.out.println();

System.out.println("Creating U.S. Address and Phone Number:");

*AddressFactory usAddressFactory = new USAddressFactory();*

Address usAddress = usAddressFactory.createAddress();

PhoneNumber usPhone = usAddressFactory.createPhoneNumber();

usAddress.setStreet("142 Lois Lane");

usAddress.setCity("Metropolis");

usAddress.setRegion("WY");

usAddress.setPostalCode("54321");

usPhone.setPhoneNumber("7039214722");

System.out.println("U.S. address:");

System.out.println(usAddress.getFullAddress());

System.out.println("U.S. phone number:");

System.out.println(usPhone.getPhoneNumber());

System.out.println();

System.out.println();

System.out.println("Creating French Address and Phone Number:");

*AddressFactory frenchAddressFactory = new FrenchAddressFactory();*

Address frenchAddress = frenchAddressFactory.createAddress();

PhoneNumber frenchPhone = frenchAddressFactory.createPhoneNumber();

frenchAddress.setStreet("21 Rue Victor Hugo");

frenchAddress.setCity("Courbevoie");

frenchAddress.setPostalCode("40792");

frenchPhone.setPhoneNumber("011324290");

System.out.println("French address:");

System.out.println(frenchAddress.getFullAddress());

System.out.println("French phone number:");

System.out.println(frenchPhone.getPhoneNumber());

}

}

**Abstract factory Pattern Example 2**

/\* GUIFactory example -- \*/

interface GUIFactory {

public Button createButton();

}

class WinFactory implements GUIFactory {

public Button createButton() {

return new WinButton();

}

}

class OSXFactory implements GUIFactory {

public Button createButton() {

return new OSXButton();

}

}

interface Button {

public void paint();

}

class WinButton implements Button {

public void paint() {

System.out.println("I'm a WinButton");

}

}

class OSXButton implements Button {

public void paint() {

System.out.println("I'm an OSXButton");

}

}

class Application {

public Application(GUIFactory factory){

Button button = factory.createButton();

button.paint();

}

}

public class ApplicationRunner {

public static void main(String[] args) {

new Application(createOsSpecificFactory());

}

public static GUIFactory createOsSpecificFactory() {

int sys = readFromConfigFile("OS\_TYPE");

if (sys == 0) {

return new WinFactory();

} else {

return new OSXFactory();

}

}

}

1. **Builder Pattern**

**Example 1**

/\*\* "Product" \*/

class Pizza {

private String dough = "";

private String sauce = "";

private String topping = "";

public void setDough(String dough) {

this.dough = dough;

}

public void setSauce(String sauce) {

this.sauce = sauce;

}

public void setTopping(String topping) {

this.topping = topping;

}

}

/\*\* "Abstract Builder" \*/

abstract class PizzaBuilder {

protected Pizza pizza;

public Pizza getPizza() {

return pizza;

}

public void createNewPizzaProduct() {

pizza = new Pizza();

}

public abstract void buildDough();

public abstract void buildSauce();

public abstract void buildTopping();

}

/\*\* "ConcreteBuilder" \*/

class HawaiianPizzaBuilder extends PizzaBuilder {

public void buildDough() {

pizza.setDough("cross");

}

public void buildSauce() {

pizza.setSauce("mild");

}

public void buildTopping() {

pizza.setTopping("ham+pineapple");

}

}

/\*\* "ConcreteBuilder" \*/

class SpicyPizzaBuilder extends PizzaBuilder {

public void buildDough() {

pizza.setDough("pan baked");

}

public void buildSauce() {

pizza.setSauce("hot");

}

public void buildTopping() {

pizza.setTopping("pepperoni+salami");

}

}

/\*\* "Director" \*/

class Cook {

private PizzaBuilder pizzaBuilder;

public void setPizzaBuilder(PizzaBuilder pb) {

pizzaBuilder = pb;

}

public Pizza getPizza() {

return pizzaBuilder.getPizza();

}

public void constructPizza() {

pizzaBuilder.createNewPizzaProduct();

pizzaBuilder.buildDough();

pizzaBuilder.buildSauce();

pizzaBuilder.buildTopping();

}

}

/\*\* A given type of pizza being constructed. \*/

public class BuilderExample {

public static void main(String[] args) {

Cook cook = new Cook();

PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();

PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

cook.setPizzaBuilder(hawaiianPizzaBuilder);

cook.constructPizza();

Pizza hawaiian = cook.getPizza();

cook.setPizzaBuilder(spicyPizzaBuilder);

cook.constructPizza();

Pizza spicy = cook.getPizza();

}

}

**Example 2**

This code example shows how to use the Builder pattern to create an appointment for the PIM. The following list summarizes each class’s purpose:

AppointmentBuilder, MeetingBuilder – Builder classes

Scheduler – Director class

Appointment – Product

Address, Contact – Support classes, used to hold information relevant to the Appointment

InformationRequiredException – An Exception class produced when more data is required

For the base pattern, the AppointmentBuilder manages the creation of a complex product, an Appointment here.

The AppointmentBuilder uses a series of build methods—buildAppointment, buildLocation, buildDates, and buildAttendees—to create an Appointment and populate it with data.

//AppointmentBuilder.java

import java.util.Date;

import java.util.ArrayList;

public class AppointmentBuilder{

public static final int START\_DATE\_REQUIRED = 1;

public static final int END\_DATE\_REQUIRED = 2;

public static final int DESCRIPTION\_REQUIRED = 4;

public static final int ATTENDEE\_REQUIRED = 8;

public static final int LOCATION\_REQUIRED = 16;

protected Appointment appointment;

protected int requiredElements;

public void buildAppointment(){

appointment = new Appointment();

}

public void buildDates(Date startDate, Date endDate){

Date currentDate = new Date();

if ((startDate != null) && (startDate.after(currentDate))){

appointment.setStartDate(startDate);

}

if ((endDate != null) && (endDate.after(startDate))){

appointment.setEndDate(endDate);

}

}

public void buildDescription(String newDescription){

appointment.setDescription(newDescription);

}

public void buildAttendees(ArrayList attendees){

if ((attendees != null) && (!attendees.isEmpty())){

appointment.setAttendees(attendees);

}

}

public void buildLocation(Location newLocation){

if (newLocation != null){

appointment.setLocation(newLocation);

}

}

public Appointment getAppointment() throws InformationRequiredException{

requiredElements = 0;

if (appointment.getStartDate() == null){

requiredElements += START\_DATE\_REQUIRED;

}

if (appointment.getLocation() == null){

requiredElements += LOCATION\_REQUIRED;

}

if (appointment.getAttendees().isEmpty()){

requiredElements += ATTENDEE\_REQUIRED;

}

if (requiredElements > 0){

throw new InformationRequiredException(requiredElements);

}

return appointment;

}

public int getRequiredElements(){ return requiredElements; }

}

//**Appointment.java**

import java.util.ArrayList;

import java.util.Date;

public class Appointment{

private Date startDate;

private Date endDate;

private String description;

private ArrayList attendees = new ArrayList();

private Location location;

public static final String EOL\_STRING =

System.getProperty("line.separator");

public Date getStartDate(){ return startDate; }

public Date getEndDate(){ return endDate; }

public String getDescription(){ return description; }

public ArrayList getAttendees(){ return attendees; }

public Location getLocation(){ return location; }

public void setDescription(String newDescription){

description = newDescription;

}

public void setLocation(Location newLocation){ location = newLocation; }

public void setStartDate(Date newStartDate){ startDate = newStartDate; }

public void setEndDate(Date newEndDate){ endDate = newEndDate; }

public void setAttendees(ArrayList newAttendees){

if (newAttendees != null){

attendees = newAttendees;

}

}

public void addAttendee(Contact attendee){

if (!attendees.contains(attendee)){

attendees.add(attendee);

}

}

public void removeAttendee(Contact attendee){

attendees.remove(attendee);

}

public String toString(){

return " Description: " + description + EOL\_STRING +

" Start Date: " + startDate + EOL\_STRING +

" End Date: " + endDate + EOL\_STRING +

" Location: " + location + EOL\_STRING +

" Attendees: " + attendees;

}

}

**// Scheduler.java**

import java.util.Date;

import java.util.ArrayList;

public class Scheduler{

public Appointment createAppointment(AppointmentBuilder builder,

Date startDate, Date endDate, String description,

Location location, ArrayList attendees) throws InformationRequiredException{

if (builder == null){

builder = new AppointmentBuilder();

}

builder.buildAppointment();

builder.buildDates(startDate, endDate);

builder.buildDescription(description);

builder.buildAttendees(attendees);

builder.buildLocation(location);

return builder.getAppointment();

}

}

The MeetingBuilder class in Example 1.13 demonstrates one of the benefits of the Builder pattern. To add additional rules for the Appointment, extend the existing builder. In this case, the MeetingBuilder enforces an additional constraint: for a meeting Appointment, start and end dates must be specified.

**Example 1.13 MeetingBuilder.java**

import java.util.Date;

import java.util.Vector;

public class MeetingBuilder extends AppointmentBuilder{

public Appointment getAppointment() throws InformationRequiredException {

try {

super.getAppointment();

}

finally {

if (appointment.getEndDate() == null) {

requiredElements += END\_DATE\_REQUIRED;

}

if (requiredElements > 0) {

throw new InformationRequiredException(requiredElements);

}

}

return appointment;

}

}

**// InformationRequiredException.java**

public class InformationRequiredException extends Exception{

private static final String MESSAGE = "Appointment cannot be created because further information is required";

public static final int START\_DATE\_REQUIRED = 1;

public static final int END\_DATE\_REQUIRED = 2;

public static final int DESCRIPTION\_REQUIRED = 4;

public static final int ATTENDEE\_REQUIRED = 8;

public static final int LOCATION\_REQUIRED = 16;

private int informationRequired;

public InformationRequiredException(int itemsRequired){

super(MESSAGE);

informationRequired = itemsRequired;

}

public int getInformationRequired(){ return informationRequired; }

}

//Location

import java.io.Serializable;

public interface Location extends Serializable {

public String getLocation();

public void setLocation(String newLocation);

}

**// LocationImpl.java**

public class LocationImpl implements Location{

private String location;

public LocationImpl(){ }

public LocationImpl(String newLocation){

location = newLocation;

}

public String getLocation(){ return location; }

public void setLocation(String newLocation){ location = newLocation; }

public String toString(){ return location; }

}

**// Contact.java**

import java.io.Serializable;

public interface Contact extends Serializable{

public static final String SPACE = " ";

public String getFirstName();

public String getLastName();

public String getTitle();

public String getOrganization();

public void setFirstName(String newFirstName);

public void setLastName(String newLastName);

public void setTitle(String newTitle);

public void setOrganization(String newOrganization);

}

**// ContactImpl.java**

public class ContactImpl implements Contact{

private String firstName;

private String lastName;

private String title;

private String organization;

public ContactImpl(String newFirstName, String newLastName,

String newTitle, String newOrganization){

firstName = newFirstName;

lastName = newLastName;

title = newTitle;

organization = newOrganization;

}

public String getFirstName(){ return firstName; }

public String getLastName(){ return lastName; }

public String getTitle(){ return title; }

public String getOrganization(){ return organization; }

public void setFirstName(String newFirstName){

firstName =newFirstName; }

public void setLastName(String newLastName){ lastName = newLastName; }

public void setTitle(String newTitle){ title = newTitle; }

public void setOrganization(String newOrganization){

organization = newOrganization; }

public String toString(){

return firstName + SPACE + lastName;

}

}

The RunPattern file executes this example. It demonstrates the use of the Builder pattern by creating three separate Appointment objects using the AppointmentBuilder and MeetingBuilder.

**// RunPattern.java**

import java.util.Calendar;

import java.util.Date;

import java.util.ArrayList;

public class RunPattern{

private static Calendar dateCreator = Calendar.getInstance();

public static void main(String [] arguments){

Appointment appt = null;

System.out.println("Example for the Builder pattern");

System.out.println();

System.out.println("This example demonstrates the use of the Builder");

System.out.println("pattern to create Appointment objects for the PIM.");

System.out.println();

System.out.println("Creating a Scheduler for the example.");

Scheduler pimScheduler = new Scheduler();

System.out.println("Creating an AppointmentBuilder for the example.");

System.out.println();

AppointmentBuilder apptBuilder = new AppointmentBuilder();

try{

System.out.println("Creating a new Appointment with an AppointmentBuilder");

appt = pimScheduler.createAppointment(

apptBuilder, createDate(2066, 9, 22, 12, 30),

null, "Trek convention", new LocationImpl("Fargo, ND"),

createAttendees(4));

System.out.println("Successfully created an Appointment.");

System.out.println("Appointment information:");

System.out.println(appt);

System.out.println();

}catch (InformationRequiredException exc){

printExceptions(exc);

}

System.out.println("Creating a MeetingBuilder for the example.");

MeetingBuilder mtgBuilder = new MeetingBuilder();

try{

System.out.println("Creating a new Appointment with a MeetingBuilder");

System.out.println("(notice that the same create arguments will produce");

System.out.println(" an exception, since the MeetingBuilder enforces a");

System.out.println(" mandatory end date)");

appt = pimScheduler.createAppointment(

mtgBuilder, createDate(2066, 9, 22, 12, 30),

null, "Trek convention", new LocationImpl("Fargo, ND"),

createAttendees(4));

System.out.println("Successfully created an Appointment.");

System.out.println("Appointment information:");

System.out.println(appt);

System.out.println();

}catch (InformationRequiredException exc){

printExceptions(exc);

}

System.out.println("Creating a new Appointment with a MeetingBuilder");

System.out.println("(This time, the MeetingBuilder will provide an end date)");

try{

appt = pimScheduler.createAppointment(

mtgBuilder,

createDate(2002, 4, 1, 10, 00),

createDate(2002, 4, 1, 11, 30),

"OOO Meeting",

new LocationImpl("Butte, MT"),

createAttendees(2));

System.out.println("Successfully created an Appointment.");

System.out.println("Appointment information:");

System.out.println(appt);

System.out.println();

}

catch (InformationRequiredException exc){

printExceptions(exc);

}

}

public static Date createDate(int year, int month, int day, int hour, int minute){

dateCreator.set(year, month, day, hour, minute);

return dateCreator.getTime();

}

public static ArrayList createAttendees(int numberToCreate){

ArrayList group = new ArrayList();

for (int i = 0; i < numberToCreate; i++){

group.add(new ContactImpl("John", getLastName(i), "Employee (nonexempt)", "Yoyodyne

Corporation"));

}

return group;

}

public static String getLastName(int index){

String name = "";

switch (index % 6){

case 0: name = "Worfin";

break;

case 1: name = "Smallberries";

break;

case 2: name = "Bigbootee";

break;

case 3: name = "Haugland";

break;

case 4: name = "Maassen";

break;

case 5: name = "Sterling";

break;

}

return name;

}

public static void printExceptions(InformationRequiredException exc){

int statusCode = exc.getInformationRequired();

System.out.println("Unable to create Appointment: additional information is

required");

if ((statusCode & InformationRequiredException.START\_DATE\_REQUIRED) > 0){

System.out.println(" A start date is required for this appointment to be

complete.");

}

if ((statusCode & InformationRequiredException.END\_DATE\_REQUIRED) > 0){

System.out.println(" An end date is required for this appointment to be complete.");

}

if ((statusCode & InformationRequiredException.DESCRIPTION\_REQUIRED) > 0){

System.out.println(" A description is required for this appointment to be complete.");

}

if ((statusCode & InformationRequiredException.ATTENDEE\_REQUIRED) > 0){

System.out.println(" At least one attendee is required for this appointment to be complete.");

}

if ((statusCode & InformationRequiredException.LOCATION\_REQUIRED) > 0){

System.out.println(" A location is required for this appointment to becomplete.");

}

System.out.println();

}

}

1. **Method Factory Pattern**

**Example:** Let’s suppose an application asks for entering the name and sex of a person. If the sex is Male (M), it displays welcome message saying Hello Mr. <Name> and if the sex is Female (F), it displays message saying Hello Ms <Name>.  
  
The skeleton of the code can be given here.

|  |  |
| --- | --- |
| public class Person { | |
|  | // name string public String name; // gender : M or F private String gender;  public String getName() { return name; }  public String getGender() { return gender; } |
| }// End of class | |

This is a simple class Person having methods for name and gender. Now, we will have two sub-classes, Male and Female which will print the welcome message on the screen.

|  |  |
| --- | --- |
| public class Male extends Person { | |
|  | public Male(String fullName) { System.out.println("Hello Mr. "+fullName); } |
| }// End of class | |

Also, the class Female

|  |  |
| --- | --- |
| public class Female extends Person { | |
|  | public Female(String fullNname) { System.out.println("Hello Ms. "+fullNname); } |
| }// End of class | |

Now, we have to create a client, or a SalutationFactory which will return the welcome message depending on the data provided.

|  |  |
| --- | --- |
| public class SalutationFactory { | |
|  | public static void main(String args[]) { SalutationFactory factory = new SalutationFactory(); factory.getPerson(args[0], args[1]); }  public Person getPerson(String name, String gender) { if (gender.equals("M")) return new Male(name); else if(gender.equals("F")) return new Female(name); else return null; } |
| }// End of class | |

1. This class accepts two arguments from the system at runtime and prints the names.

Running the program:

After compiling and running the code on my computer with the arguments Prashant and M:

java Prashant M

The result returned is: “Hello Mr. Prashant”.

When to use a Factory Pattern?  
The Factory patterns can be used in following cases:  
1. When a class does not know which class of objects it must create.  
2. A class specifies its sub-classes to specify which objects to create.  
3. In programmer’s language (very raw form), you can use factory pattern where you have to create an object of any one of sub-classes depending on the data provided.

1. **Prototype Pattern**

**Example 1**

**// Copyable.java**

public interface Copyable{

public Object copy();

}

The Copyable interface defines a copy method and guarantees that any classes that implement the interface will define a copy operation. This example produces a shallow copy—that is, it copies the object references from the original address to the duplicate.

The code also demonstrates an important feature of the copy operation: not all fields must necessarily be duplicated. In this case, the address type is not copied to the new object. A user would manually specify a new address type from the PIM user interface.

**// Address.java**

public class Address implements Copyable{

private String type;

private String street;

private String city;

private String state;

private String zipCode;

public static final String EOL\_STRING =

System.getProperty("line.separator");

public static final String COMMA = ",";

public static final String HOME = "home";

public static final String WORK = "work";

public Address(String initType, String initStreet,String initCity, String initState, String initZip){

type = initType;

street = initStreet;

city = initCity;

state = initState;

zipCode = initZip;

}

public Address(String initStreet, String initCity, String initState, String initZip){

this(WORK, initStreet, initCity, initState, initZip);

}

public Address(String initType){

type = initType;

}

public Address(){ }

public String getType(){ return type; }

public String getStreet(){ return street; }

public String getCity(){ return city; }

public String getState(){ return state; }

public String getZipCode(){ return zipCode; }

public void setType(String newType){ type = newType; }

public void setStreet(String newStreet){ street = newStreet; }

public void setCity(String newCity){ city = newCity; }

public void setState(String newState){ state = newState; }

public void setZipCode(String newZip){ zipCode = newZip; }

***public Object copy(){***

***return new Address(street, city, state, zipCode);***

***}***

public String toString(){

return "\t" + street + COMMA + " " + EOL\_STRING +

"\t" + city + COMMA + " " + state + " " + zipCode;

}

}

**// RunPattern.java**

public class RunPattern{

public static void main(String [] arguments){

System.out.println("Example for Prototype pattern");

System.out.println();

System.out.println("This example will create an Address object,");

System.out.println(" which it will then duplicate by calling the");

System.out.println(" object's clone method.");

System.out.println();

System.out.println("Creating first address.");

Address address1 = new Address("8445 Silverado Trail", "Rutherford", "CA", "91734");

System.out.println("First address created.");

System.out.println(" Hash code = " + address1.hashCode());

System.out.println(address1);

System.out.println();

System.out.println("Creating second address using the clone() method.");

***Address address2 = (Address)address1.copy();***

System.out.println("Second address created.");

System.out.println(" Hash code = " + address2.hashCode());

System.out.println(address2);

System.out.println();

}

}

**Example 2**

/\*\*

\* Prototype Class

\*/

abstract class PrototypeFactory implements Cloneable {

public Object clone() throws CloneNotSupportedException {

// call Object.clone()

PrototypeFactory copy = (PrototypeFactory) super.clone();

//In an actual implementation of this pattern you might now change references to

//the expensive to produce parts from the copies that are held inside the prototype.

return copy;

}

abstract void prototypeFactory(int x);

abstract void printValue();

}

/\*\*

\* Concrete Prototypes to clone

\*/

class PrototypeImpl extends PrototypeFactory {

int x;

public PrototypeImpl(int x) {

this.x = x;

}

@Override

void prototypeFactory(int x) {

this.x = x;

}

public void printValue() {

System.out.println("Value :" + x);

}

}

/\*\*

\* Client Class

\*/

public class PrototypeExample {

private PrototypeFactory example; // Could have been a private Cloneable example.

public PrototypeExample(PrototypeFactory example) {

this.example = example;

}

public PrototypeFactory makeCopy() throws CloneNotSupportedException {

return (PrototypeFactory) this.example.clone();

}

public static void main(String args[]) {

try {

PrototypeFactory tempExample = null;

int num = 1000;

PrototypeFactory prot = new PrototypeImpl(1000);

PrototypeExample cm = new PrototypeExample(prot);

for (int i = 0; i < 10; i++) {

tempExample = cm.makeCopy();

tempExample.prototypeFactory(i \* num);

tempExample.printValue();

}

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

}

}

/\*

\*\*Code output\*\*

Value :0

Value :1000

Value :2000

Value :3000

Value :4000

Value :5000

Value :6000

Value :7000

Value :8000

Value :9000

\*/

**5. Singleton pattern**

**Example 1**

public class Singleton {

// Private constructor prevents instantiation from other classes

private Singleton() {

}

/\*\*

\* SingletonHolder is loaded on the first execution of Singleton.getInstance()

\* or the first access to SingletonHolder.INSTANCE, not before.

\*/

private static class SingletonHolder {

public static final Singleton INSTANCE = new Singleton();

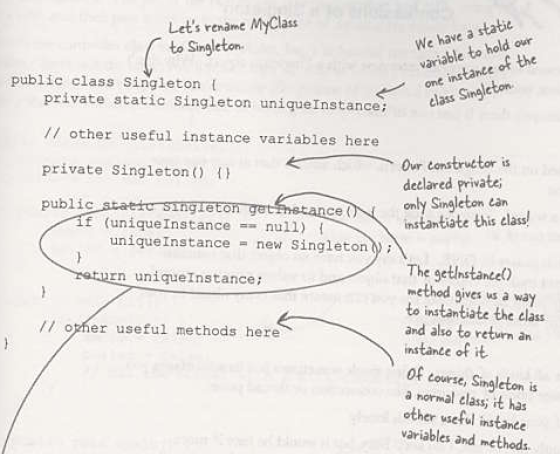
}

public static Singleton getInstance() {

return SingletonHolder.INSTANCE;

}

}

**Example 2**