Design Patterns

Builder Pattern & prototype pattern

Prepared by: wahidullah mudaser

introduction

- Builds a complex object using simple objects and using a step by step approach.
- This type pattern comes under creational pattern as this type pattern provides one of the best ways to create an object.

- Builder class builds the final object step by step. This builder is independent.

Implementation-Example

- We have considered a business of fast-food restaurant where a typical meal could be a burger and a cold drink.

- Burger could be either a Veg Burger or Chicken Burger and will be packed by a wrapper.

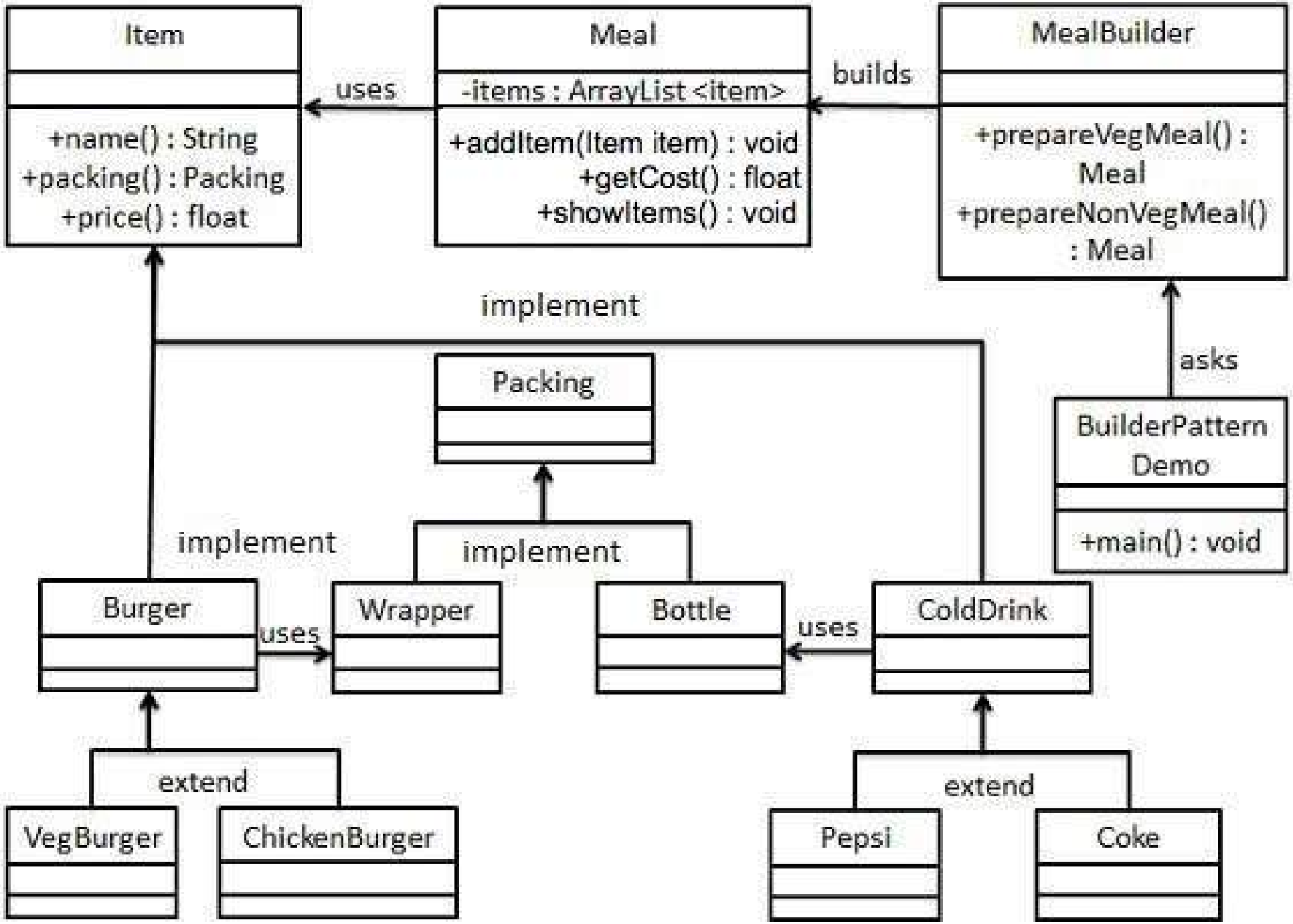
- Cold drink could be either a coke or PEPSI and will be packed in a bottle.

- We are going to create an *Item* interface representing food items such as burgers and cold drinks and concrete classes implementing the *Item* interface and a *Packing* interface representing packaging of food items and concrete classes implementing the *Packing* interface as burger would be packed in wrapper and cold drink would be packed as bottle.

Cont…

- We then create a *Meal* class having *ArrayList* of *Item* and a *MealBuilder* to build different types of *Meal* objects by combining *Item*.
- *BuilderPatternDemo*, our demo class will use *MealBuilder* to build a *Meal*.

Cont…


Item
+name() : String
+packing() : Packing
+price() : float
uses
Meal
-items : ArrayList <item>
+addItem(Item item) : void
+getCost() : float
+showItems() : void
builds
MealBuilder
+prepareVegMeal() : Meal
+prepareNonVegMeal() : Meal
implement
asks
Packing
BuilderPattern Demo
+main() : void
implement
implement
Burger
uses
Wrapper
Bottle
uses
ColdDrink
extend
extend
VegBurger
ChickenBurger
Pepsi
Coke

implementation-

- **Step-1:**
  - Create an interface Item representing food item and packing.
  - *Item.java*

```java
public interface Item {
   public String name();
   public Packing packing();
   public float price();
}
```

  - *Packing.java*

```java
public interface Packing {
   public String pack();
}
```

- **Step-2:**
  - Create concrete classes implementing the Packing interface.
  - *Wrapper.java*

```java
public class Wrapper implements Packing {

   @Override
   public String pack() {
      return "Wrapper";
   }
}
```

  - *Bottle.java*

```java
public class Bottle implements Packing {

   @Override
   public String pack() {
      return "Bottle";
   }
}
```

Step-3

- Create abstract classes implementing the item interface providing default functionalities.

- ***Burger.java***

```
public abstract class Burger implements Item {

   @Override
   public Packing packing() {
      return new Wrapper();
   }

   @Override
   public abstract float price();
}
```

- ***ColdDrink.java***

```
public abstract class ColdDrink implements Item {

        @Override
        public Packing packing() {
       return new Bottle();
        }

        @Override
        public abstract float price();
}
```

Step-4

- Create concrete classes extending Burger and ColdDrink classes

- ***VegBurger.java***

```
public class VegBurger extends Burger {

   @Override
   public float price() {
      return 25.0f;
   }

   @Override
   public String name() {
      return "Veg Burger";
   }
}
```

- ***ChickenBurger.java***

```
public class ChickenBurger extends Burger {

   @Override
   public float price() {
      return 50.5f;
   }

   @Override
   public String name() {
      return "Chicken Burger";
   }
}
```

- ***Coke.java***

```
public class Coke extends ColdDrink {

   @Override
   public float price() {
      return 30.0f;
   }

   @Override
   public String name() {
      return "Coke";
   }
}
```

- ***Pepsi.java***

```
public class Pepsi extends ColdDrink {

   @Override
   public float price() {
      return 35.0f;
   }

   @Override
   public String name() {
      return "Pepsi";
   }
}
```

Step-5

- Create a Meal class having Item objects defined above.
- *Meal.java*

```java
import java.util.ArrayList;
import java.util.List;

public class Meal {
   private List<Item> items = new ArrayList<Item>();

   public void addItem(Item item){
      items.add(item);
   }

   public float getCost(){
      float cost = 0.0f;

      for (Item item : items) {
         cost += item.price();
      }
      return cost;
   }

   public void showItems(){

      for (Item item : items) {
         System.out.print("Item : " + item.name());
         System.out.print(", Packing : " + item.packing().pack());
         System.out.println(", Price : " + item.price());
      }
   }
}
```

- **Step-6**
- Create a MealBuilder class, the actual builder class responsible to create Meal objects.
- *MealBuilder.java*

```
public class MealBuilder {

   public Meal prepareVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new VegBurger());
      meal.addItem(new Coke());
      return meal;
   }

   public Meal prepareNonVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new ChickenBurger());
      meal.addItem(new Pepsi());
      return meal;
   }
}
```

- **Step-7**
- BuiderPatternDemo uses MealBuider to demonstrate builder pattern.
- ***BuilderPatternDemo.java***

```
public class BuilderPatternDemo {
   public static void main(String[] args) {

      MealBuilder mealBuilder = new MealBuilder();

      Meal vegMeal = mealBuilder.prepareVegMeal();
      System.out.println("Veg Meal");
      vegMeal.showItems();
      System.out.println("Total Cost: " + vegMeal.getCost());

      Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
      System.out.println("\n\nNon-Veg Meal");
      nonVegMeal.showItems();
      System.out.println("Total Cost: " + nonVegMeal.getCost());
   }
}
```

Prototype Pattern

introduction

- Prototype pattern refers to creating duplicate object while keeping performance in mind.

- This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

- This pattern involves implementing a prototype interface which tells to create a clone of the current object.

Cont…

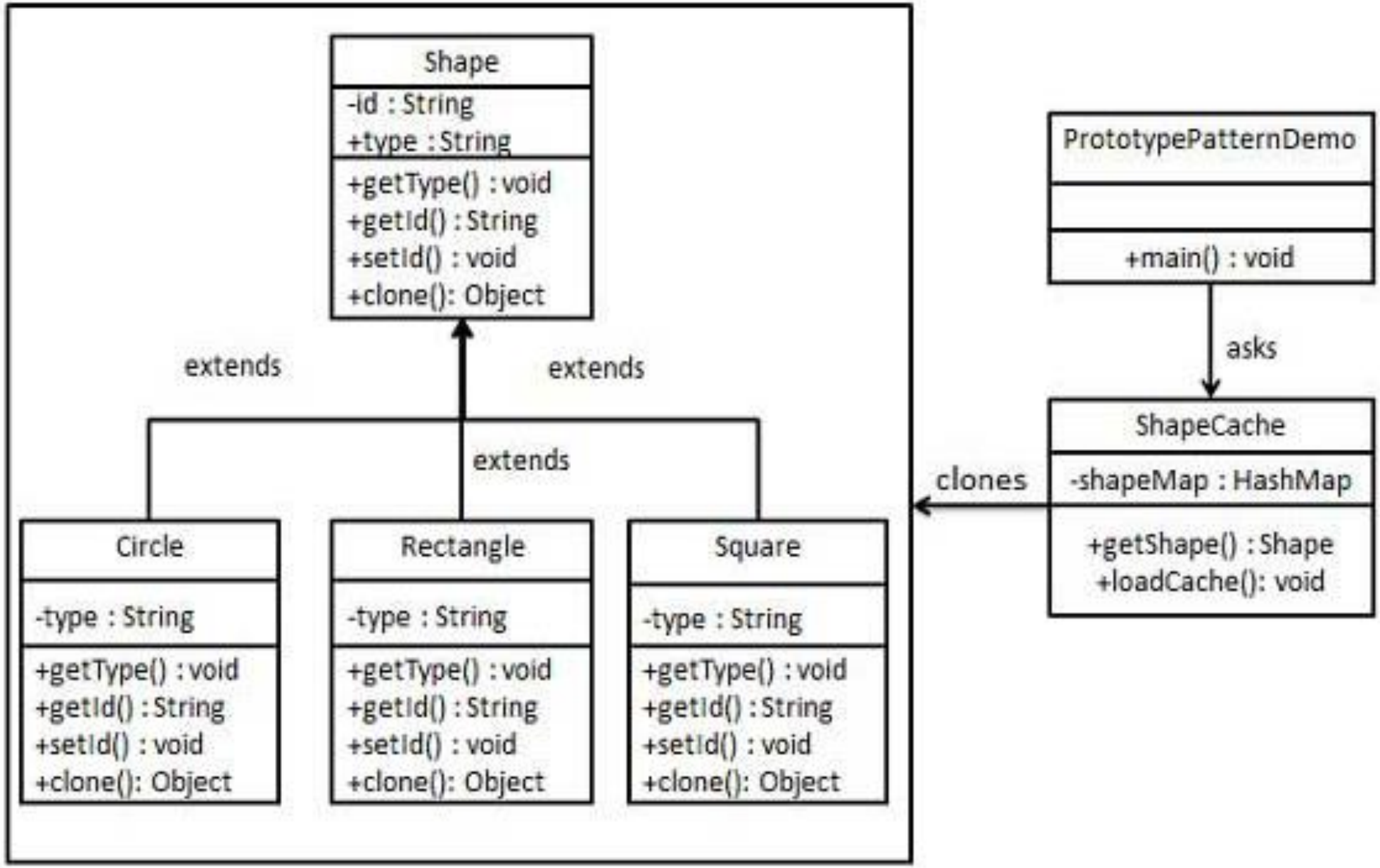
- This pattern is used when creation of object directly is costly.
- For example, an object is to be created after a costly database operation.
- We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.

Implementation-Example

- We're going to create an abstract class *Shape* and concrete classes extending the *Shape* class.

- A class *ShapeCache* is defined as a next step which stores shape objects in a *Hashtable* and returns their clone when requested.

- *PrototypPatternDemo*, our demo class will use *ShapeCache* class to get a *Shape* object.


Shape
-id : String
+type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
extends
extends
extends
Circle
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
Rectangle
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
Square
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
clones
PrototypePatternDemo
+main() : void
asks
ShapeCache
-shapeMap : HashMap
+getShape() : Shape
+loadCache(): void

Step-1

- Create an abstract class implementing *Clonable* interface.
- ***Shape.java***

```
public abstract class Shape implements Cloneable {

   private String id;
   protected String type;

   abstract void draw();

   public String getType(){
      return type;
   }

   public String getId() {
      return id;
   }

   public void setId(String id) {
      this.id = id;
   }

   public Object clone() {
      Object clone = null;

      try {
         clone = super.clone();

      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }

      return clone;
   }
}
```

Step-2

- Create concrete classes extending the above class.

- ***Rectangle.java***

```
public class Rectangle extends Shape {

   public Rectangle(){
     type = "Rectangle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

- ***Circle.java***

```
public class Circle extends Shape {

   public Circle(){
     type = "Circle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

- ***Square.java***

```
public class Square extends Shape {

   public Square(){
     type = "Square";
   }

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Step-3

- Create a class to get concrete classes from database and store them in a *Hashtable*.
- *ShapeCache.java*

```java
import java.util.Hashtable;

public class ShapeCache {

   private static Hashtable<String, Shape> shapeMap  = new Hashtable<String, Shape>();

   public static Shape getShape(String shapeId) {
      Shape cachedShape = shapeMap.get(shapeId);
      return (Shape) cachedShape.clone();
   }

   // for each shape run database query and create shape
   // shapeMap.put(shapeKey, shape);
   // for example, we are adding three shapes

   public static void loadCache() {
      Circle circle = new Circle();
      circle.setId("1");
      shapeMap.put(circle.getId(),circle);

      Square square = new Square();
      square.setId("2");
      shapeMap.put(square.getId(),square);

      Rectangle rectangle = new Rectangle();
      rectangle.setId("3");
      shapeMap.put(rectangle.getId(), rectangle);
   }
}
```

Step-4

- *PrototypePatternDemo* uses *ShapeCache* class to get clones of shapes stored in a *Hashtable*.

- *PrototypePatternDemo.java*

```
public class PrototypePatternDemo {
   public static void main(String[] args) {
      ShapeCache.loadCache();

      Shape clonedShape = (Shape) ShapeCache.getShape("1");
      System.out.println("Shape : " + clonedShape.getType());

      Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
      System.out.println("Shape : " + clonedShape2.getType());

      Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
      System.out.println("Shape : " + clonedShape3.getType());
   }
}
```

Any