# Singleton

1. Singleton is used as a template for writing code, but its peculiarity is that it is used as a unique class that is disclosed only once. In the industries it is used in logging, caching and video systems.
2. Pseudocode:

package com.company;  
  
import java.util.Scanner;  
  
class Singleton{  
 private static Singleton *have*;  
 public int x;  
  
 private Singleton(int newx){  
 this.x = newx;  
 }  
  
 public static Singleton getInstance(int newx){  
 if(*have* == null){  
 *have* = new Singleton(newx);  
 }  
 return *have*;  
 }  
}  
  
public class Main{  
 public static void main(String args[]){  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.print("Insert a number: ");  
 int x = sc.nextInt();  
 Singleton fir = Singleton.*getInstance*(x);  
 System.*out*.print("Insert a number: ");  
 int y = sc.nextInt();  
 Singleton sec = Singleton.*getInstance*(y);  
 System.*out*.println("first number: " + fir.x);  
 System.*out*.println("second number: " + sec.x);  
 }  
}

1. Explanation:

As we can see despite the second inputted number the resultant value is same as the first number. It is the obvious example.

# Factory Method

1. Factory Method is a generative pattern that does design. It solves the problem of creating different products, while not defining specific product classes.
2. Pseudocode:

package com.company;  
  
import java.util.Scanner;  
  
interface ImageReader {  
 DecodedImage getDecodeImage();  
}  
  
class DecodedImage {  
 private String image;  
  
 public DecodedImage(String image) {  
 this.image = image;  
 }  
  
 @Override  
 public String toString() {  
 return image + ": is decoded";  
 }  
}  
  
class GifReader implements ImageReader {  
 private DecodedImage decodedImage;  
  
 public GifReader(String image) {  
 this.decodedImage = new DecodedImage(image);  
 }  
  
 @Override  
 public DecodedImage getDecodeImage() {  
 return decodedImage;  
 }  
}  
  
class JpegReader implements ImageReader {  
 private DecodedImage decodedImage;  
  
 public JpegReader(String image) {  
 decodedImage = new DecodedImage(image);  
 }  
  
 @Override  
 public DecodedImage getDecodeImage() {  
 return decodedImage;  
 }  
}  
  
public class Main{  
 public static void main(String[] args){  
 Scanner sc = new Scanner(System.*in*);  
 DecodedImage decodedImage;  
 ImageReader reader = null;  
 System.*out*.println("Input the name of picture: ");  
 String image = sc.nextLine();  
 String format = image.substring(image.indexOf('.') + 1, (image.length()));  
 if (format.equals("gif")) {  
 reader = new GifReader(image);  
 }  
 if (format.equals("jpeg")) {  
 reader = new JpegReader(image);  
 }  
 assert reader != null;  
 decodedImage = reader.getDecodeImage();  
 System.*out*.println(decodedImage);  
 }  
}

1. Explanation:

In this photo, at the beginning there is only an object reader that does not have a class. But then it will be listed as Gifreader or Jpegreader.

# Abstract Factory

1. An abstract factory is a generative design pattern that is needed to solve the problems of creating entire families, while not stating certain product classes.
2. Pseudocode:

package com.company;  
  
class Client {  
 private AbstractProductA productA;  
 private AbstractProductB productB;  
  
 Client(AbstractFactory factory) {  
 productA = factory.createProductA();  
 productB = factory.createProductB();  
 }  
  
 void execute() {  
 productB.interact(productA);  
 }  
}  
  
interface AbstractFactory {  
 AbstractProductA createProductA();  
 AbstractProductB createProductB();  
}  
  
interface AbstractProductA {  
 void interact(AbstractProductB b);  
}  
  
interface AbstractProductB {  
 void interact(AbstractProductA a);  
}  
  
class ConcreteFactory1 implements AbstractFactory {  
  
 @Override  
 public AbstractProductA createProductA() {  
 return new ProductA1();  
 }  
 @Override  
 public AbstractProductB createProductB() {  
 return new ProductB1();  
 }  
}  
  
class ConcreteFactory2 implements AbstractFactory {  
  
 @Override  
 public AbstractProductA createProductA() {  
 return new ProductA2();  
 }  
 @Override  
 public AbstractProductB createProductB() {  
 return new ProductB2();  
 }  
}  
  
class ProductA1 implements AbstractProductA {  
 @Override  
 public void interact(AbstractProductB b) {  
 System.*out*.println(this.getClass().getName() + " interacts with " + b.getClass().getName());  
 }  
}  
  
class ProductB1 implements AbstractProductB {  
  
 @Override  
 public void interact(AbstractProductA a) {  
 System.*out*.println(this.getClass().getName() + " interacts with " + a.getClass().getName());  
 }  
  
}  
  
class ProductA2 implements AbstractProductA {  
 @Override  
 public void interact(AbstractProductB b) {  
 System.*out*.println(this.getClass().getName() + " interacts with " + b.getClass().getName());  
 }  
}  
  
class ProductB2 implements AbstractProductB {  
  
 @Override  
 public void interact(AbstractProductA a) {  
 System.*out*.println(this.getClass().getName() + " interacts with " + a.getClass().getName());  
 }  
  
}  
  
public class Main {  
  
 public static void main(String[] args) {  
  
 AbstractFactory factory1 = new ConcreteFactory1();  
 Client client1 = new Client(factory1);  
 client1.execute();  
  
 AbstractFactory factory2 = new ConcreteFactory2();  
 Client client2 = new Client(factory2);  
 client2.execute();  
 }  
}

## Builder

1. Builder pattern is included in the creation templates, since it gives the best conditions for the creation of certain objects. It creates complex objects using ordinary objects and by certain steps.
2. Pseudocode:

package com.company;  
  
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; }  
  
 @Override  
 public String toString() {  
 return "Pizza{" +  
 "dough='" + dough + '\'' +  
 ", sauce='" + sauce + '\'' +  
 ", topping='" + topping + '\'' +  
 '}';  
 }  
}  
  
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();  
}  
  
  
class HawaiianPizzaBuilder extends PizzaBuilder {  
 public void buildDough() { pizza.setDough("cross"); }  
 public void buildSauce() { pizza.setSauce("mild"); }  
 public void buildTopping() { pizza.setTopping("ham+pineapple"); }  
}  
  
class SpicyPizzaBuilder extends PizzaBuilder {  
 public void buildDough() { pizza.setDough("pan baked"); }  
 public void buildSauce() { pizza.setSauce("hot"); }  
 public void buildTopping() { pizza.setTopping("pepperoni+salami"); }  
}  
  
  
class Waiter {  
 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();  
 }  
}  
  
  
public class Main {  
 public static void main(String[] args) {  
 Waiter waiter = new Waiter();  
 PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();  
 waiter.setPizzaBuilder(hawaiianPizzaBuilder);  
 waiter.constructPizza();  
 Pizza pizza = waiter.getPizza();  
 System.*out*.println(pizza.toString());  
 }  
}