**Singleton:**

**Lazy initialization**

This method uses [double-checked locking](http://en.wikipedia.org/wiki/Double_checked_locking_pattern#Usage_in_Java), which should not be used prior to [J2SE 5.0](http://en.wikipedia.org/wiki/Java_Platform,_Standard_Edition), as it is vulnerable to subtle bugs. The problem is that an out-of-order write may allow the instance reference to be returned before the Singleton constructor is executed.[[9]](http://en.wikipedia.org/wiki/Singleton_pattern#cite_note-IBM-9)

public class SingletonDemo {

private static SingletonDemo instance = null;

private SingletonDemo() { }

public static SingletonDemo getInstance() {

if (instance == null) {

synchronized (SingletonDemo .class){

if (instance == null) {

instance = new SingletonDemo ();

}

}

}

return instance;

}

}

**Eager initialization**

If the program will always need an instance, or if the cost of creating the instance is not too large in terms of time/resources, the programmer can switch to eager initialization, which always creates an instance:

public class Singleton {

private static final Singleton instance = new Singleton();

private Singleton() {}

public static Singleton getInstance() {

return instance;

}

}

Best:

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();

}

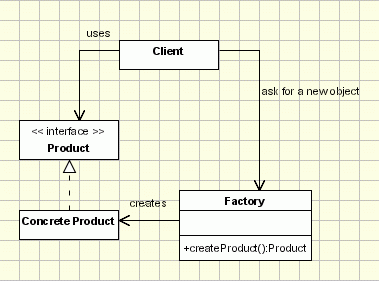
public static Singleton getInstance() {

return SingletonHolder.INSTANCE;

}

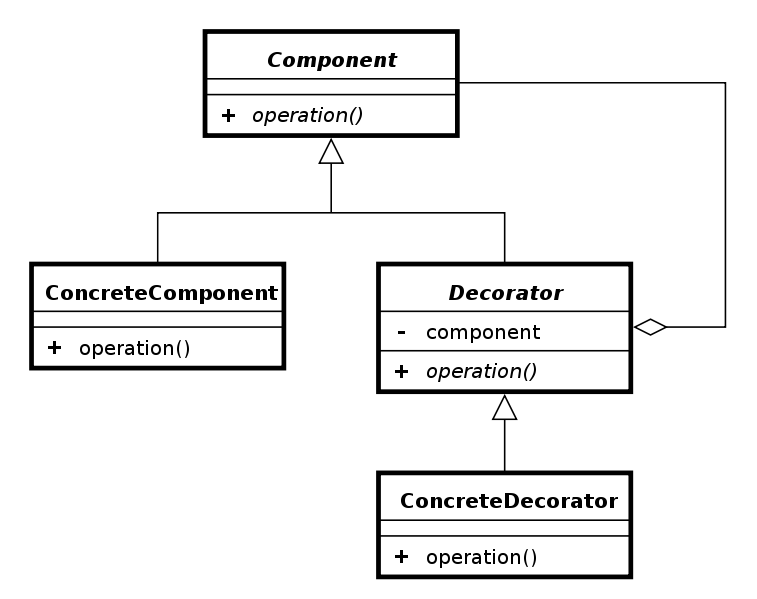
}

Factory:



public class ProductFactory{  
 public Product createProduct(String ProductID){  
 if (id==ID1)  
 return new OneProduct();  
 if (id==ID2) return  
 return new AnotherProduct();  
 ... // so on for the other Ids  
   
 return null; //if the id doesn't have any of the expected values  
 }  
 ...  
}

Decrorator:

[](http://upload.wikimedia.org/wikipedia/commons/e/e9/Decorator_UML_class_diagram.svg)

The next Java example illustrates the use of decorators using coffee making scenario. In this example, the scenario only includes cost and ingredients.

// The Coffee Interface defines the functionality of Coffee implemented by decorator

public abstract class Coffee {

public abstract double getCost(); // returns the cost of the coffee

public abstract String getIngredients(); // returns the ingredients of the coffee

}

// extension of a simple coffee without any extra ingredients

public class SimpleCoffee extends Coffee {

public double getCost() {

return 1;

}

public String getIngredients() {

return "Coffee";

}

}

The following classes contain the decorators for all Coffee classes, including the decorator classes themselves..

// abstract decorator class - note that it extends Coffee abstract class

public abstract class CoffeeDecorator extends Coffee {

protected final Coffee decoratedCoffee;

protected String ingredientSeparator = ", ";

public CoffeeDecorator(Coffee decoratedCoffee) {

this.decoratedCoffee = decoratedCoffee;

}

public double getCost() { // implementing methods of the abstract class

return decoratedCoffee.getCost();

}

public String getIngredients() {

return decoratedCoffee.getIngredients();

}

}

// Decorator Milk that mixes milk with coffee

// note it extends CoffeeDecorator

class Milk extends CoffeeDecorator {

public Milk(Coffee decoratedCoffee) {

super(decoratedCoffee);

}

public double getCost() { // overriding methods defined in the abstract superclass

return super.getCost() + 0.5;

}

public String getIngredients() {

return super.getIngredients() + ingredientSeparator + "Milk";

}

}

// Decorator Whip that mixes whip with coffee

// note it extends CoffeeDecorator

class Whip extends CoffeeDecorator {

public Whip(Coffee decoratedCoffee) {

super(decoratedCoffee);

}

public double getCost() {

return super.getCost() + 0.7;

}

public String getIngredients() {

return super.getIngredients() + ingredientSeparator + "Whip";

}

}

// Decorator Sprinkles that mixes sprinkles with coffee

// note it extends CoffeeDecorator

class Sprinkles extends CoffeeDecorator {

public Sprinkles(Coffee decoratedCoffee) {

super(decoratedCoffee);

}

public double getCost() {

return super.getCost() + 0.2;

}

public String getIngredients() {

return super.getIngredients() + ingredientSeparator + "Sprinkles";

}

}

Here's a test program that creates a Coffee instance which is fully decorated (i.e., with milk, whip, sprinkles), and calculate cost of coffee and prints its ingredients:

public class Main {

public static final void main(String[] args) {

Coffee c = new SimpleCoffee();

System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

c = new Milk(c);

System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

c = new Sprinkles(c);

System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

c = new Whip(c);

System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

// Note that you can also stack more than one decorator of the same type

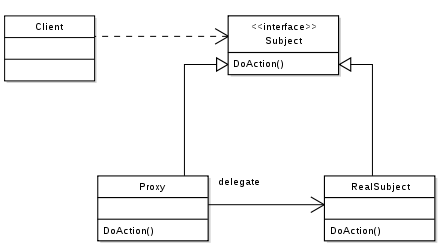
c = new Sprinkles(c);

System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

}

}

Proxy:

[](http://upload.wikimedia.org/wikipedia/commons/7/75/Proxy_pattern_diagram.svg)

interface Image {

public void displayImage();

}

//on System A

class RealImage implements Image {

private String filename = null;

/\*\*

\* Constructor

\* @param FILENAME

\*/

public RealImage(final String FILENAME) {

filename = FILENAME;

loadImageFromDisk();

}

/\*\*

\* Loads the image from the disk

\*/

private void loadImageFromDisk() {

System.out.println("Loading " + filename);

}

/\*\*

\* Displays the image

\*/

public void displayImage() {

System.out.println("Displaying " + filename);

}

}

//on System B

class ProxyImage implements Image {

private RealImage image = null;

private String filename = null;

/\*\*

\* Constructor

\* @param FILENAME

\*/

public ProxyImage(final String FILENAME) {

filename = FILENAME;

}

/\*\*

\* Displays the image

\*/

public void displayImage() {

if (image == null) {

image = new RealImage(filename);

}

image.displayImage();

}

}

class ProxyExample {

/\*\*

\* Test method

\*/

public static void main(String[] args) {

final Image IMAGE1 = new ProxyImage("HiRes\_10MB\_Photo1");

final Image IMAGE2 = new ProxyImage("HiRes\_10MB\_Photo2");

IMAGE1.displayImage(); // loading necessary

IMAGE1.displayImage(); // loading unnecessary

IMAGE2.displayImage(); // loading necessary

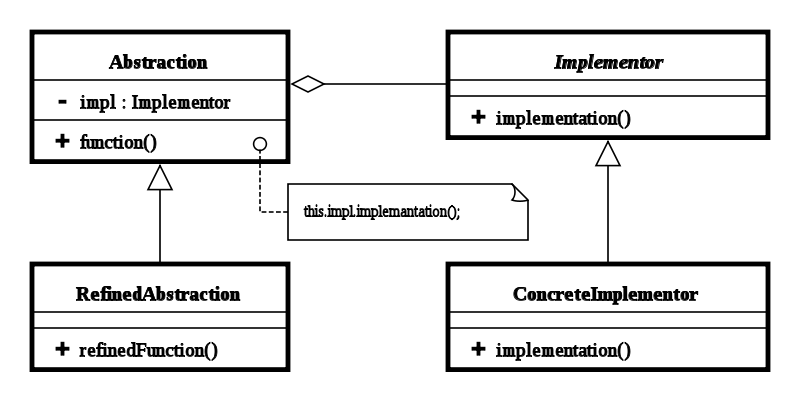
IMAGE2.displayImage(); // loading unnecessary

IMAGE1.displayImage(); // loading unnecessary

}

}

Bridge:

[](http://upload.wikimedia.org/wikipedia/commons/c/cf/Bridge_UML_class_diagram.svg)

/\*\* "Implementor" \*/

interface DrawingAPI {

public void drawCircle(double x, double y, double radius);

}

/\*\* "ConcreteImplementor" 1/2 \*/

class DrawingAPI1 implements DrawingAPI {

public void drawCircle(double x, double y, double radius) {

System.out.printf("API1.circle at %f:%f radius %f\n", x, y, radius);

}

}

/\*\* "ConcreteImplementor" 2/2 \*/

class DrawingAPI2 implements DrawingAPI {

public void drawCircle(double x, double y, double radius) {

System.out.printf("API2.circle at %f:%f radius %f\n", x, y, radius);

}

}

/\*\* "Abstraction" \*/

abstract class Shape {

protected DrawingAPI drawingAPI;

protected Shape(DrawingAPI drawingAPI){

this.drawingAPI = drawingAPI;

}

public abstract void draw(); // low-level

public abstract void resizeByPercentage(double pct); // high-level

}

/\*\* "Refined Abstraction" \*/

class CircleShape extends Shape {

private double x, y, radius;

public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI) {

super(drawingAPI);

this.x = x; this.y = y; this.radius = radius;

}

// low-level i.e. Implementation specific

public void draw() {

drawingAPI.drawCircle(x, y, radius);

}

// high-level i.e. Abstraction specific

public void resizeByPercentage(double pct) {

radius \*= pct;

}

}

/\*\* "Client" \*/

class BridgePattern {

public static void main(String[] args) {

Shape[] shapes = new Shape[] {

new CircleShape(1, 2, 3, new DrawingAPI1()),

new CircleShape(5, 7, 11, new DrawingAPI2()),

};

for (Shape shape : shapes) {

shape.resizeByPercentage(2.5);

shape.draw();

}

}

}

**Adaptor:**

[](http://upload.wikimedia.org/wikipedia/commons/d/d7/ObjectAdapter.png)

**Façade:**

class CPU {

public void freeze() { ... }

public void jump(long position) { ... }

public void execute() { ... }

}

class Memory {

public void load(long position, byte[] data) { ... }

}

class HardDrive {

public byte[] read(long lba, int size) { ... }

}

/\* Facade \*/

class Computer {

private CPU cpu;

private Memory memory;

private HardDrive hardDrive;

public Computer() {

this.cpu = new CPU();

this.memory = new Memory();

this.hardDrive = new HardDrive();

}

public void startComputer() {

cpu.freeze();

memory.load(BOOT\_ADDRESS, hardDrive.read(BOOT\_SECTOR, SECTOR\_SIZE));

cpu.jump(BOOT\_ADDRESS);

cpu.execute();

}

}

/\* Client \*/

class You {

public static void main(String[] args) {

Computer facade = new Computer();

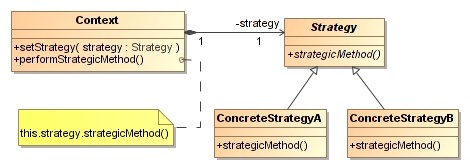
facade.startComputer();

}

}

**Strategy:**

The main difference (even though both patterns have the same UML) is that unlike the bridge pattern (which is a structural pattern), the strategy pattern is a behavioral pattern. Structural patterns suggest ways in which objects are composed or associated or inherited to forms larger objects i.e. they focus on object composition. While behavioral patterns deal with the algorithm or business logic (and not on the object creation itself) i.e. they focus on the collaboration between objects.

[](http://upload.wikimedia.org/wikipedia/commons/3/32/Strategy_Pattern.jpg)

/\*\* The classes that implement a concrete strategy should implement this.

\* The Context class uses this to call the concrete strategy. \*/

interface Strategy {

int execute(int a, int b);

}

/\*\* Implements the algorithm using the strategy interface \*/

class Add implements Strategy {

public int execute(int a, int b) {

System.out.println("Called Add's execute()");

return a + b; // Do an addition with a and b

}

}

class Subtract implements Strategy {

public int execute(int a, int b) {

System.out.println("Called Subtract's execute()");

return a - b; // Do a subtraction with a and b

}

}

class Multiply implements Strategy {

public int execute(int a, int b) {

System.out.println("Called Multiply's execute()");

return a \* b; // Do a multiplication with a and b

}

}

/\*\* Configured with a ConcreteStrategy object and maintains a reference to a Strategy object \*/

class Context {

private Strategy strategy;

public Context(Strategy strategy) {

this.strategy = strategy;

}

public int executeStrategy(int a, int b) {

return strategy.execute(a, b);

}

}

/\*\* Tests the pattern \*/

class StrategyExample {

public static void main(String[] args) {

Context context;

// Three contexts following different strategies

context = new Context(new Add());

int resultA = context.executeStrategy(3,4);

context = new Context(new Subtract());

int resultB = context.executeStrategy(3,4);

context = new Context(new Multiply());

int resultC = context.executeStrategy(3,4);

System.out.println("Result A : " + resultA );

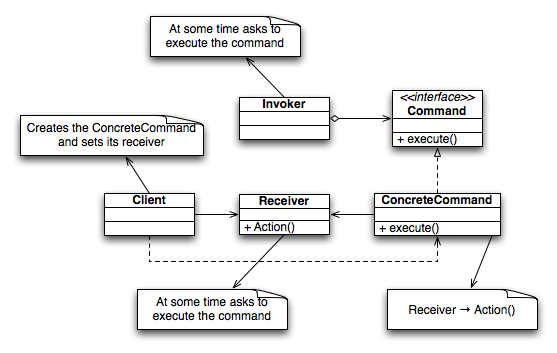
System.out.println("Result B : " + resultB );

System.out.println("Result C : " + resultC );

}

}

**Command Pattern: en.wikipedia.org/wiki/Command\_pattern**

[](http://upload.wikimedia.org/wikipedia/commons/8/8e/Command_Design_Pattern_Class_Diagram.png)

/\* The Command interface \*/

public interface Command {

void execute();

}

import java.util.List;

import java.util.ArrayList;

/\* The Invoker class \*/

public class Switch {

private List<Command> history = new ArrayList<Command>();

public Switch() {

}

public void storeAndExecute(Command cmd) {

this.history.add(cmd); // optional

cmd.execute();

}

}

/\* The Receiver class \*/

public class Light {

public Light() {

}

public void turnOn() {

System.out.println("The light is on");

}

public void turnOff() {

System.out.println("The light is off");

}

}

/\* The Command for turning on the light - ConcreteCommand #1 \*/

public class FlipUpCommand implements Command {

private Light theLight;

public FlipUpCommand(Light light) {

this.theLight = light;

}

public void execute(){

theLight.turnOn();

}

}

/\* The Command for turning off the light - ConcreteCommand #2 \*/

public class FlipDownCommand implements Command {

private Light theLight;

public FlipDownCommand(Light light) {

this.theLight = light;

}

public void execute() {

theLight.turnOff();

}

}

/\* The test class or client \*/

public class PressSwitch {

public static void main(String[] args){

Light lamp = new Light();

Command switchUp = new FlipUpCommand(lamp);

Command switchDown = new FlipDownCommand(lamp);

Switch s = new Switch();

try {

if (args[0].equalsIgnoreCase("ON")) {

s.storeAndExecute(switchUp);

}

else if (args[0].equalsIgnoreCase("OFF")) {

s.storeAndExecute(switchDown);

}

else {

System.out.println("Argument \"ON\" or \"OFF\" is required.");

}

} catch (Exception e) {

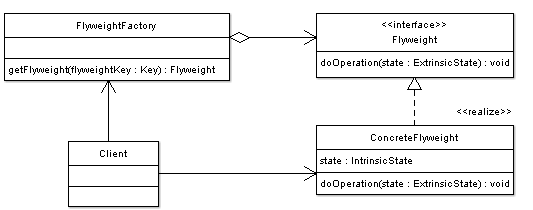
System.out.println("Arguments required.");

}

}

}

# Flyweight pattern <http://en.wikipedia.org/wiki/Flyweight_pattern>



import java.util.HashMap;

import java.util.Map;

// Flyweight object interface

interface CoffeeOrder {

public void serveCoffee(CoffeeOrderContext context);

}

// ConcreteFlyweight object that creates ConcreteFlyweight

class CoffeeFlavor implements CoffeeOrder {

private final String flavor;

public CoffeeFlavor(String newFlavor) {

this.flavor = newFlavor;

}

public String getFlavor() {

return this.flavor;

}

public void serveCoffee(CoffeeOrderContext context) {

System.out.println("Serving Coffee flavor " + flavor + " to table number " + context.getTable());

}

}

class CoffeeOrderContext {

private final int tableNumber;

public CoffeeOrderContext(int tableNumber) {

this.tableNumber = tableNumber;

}

public int getTable() {

return this.tableNumber;

}

}

//FlyweightFactory object

class CoffeeFlavorFactory {

private Map<String, CoffeeFlavor> flavors = new HashMap<String, CoffeeFlavor>();

public CoffeeFlavor getCoffeeFlavor(String flavorName) {

CoffeeFlavor flavor = flavors.get(flavorName);

if (flavor == null) {

flavor = new CoffeeFlavor(flavorName);

flavors.put(flavorName, flavor);

}

return flavor;

}

public int getTotalCoffeeFlavorsMade() {

return flavors.size();

}

}

class TestFlyweight {

/\*\* The flavors ordered. \*/

private static CoffeeFlavor[] flavors = new CoffeeFlavor[100];

/\*\* The tables for the orders. \*/

private static CoffeeOrderContext[] tables = new CoffeeOrderContext[100];

private static int ordersMade = 0;

private static CoffeeFlavorFactory flavorFactory;

public static void takeOrders(String flavorIn, int table) {

flavors[ordersMade] = flavorFactory.getCoffeeFlavor(flavorIn);

tables[ordersMade++] = new CoffeeOrderContext(table);

}

public static void main(String[] args) {

flavorFactory = new CoffeeFlavorFactory();

takeOrders("Cappuccino", 2);

takeOrders("Cappuccino", 2);

takeOrders("Frappe", 1);

takeOrders("Frappe", 1);

takeOrders("Xpresso", 1);

takeOrders("Frappe", 897);

takeOrders("Cappuccino", 97);

takeOrders("Cappuccino", 97);

takeOrders("Frappe", 3);

takeOrders("Xpresso", 3);

takeOrders("Cappuccino", 3);

takeOrders("Xpresso", 96);

takeOrders("Frappe", 552);

takeOrders("Cappuccino", 121);

takeOrders("Xpresso", 121);

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

flavors[i].serveCoffee(tables[i]);

}

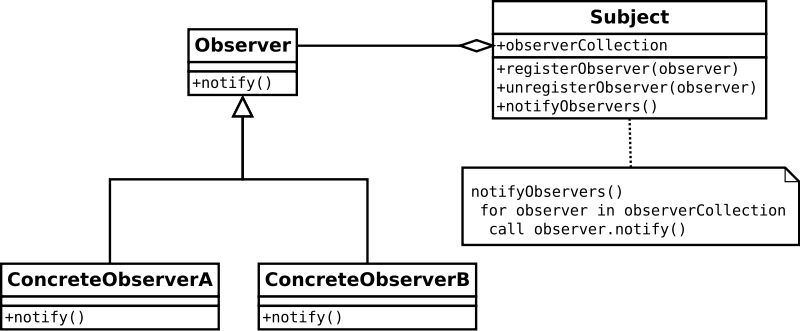
System.out.println(" ");

System.out.println("total CoffeeFlavor objects made: " + flavorFactory.getTotalCoffeeFlavorsMade());

}

}

**Observer Pattern**

[](http://upload.wikimedia.org/wikipedia/commons/8/8d/Observer.svg)

/\* File Name : EventSource.java \*/

package org.wikipedia.obs;

import java.util.Observable; //Observable is here

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

public class EventSource extends Observable implements Runnable {

public void run() {

try {

final InputStreamReader isr = new InputStreamReader(System.in);

final BufferedReader br = new BufferedReader(isr);

while (true) {

String response = br.readLine();

setChanged();

notifyObservers(response);

}

}

catch (IOException e) {

e.printStackTrace();

}

}

}

/\* File Name: ResponseHandler.java \*/

package org.wikipedia.obs;

import java.util.Observable;

import java.util.Observer; /\* this is Event Handler \*/

public class ResponseHandler implements Observer {

private String resp;

public void update(Observable obj, Object arg) {

if (arg instanceof String) {

resp = (String) arg;

System.out.println("\nReceived Response: " + resp );

}

}

}

/\* Filename : MyApp.java \*/

/\* This is the main program \*/

package org.wikipedia.obs;

public class MyApp {

public static void main(String[] args) {

System.out.println("Enter Text >");

// create an event source - reads from stdin

final EventSource eventSource = new EventSource();

// create an observer

final ResponseHandler responseHandler = new ResponseHandler();

// subscribe the observer to the event source

eventSource.addObserver(responseHandler);

// starts the event thread

Thread thread = new Thread(eventSource);

thread.start();

}

}

H2O:

import java.util.\*;  
import java.util.concurrent.\*;  
public class ThreadH2OSema {      
    private final Semaphore hCalled = new Semaphore(0);  
    private final Semaphore hUsed = new Semaphore(0);  
    private final Semaphore oCalled = new Semaphore(0);      
    private final Semaphore oUsed = new Semaphore(0);      
  
    public void H() {  
        try {  
            hCalled.release();  
            System.out.println("One H created");  
            hUsed.acquire();  
        }  
        catch(InterruptedException e) {  
            System.out.println("Interrupted..");  
        }  
    }      
    public void O() {  
        try {  
            oCalled.release();  
            System.out.println("One O created");  
            oUsed.acquire();  
        }  
        catch(InterruptedException e) {  
            System.out.println("Interrupted..");  
        }  
    }  
  
    public void H2O() {  
        int i = 1;  
        while(true) {  
            try {              
                hCalled.acquire(2);  
                oCalled.acquire(1);                  
                hUsed.release(2);  
                oUsed.release(1);  
                System.out.println(i + " H2O created..");  
                i++;  
            }  
            catch(InterruptedException e) {  
                System.out.println("Interrupted..");  
                break;  
            }  
        }  
    }  
  
    private static class Caller extends Thread {  
        private ThreadH2OSema h2o;  
        public Caller(ThreadH2OSema h2o) {  
            this.h2o = h2o;   
        }  
        public void run() {  
            Random r = new Random();  
            // Call H() or O() randomly  
            if (r.nextBoolean()) h2o.H();  
            else h2o.O();  
        }  
    }  
      
    private static class Creator extends Thread {  
        private ThreadH2OSema h2o;  
        public Creator(ThreadH2OSema h2o) {  
            this.h2o = h2o;   
        }  
        public void run() {  
            h2o.H2O();  
        }  
    }      
  
    public static void main(String[] args) {  
        ThreadH2OSema h2o = new ThreadH2OSema();  
        Thread creator = new Creator(h2o);  
        creator.start();  
          
        Thread[] threads = new Thread[3];   
        for (int i = 0; i < threads.length; i++) {  
            threads[i] = new Caller(h2o);  
            threads[i].start();  
        }                  
        try {  
            for (int i = 0; i < threads.length; i++) {  
                threads[i].join();  
            }      
        }  
        catch(InterruptedException e) {  
            System.out.println("Interrupted..");      
        }          
    }  
}

Blocking Queue:

public class BlockingQueue {  
  
  private List queue = new LinkedList();  
  private int  limit = 10;  
  
  public BlockingQueue(int limit){  
    this.limit = limit;  
  }  
  
  
  public synchronized void enqueue(Object item)  
  throws InterruptedException  {  
    while(this.queue.size() == this.limit) {  
      wait();  
    }  
    if(this.queue.size() == 0) {  
      notifyAll();  
    }  
    this.queue.add(item);  
  }  
  
public synchronized Object dequeue()  
  throws InterruptedException{  
    while(this.queue.size() == 0){  
      wait();  
    }  
    if(this.queue.size() == this.limit){  
      notifyAll();  
    }  
  
    return this.queue.remove(0);  
}