260CT
Software Engineering

Dr. Yih-Ling Hedley
Email: aa0817@coventry.ac.uk

Software Design Patterns

- '**Gang of Four**' (GOF, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) analysed 23 Design patterns which provide solutions to general problems faced during software development as follows:
  - **Creational** patterns - manage the **creation of objects**
  - **Structural** patterns - describe **how objects are connected** together to form more complex objects
  - **Behavioural** patterns - describe how code is organized, to assign responsibility or roles to certain classes, and to specify the way objects **communicate** with each other

Design Patterns: Gang of Four

Creational	Structural	Behavioural
• AbstractFactory • Builder • FactoryMethod • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• ChainOfResponsibility • Command • Interpreter • Iterator • Mediator • Memento • Observer • State • Strategy • TemplateMethod • Visitor

Creational Patterns

- The **creational patterns**
  - **separates a system from the creation, composition and representation of its objects ,** which increases the system's flexibility in what, who, how, and when of object creation.
  - encapsulates the knowledge about which classes a system uses and **hides the details of how the instances of these classes are created and structured**.

Factory Pattern 1

- Factory Method pattern :
  - used to create objects, but allow subclasses to decide exactly which class to instantiate with various subclasses implementing the interface
  - instantiates the appropriate subclass based on information supplied by the client or extracted from the current state.
  - is useful when requiring the **creation of many different types of objects**, all derived from a common base type.

Factory Pattern 2

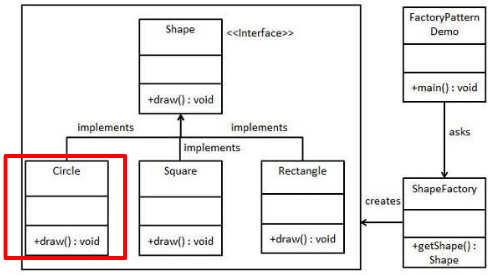
- Factory Method pattern :
  - defines **a method for creating the objects**, which subclasses can then override to specify the derived type to be created.
  - At run time, can be passed a description of an object and return **a base class pointer to a new instance of that object.**
  - Requires a well-designed **interface** for the base class, instead of casting the returned object (e.g., in Java)

Factory Method: Benefits

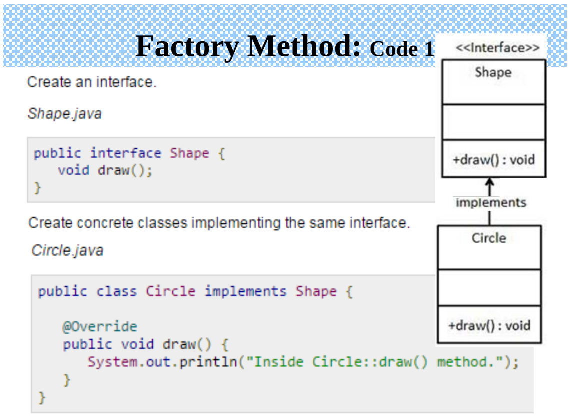
- To **support additional object types** – when an additional class is required, and objects are requested through a user interface, this pattern would simply pass on the new information to the factory, which would then handle the new types entirely.

Factory Method: Example and UML

- An object is created without exposing the creation logic to the external using a common interface.



Factory Method: Code 1



Create an interface.

Shape.java

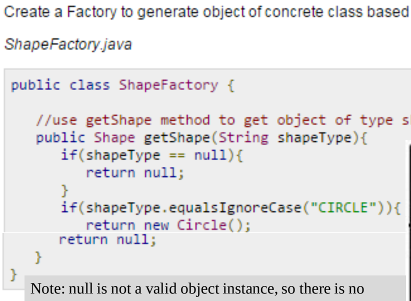
```
public interface Shape {
   void draw();
}
```

Create concrete classes implementing the same interface.

Circle.java

```
public class Circle implements Shape {

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

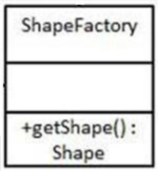
Factory Method: Code 2



Create a Factory to generate object of concrete class based on given information.

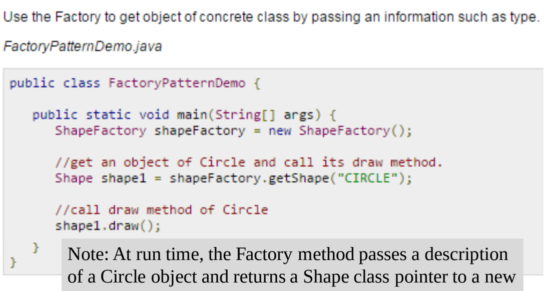
ShapeFactory.java

```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      return null;
   }
}
```



Note: null is not a valid object instance, so there is no memory allocated for it. It indicates that the object reference is not currently referring to an object.

Factory Method: Code 3



Use the Factory to get object of concrete class by passing an information such as type.

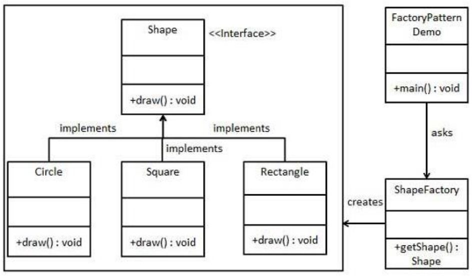
FactoryPatternDemo.java

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();
   }
}
```

Note: At run time, the Factory method passes a description of a Circle object and returns a Shape class pointer to a new instance of the Circle object, then completing the operation of drawing a circle.

Factory Method: Exercise

- Complete and implement the following in a programming language of your choice – i.e. Square, Rectangle



Abstract Factory Pattern

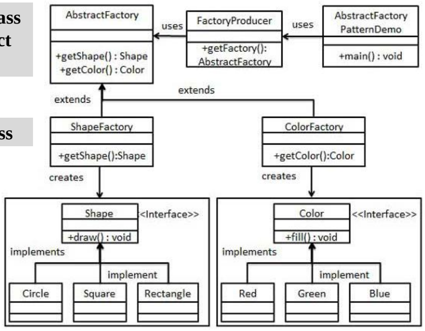
- **Abstract factory** – to provide an **interface for creating families of related or dependent objects** without specifying their concrete classes.

Abstract Factory: Example and UML

Super-class (abstract class)

Sub-class

Interface



Abstract Class and Interface: Exercise

- **Q: Differences between an abstract class and interface?**

Abstract Class and Interface: Feedback

- **Abstract Class**
  - cannot be instantiated, but to allow other classes to inherit from.
- **Interface**
  - is an entity that is has no implementation; it only has the signature to provide the definition of the methods without the body, which must be overridden by the implemented classes.

Abstract Class vs. Interface 1

- **Abstract Class vs. Interface**
  - Similarity: Abstract class and interface are used as a **contract** to define hierarchies for all subclasses or specific set of methods and their arguments.
  - Differences 1: for some programming languages, a class can implement more than one interface but can only inherit from one abstract class. For example, Java and C# do not support multiple inheritance, interfaces are used to implement multiple inheritance.
  - Differences 2: No fields (i.e. attributes) can be defined in interfaces. An abstract class can have fields and constants defined

Abstract Class vs. Interface 2

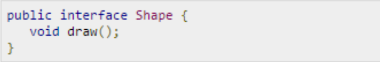
- **Abstract Class vs. Interface**
  - Difference 3: An abstract class enables a base class that might have one or more implemented methods but at least one or more methods are left unimplemented and declared abstract. The purpose of an abstract class is to provide a base class definition for how a set of derived classes will work and then allow the programmers to fill the implementation in the derived classes. For an interface, all the methods are not implemented.

See more from:
http://www.codeproject.com/Articles/11155/Abstract-Class-versus-Interface

Abstract Factory: Code 1

Create an interface for Shapes.

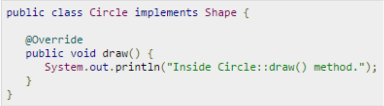
Shape.java



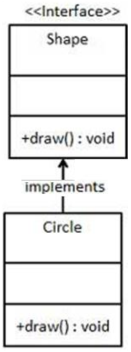
```
public interface Shape {
   void draw();
}
```

Create concrete classes implementing the same interface.

Circle.java



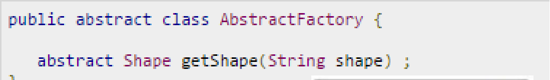
```
public class Circle implements Shape {

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



Abstract Factory: Code 2

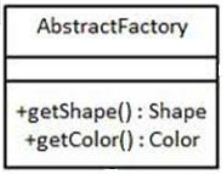
Create an Abstract class to get factories for Color and Shape Objects.

AbstractFactory.java



```
public abstract class AbstractFactory {

   abstract Shape getShape(String shape) ;
}
```

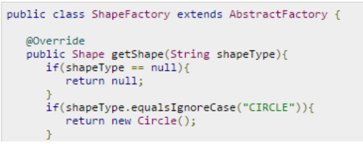
Note: In an abstract class, abstract methods contain no implementation details, it is up to individual subclasses to implement the methods



Abstract Factory: Code 3

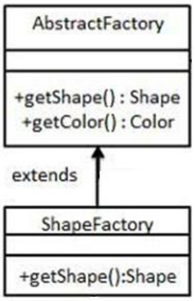
Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.

ShapeFactory.java



```
public class ShapeFactory extends AbstractFactory {

   @Override
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      }
```

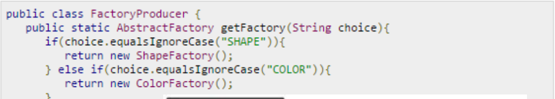
Note: extends keyword is used to implement a subclass that inherits its super-class



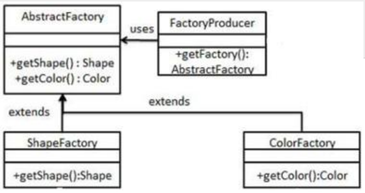
Abstract Factory: Code 4

Create a Factory generator/producer class to get factories by passing an information such as Shape or Color

FactoryProducer.java



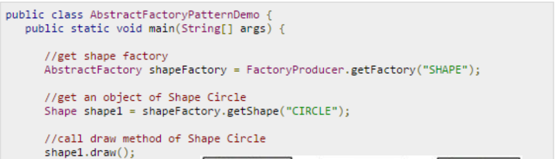
```
public class FactoryProducer {
   public static AbstractFactory getFactory(String choice){
      if(choice.equalsIgnoreCase("SHAPE")){
         return new ShapeFactory();
      } else if(choice.equalsIgnoreCase("COLOR")){
         return new ColorFactory();
      }
      return null;
   }
}
```



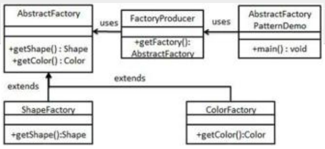
Abstract Factory: Code 5

Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

AbstractFactoryPatternDemo.java

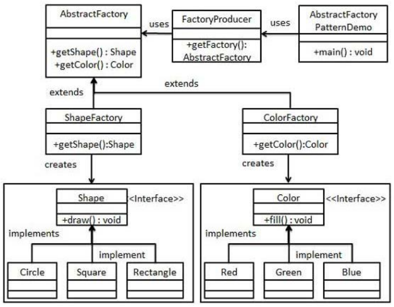


```
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {

      //get shape factory
      AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");

      //get an object of Shape Circle
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Shape Circle
      shape1.draw();
   }
}
```



Abstract Factory: Exercise

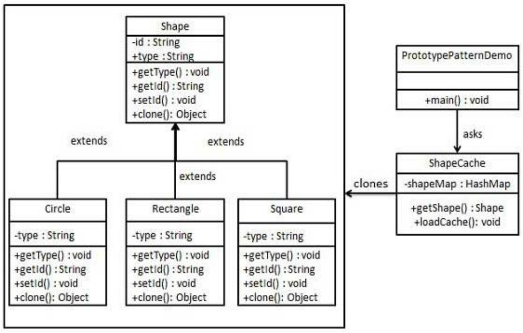
- Complete and implement the following in a programming language of your choice. i.e. Square, Rectangle, Colour (Red, Green, Blue)



Creational: Prototype Pattern

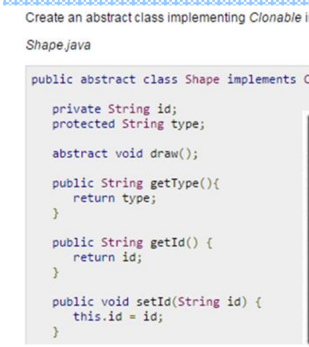
- Prototype pattern – used when the type of objects to create is determined by a prototypical instance, which is **cloned to produce new objects**.
  - is applied when the cost of creating a new object in the standard way (e.g., using the ***new*** keyword) is expensive and prohibitive for a given application.
  - Declares an abstract base class that specifies a pure virtual **clone() method**. Any class that derives itself from the abstract base class, implements the clone() operation.

Prototype Pattern: Example and UML



Prototype Pattern: Code 1

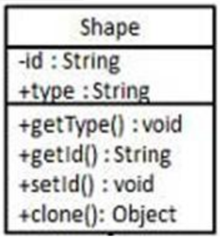
Clonable
https://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html



Create an abstract class implementing *Clonable* interface.

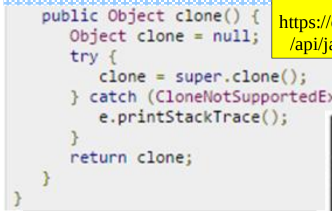
Shape.java

```
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;
   }
```



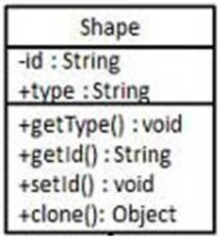
Prototype Pattern: Code 1.1

Clone method
https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#clone()

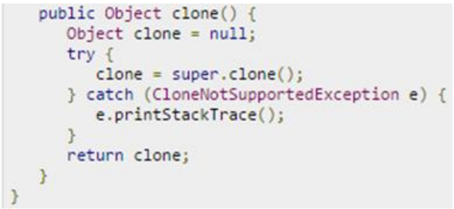


```
   public Object clone() {
      Object clone = null;
      try {
         clone = super.clone();
      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }
      return clone;
   }
}
```

Note: the abstract Shape class (implementing the Interface, Cloneable) specifies the clone method, which allows its subclasses to implement the method



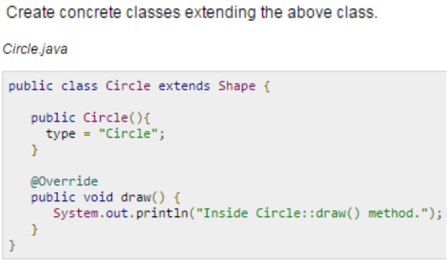
Prototype Pattern: Code 1.1



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

To allow an instance of a class to be cloned, the must implement Cloneable interface and must override Object's clone method with a public modifier.

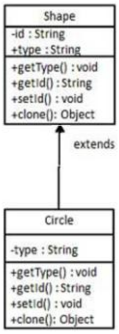
Prototype Pattern: Code 2



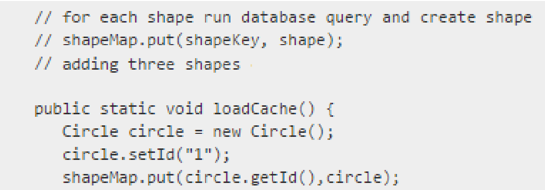
Create concrete classes extending the above class.

Circle.java

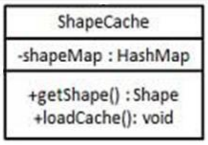
```
public class Circle extends Shape {

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

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



Prototype Pattern: Code 3

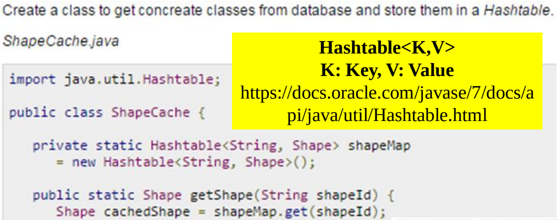


```
   // for each shape run database query and create shape
   // shapeMap.put(shapeKey, shape);
   // adding three shapes

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



An example of Circle is given here.

Prototype Pattern: Code 3.1

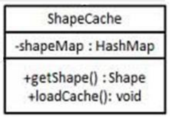


Create a class to get concreate classes from database and store them in a *Hashtable*.

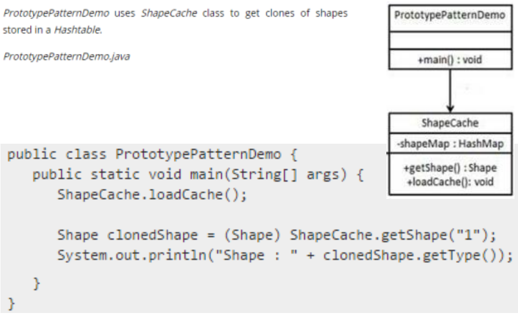
ShapeCache.java

```
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();
   }
```

Hashtable<K,V>
K: Key, V: Value
https://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html



Prototype Pattern: Code 4



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

PrototypePatternDemo.java

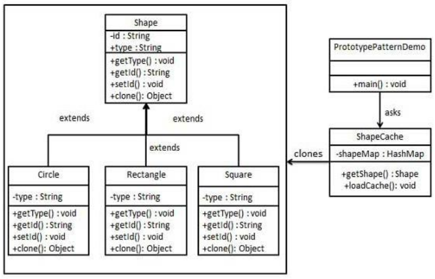
```
public class PrototypePatternDemo {
   public static void main(String[] args) {
      ShapeCache.loadCache();

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

   }
}
```

Prototype Pattern

- Prototype pattern –advantages:
  - speeds up the instantiation of very large, dynamically loaded classes by **copying objects**, instead of creating new instances)
    - creates duplicate object while **keeping performance**
    - used when creation of object is costly. For example, to create a object from a costly database operation. The solution is to cache the object and return its clone on next request and update the database as and when needed in order to reduce database calls.
  - keeps a record of identifiable parts of a large data structure that can be copied without knowing the subclass from which they were created.

Prototype Pattern: Exercise

- Complete and implement the following in a programming language of your choice – i.e. Rectangle, Square


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
PrototypePatternDemo
+main() : void
asks
ShapeCache
-shapeMap : HashMap
+getShape() : Shape
+loadCache(): void
clones