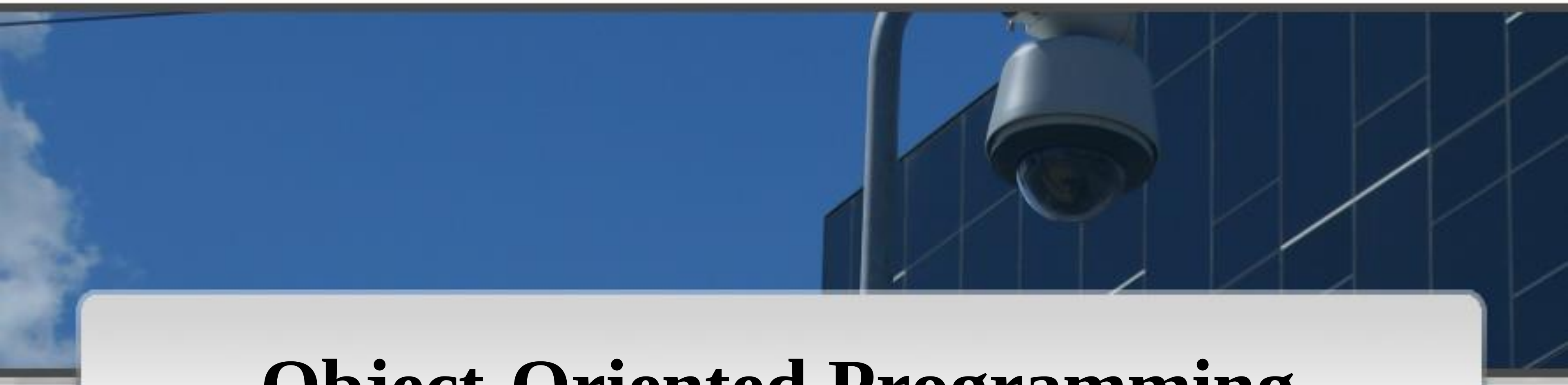


Object-Oriented Programming

Interfaces, Inner classes, Threads and Multithreading

Lab 11

COMPUTER SECURITY LABORATORY

HANYANG UNIVERSITY

Content

- **Interfaces**
  - What are interfaces
  - Ordered
  - Derived
  - Comparable
- **Inner classes**
  - Static Inner Classes
  - Public Inner Classes
  - Nested Inner Classes
  - Inheriting Inner Classes
  - Anonymous Classes
- **Threads and Multithreading**
  - Threads in Java
  - Thread Lifecycle
  - Synchronization
  - Thread Priorities

Interfaces

- **What is an Interface?**
  - An interface is a collection of “abstract” methods.
  - An interface is a template that a class must adhere to.
  - A class implements an interface, thereby inheriting the abstract methods of the interface.
- **Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.**
- **The syntax for defining an interface is similar to that of defining a class**
  - Except the word interface is used in place of class

Interfaces

- **An interface is similar to a class in the following ways:**
  - An interface can contain any number of methods.
  - An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
  - The byte code of an interface appears in a **.class** file.

Interfaces

- **An interface is different from a class in several ways:**
  - You cannot instantiate an interface.
  - An interface does not contain any constructors.
  - All of the methods in an interface are abstract.
  - An interface cannot contain instance variables. The only fields that can appear in an interface must be declared both static and final.
  - An interface is not extended by a class; it is implemented by a class.
  - An interface can extend multiple interfaces.

Interfaces

- **An interface and all of its method headings should be declared public**
  - They cannot be given **private**, **protected**, or **package** access
- **When a class implements an interface, it must make all the methods in the interface public**
- **Because an interface is a type, a method may be written with a parameter of an interface type**
  - That parameter will accept as an argument any class that implements the interface.

Interfaces

- **How do I define an Interface…**

```
public interface InterfaceName{
        public returnType method1();
        public returnType method2();
        public returnType method3();
}

public class SomeClass implements InterfaceName{
        //implement all the methods
}
```

☐ Keyword interface

☐ Methods are not defined. Note you do not use the keyword abstract

☐ The interface is implemented by a class

☐ It is possible for this class to be abstract.

The Ordered Interface

Display 13.1 The Ordered Interface

Do not forget the semicolons at the end of the method headings.

```
public interface Ordered
{
    public boolean precedes(Object other);

    /**
     For objects of the class o1 and o2,
     o1.follows(o2) == o2.preceded(o1).
    */
    public boolean follows(Object other);
}
```

Neither the compiler nor the run-time system will do anything to ensure that this comment is satisfied. It is only advisory to the programmer implementing the interface.

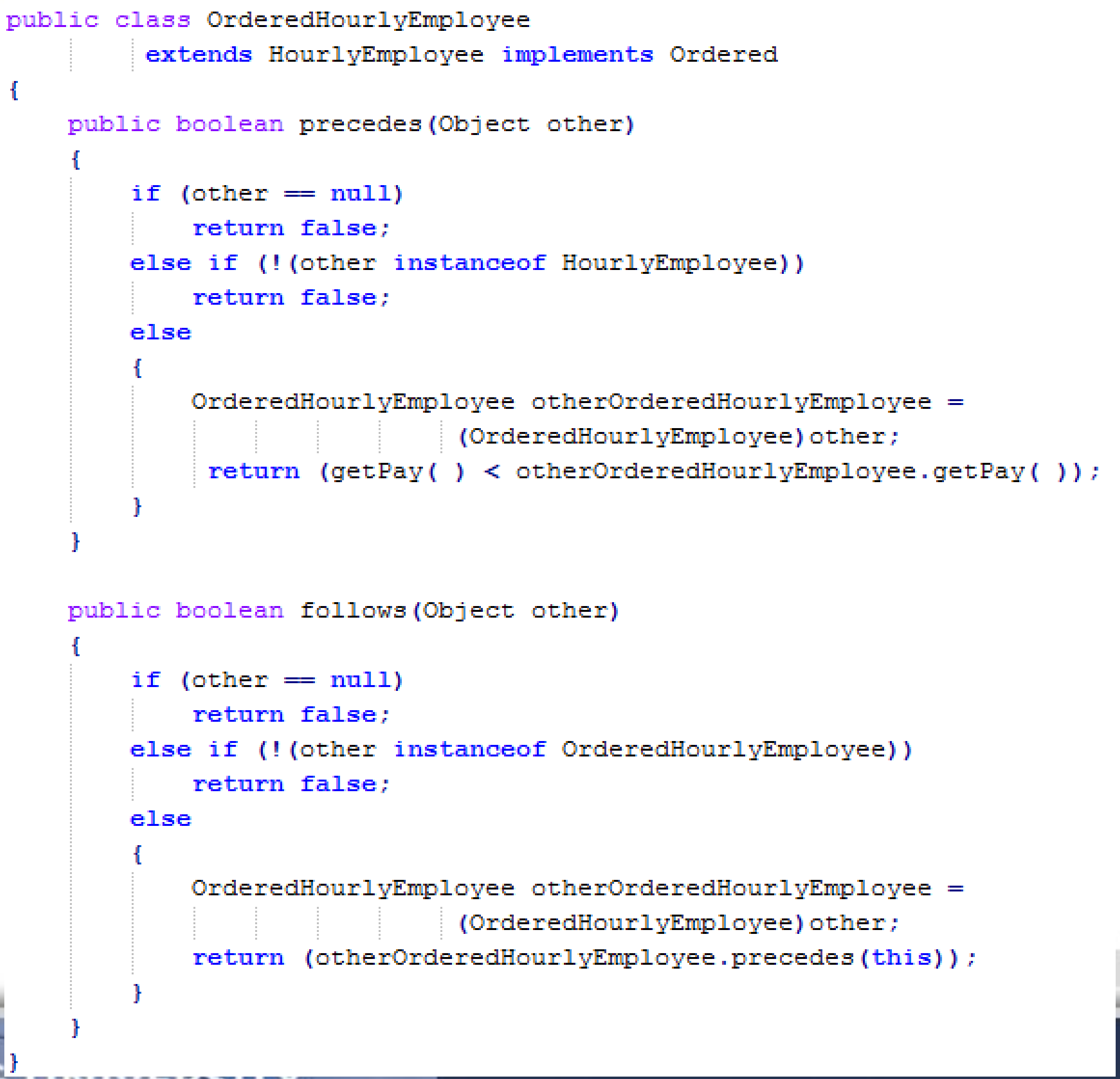
Interfaces

- To *implement an interface*, a concrete class must do two things:
  1. It must include the phrase `implements Interface_Name` at the start of the class definition
     - If more than one interface is implemented, each is listed, separated by commas
  2. The class must implement *all* the method headings listed in the definition(s) of the interface(s)

```
public class SomeClass implements Ordered, Comparable{
```

☐ *Multiple interfaces may be implemented*

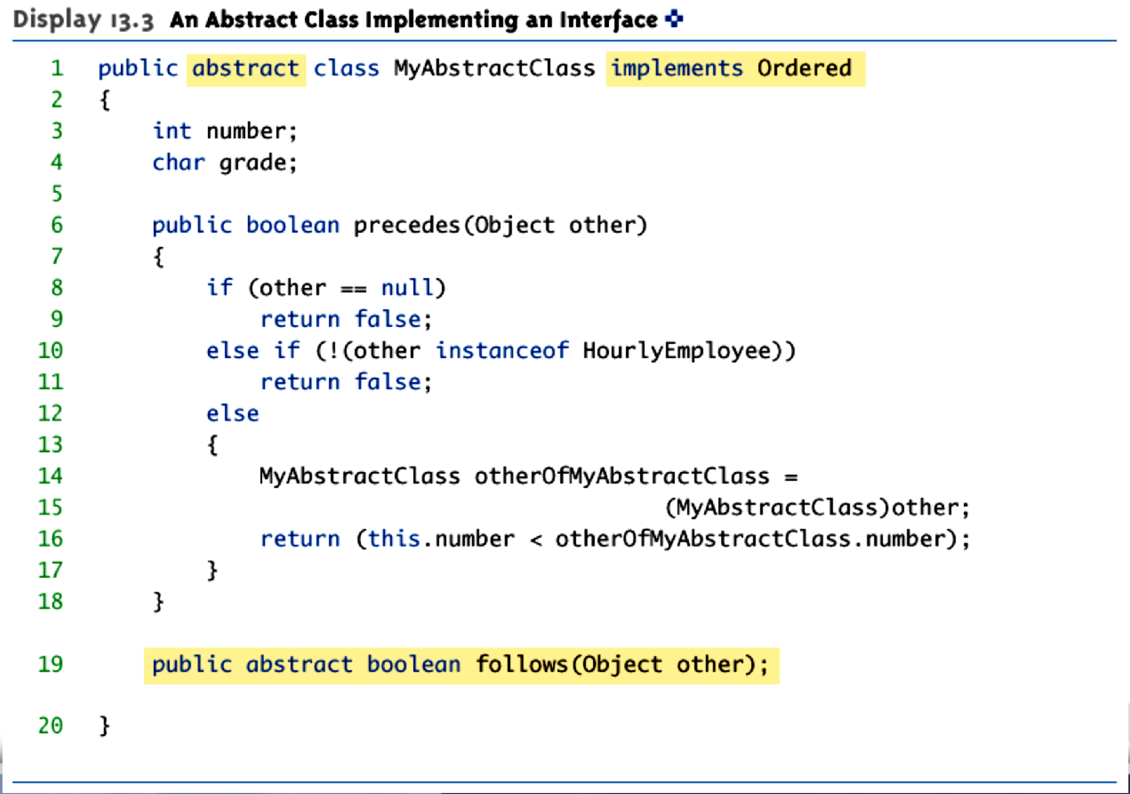
The Ordered Interface



```
public class OrderedHourlyEmployee
          extends HourlyEmployee implements Ordered
{
    public boolean precedes(Object other)
    {
        if (other == null)
            return false;
        else if (!(other instanceof HourlyEmployee))
            return false;
        else
        {
            OrderedHourlyEmployee otherOrderedHourlyEmployee =
                                (OrderedHourlyEmployee)other;
             return (getPay( ) < otherOrderedHourlyEmployee.getPay( ));
        }
    }

    public boolean follows(Object other)
    {
        if (other == null)
            return false;
        else if (!(other instanceof OrderedHourlyEmployee))
            return false;
        else
        {
            OrderedHourlyEmployee otherOrderedHourlyEmployee =
                                (OrderedHourlyEmployee)other;
            return (otherOrderedHourlyEmployee.precedes(this));
        }
    }
}
```

Interfaces

- **An abstract class may *implement an interface*, However:**
  1. It is not imperative that all methods be implemented.
  2. Unimplemented methods must be marked as abstract;



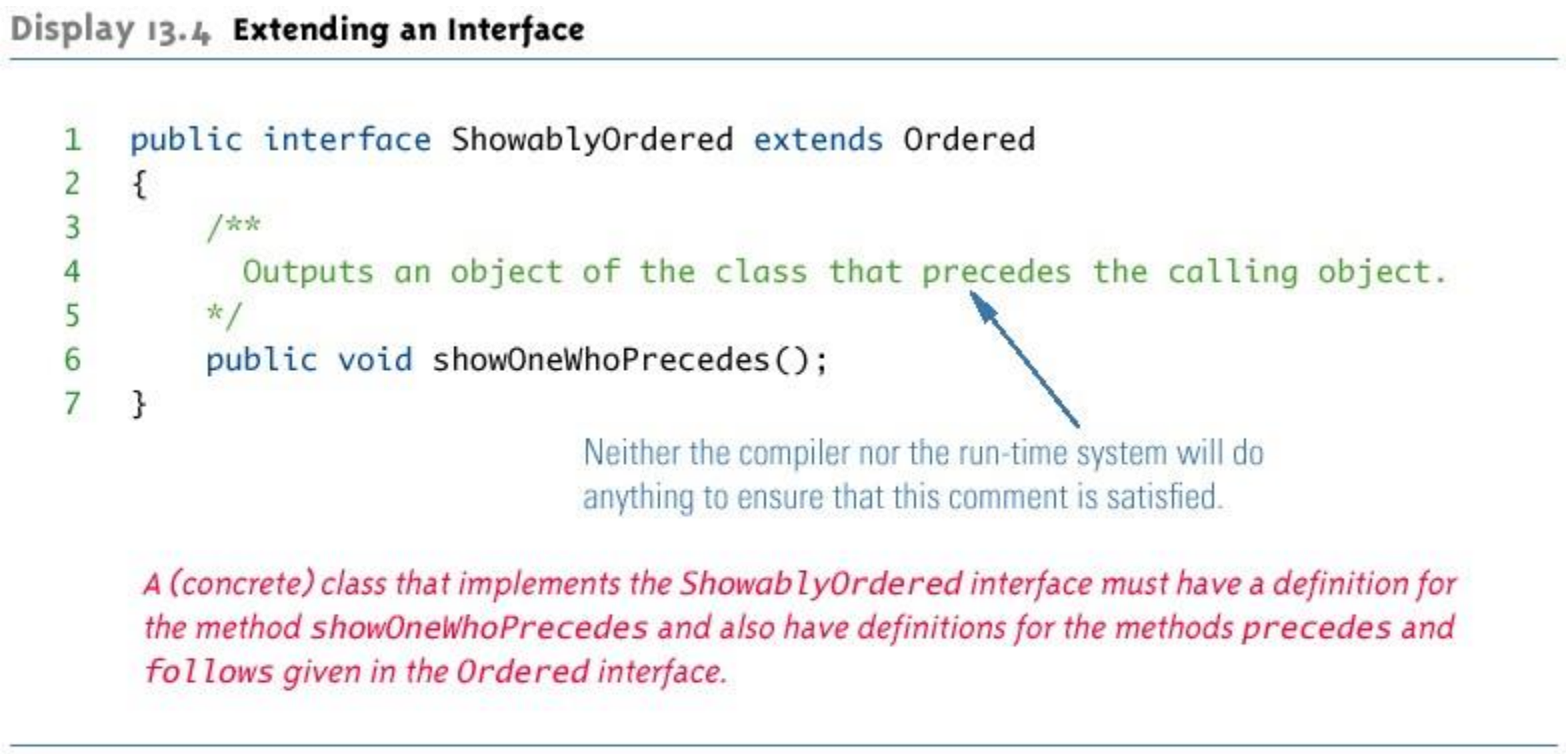
Display 13.3 An Abstract Class Implementing an Interface

```java
public abstract class MyAbstractClass implements Ordered
{
    int number;
    char grade;

    public boolean precedes(Object other)
    {
        if (other == null)
            return false;
        else if (!(other instanceof HourlyEmployee))
            return false;
        else
        {
            MyAbstractClass otherOfMyAbstractClass =
                                          (MyAbstractClass)other;
            return (this.number < otherOfMyAbstractClass.number);
        }
    }

    public abstract boolean follows(Object other);

}
```

Derived Interfaces

- **Like classes, an interface may be derived from a base interface**
  - This is called *extending* the interface
  - The derived interface must include the phrase
    - **`extends`** ***`BaseInterfaceName`***
- **A concrete class that implements a derived interface must have definitions for any methods in the derived interface as well as any methods in the base interface**

Derived Interfaces



Display 13.4 **Extending an Interface**

```
public interface ShowablyOrdered extends Ordered
{
    /**
      Outputs an object of the class that precedes the calling object.
    */
    public void showOneWhoPrecedes();
}
```

Neither the compiler nor the run-time system will do anything to ensure that this comment is satisfied.

A (concrete) class that implements the `ShowablyOrdered` interface must have a definition for the method `showOneWhoPrecedes` and also have definitions for the methods `precedes` and `follows` given in the `Ordered` interface.

Comparable interface

- Can be used to provide a single sorting method.
- The `Comparable` interface is in the `java.lang` package, and so is automatically available to any program
- It has only the following method heading that must be implemented:

```
public int compareTo(Object other);
```

☐ *compareTo allows you to compare objects of the same type based on any criteria you deci de.*

Comparable interface

- **int compareTo(Object other);**

Must return

- A negative number if the calling object "comes before" the parameter other
- A zero if the calling object "equals" the parameter other
- A positive number if the calling object "comes after" the parameter other

- **If the parameter `other` is not of the same type as the class being defined, then a `ClassCastException` should be thrown**

-1	*this* < parameter
0	*this* = parameter
1	*this* > parameter

Writing Comparable

- The following shows an example of the compareTo method

```
public int compareTo(Object obj){
            if (obj == null) throw new NullPointerException("Object is null");
            if (!this.getClass().equals(obj.getClass())) throw new
                        ClassCastException("Object not of the same type");
            Employee toCompare = (Employee) obj;
            if (this.salary > toCompare.salary) return -1;
            if (this.salary == toCompare.salary) return 0;
            else return 1;
}
```

Using Comparable

- Sort
- Arrays class in java contain a static sort() method that can be used to sort arrays.
- To sort an array it must contain only Comparable objects.
- To be able to sort objects I must be able to compare them to each other.
- We know we can compare objects if they conform to the interface Comparable.

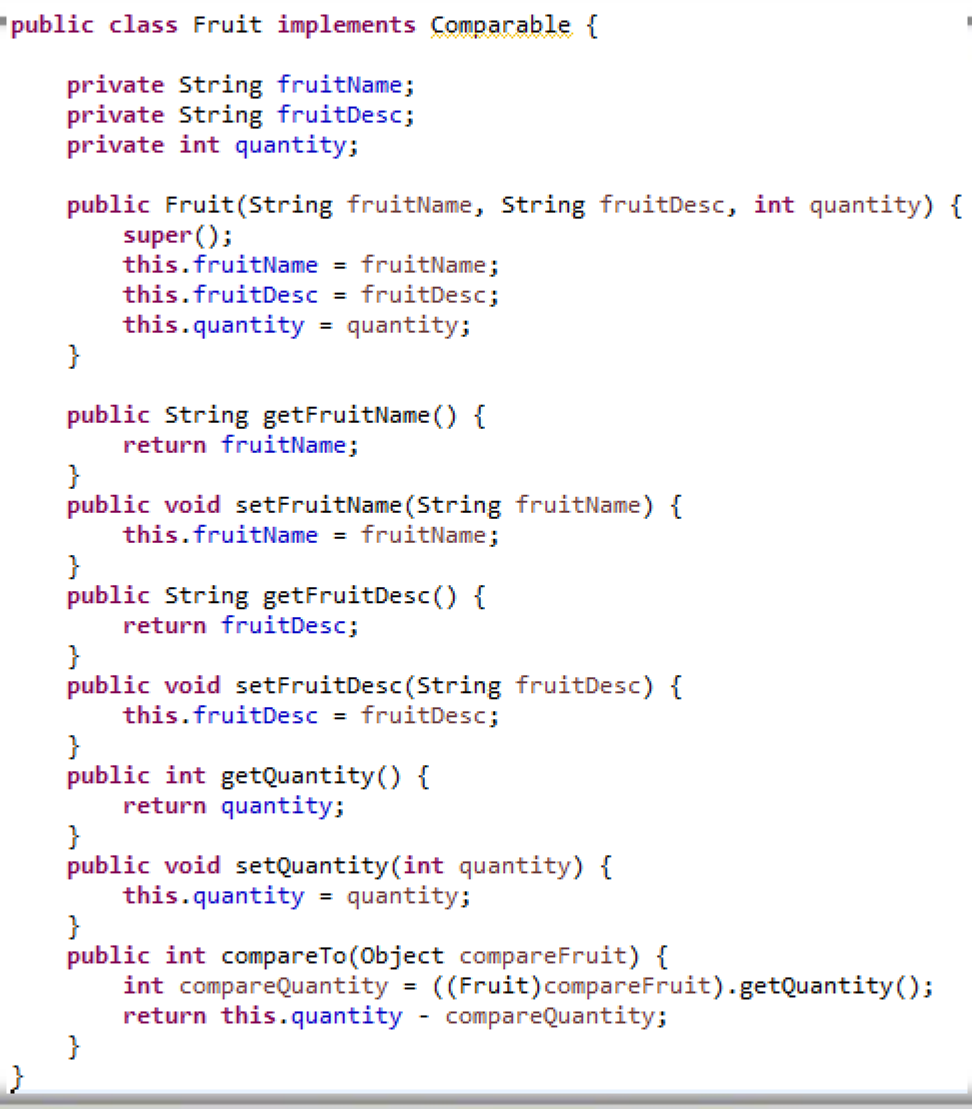
Comparable interface

```
public class Fruit{

        private String fruitName;
        private String fruitDesc;
        private int quantity;

        public Fruit(String fruitName, String fruitDesc, int quantity) {
                super();
                this.fruitName = fruitName;
                this.fruitDesc = fruitDesc;
                this.quantity = quantity;
        }

        public String getFruitName() {
                return fruitName;
        }
        public void setFruitName(String fruitName) {
                this.fruitName = fruitName;
        }
        public String getFruitDesc() {
                return fruitDesc;
        }
        public void setFruitDesc(String fruitDesc) {
                this.fruitDesc = fruitDesc;
        }
        public int getQuantity() {
                return quantity;
        }
        public void setQuantity(int quantity) {
                this.quantity = quantity;
        }
}
```

Comparable interface

```
package com.mkyong.common.action;

import java.util.Arrays;

public class SortFruitObject{

        public static void main(String args[]){

                Fruit[] fruits = new Fruit[4];

                Fruit pineappale = new Fruit("Pineapple", "Pineapple description",70);
                Fruit apple = new Fruit("Apple", "Apple description",100);
                Fruit orange = new Fruit("Orange", "Orange description",80);
                Fruit banana = new Fruit("Banana", "Banana description",90);

                fruits[0]=pineappale;
                fruits[1]=apple;
                fruits[2]=orange;
                fruits[3]=banana;

                Arrays.sort(fruits);

                int i=0;
                for(Fruit temp: fruits){
                   System.out.println("fruits " + ++i + " : " + temp.getFruitName() +
                        ", Quantity : " + temp.getQuantity());
                }

        }
}
```

Comparable interface



```
public class Fruit implements Comparable {

    private String fruitName;
    private String fruitDesc;
    private int quantity;

    public Fruit(String fruitName, String fruitDesc, int quantity) {
        super();
        this.fruitName = fruitName;
        this.fruitDesc = fruitDesc;
        this.quantity = quantity;
    }

    public String getFruitName() {
        return fruitName;
    }
    public void setFruitName(String fruitName) {
        this.fruitName = fruitName;
    }
    public String getFruitDesc() {
        return fruitDesc;
    }
    public void setFruitDesc(String fruitDesc) {
        this.fruitDesc = fruitDesc;
    }
    public int getQuantity() {
        return quantity;
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
    public int compareTo(Object compareFruit) {
        int compareQuantity = ((Fruit)compareFruit).getQuantity();
        return this.quantity - compareQuantity;
    }
}
```

☐ return this.quantity - compareQuantity

```
--------- Output ---------
Fruits 1 : Pineapple, Quantity : 70
Fruits 2 : Orange, Quantity : 80
Fruits 3 : Banana, Quantity : 90
Fruits 4 : Apple, Quantity : 100
```

☐ return compareQuantity - this.quantity

```
--------- Output ---------
Fruits 1 : Apple, Quantity : 100
Fruits 2 : Banana, Quantity : 90
Fruits 3 : Orange, Quantity : 80
Fruits 4 : Pineapple, Quantity : 70
```

Constants in Interfaces

- **Java allows for defining constants in interface**
- **To define a constant it must be marked using**
  - public static final
- **When a class implements the interface it automatically gets the constants that are declared.**

Inconsistencies in Interfaces

- **Two form of inconsistencies may arise with interfaces**
  - Inconsistent constants
  - Inconsistent methods
- **If a class implements two interfaces with the same constant defined but different values.**
- **If a class implements two interfaces with the same method header defined. {return type may be different}**

Self-Test

- **제공된 InterfaceSelfTest를 사용하여 진행한다.**
- **위 프로젝트에서 추가/수정이 필요한 부분은 아래와 같다.**
  - MyInterfaces interface를 구현(implements)한 MyCustom Class는 추상(abstract) 클래스가 아니다. 따라서, 모든 메소드가 아래 조건에 맞게 구현되어야 한다.
    - public void move(String key) 메소드는 key의 값에 따라 다음 값을 갖는다.
      - w : 1 / s : 2 / a : 3 / d : 4 / 그외 : 5
    - public void attack(String key) 메소드는 key의 값에 따라 다음 값을 갖는다.
      - spacebar : (Boolean)true / 그외 : (Boolean)false
    - public void sortItem(Item[] itemList) 메소드는 호출되면 itemList를 정렬한다. Arrays를 사용하여 정렬하도록 구현한다.
  - move_type 변수와 isAttack을 반환하는 public 메소드인 getMoveType()과 getIsAttack()을 구현한다.
  - Item Class에 compareTo를 작성한다.
    (quantity를 기준으로 오름차순 정렬이 되야 한다.)

Self-Test

- 프로그램이 정상적으로 동작한다면 아래의 결과가 Console에 표시된다.

```
Move Type : 0
Attack : false
ItemList : 5
------------------------------------
Move Type : 2
Attack : true
ItemList
Item[0] : HP (50)
Item[1] : MP (30)
Item[2] : Food (100)
Item[3] : Key (10)
Item[4] : Gold (1)
------------------------------------
ItemList
Item[0] : Gold (1)
Item[1] : Key (10)
Item[2] : MP (30)
Item[3] : HP (50)
Item[4] : Food (100)
------------------------------------
```

Inner Classes

- **Inner Classes are classes defined within other classes**
- **The class that includes the inner class is called the outer class**
- **An inner class definition is a member of the outer class in the same way that the instance variables and methods of the outer class are members**
  - An inner class is local to the outer class definition
- **Using an inner class as a helping class is one of the most useful applications of inner classes**

Class with an Inner Class

Display 13.9 Class with an Inner Class *(Part 1 of 2)*

```
public class BankAccount
{
    private class Money
    {
        private long dollars;
        private int cents;

        public Money(String stringAmount)
        {
            abortOnNull(stringAmount);
            int length = stringAmount.length();
            dollars = Long.parseLong(
                             stringAmount.substring(0, length - 3));
            cents = Integer.parseInt(
                             stringAmount.substring(length - 2, length));
        }

        public String getAmount()
        {
            if (cents > 9)
               return (dollars + "." + cents);
            else
               return (dollars + ".0" + cents);
        }
```

The modifier `private` in this line should not be changed to `public`.
However, the modifiers `public` and `private` inside the inner class `Money` can be changed to anything else and it would have no effect on the class `BankAccount`.

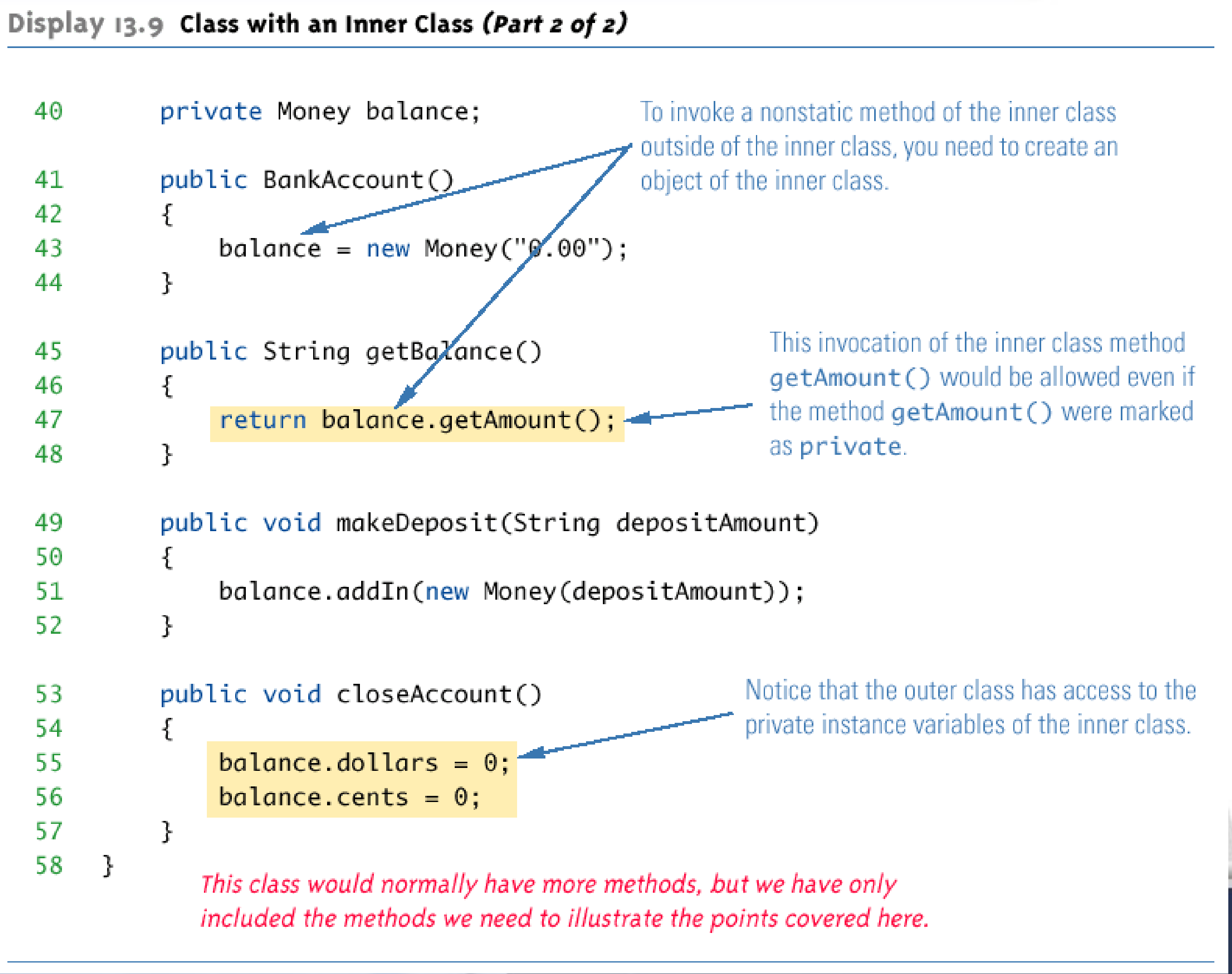
Class with an Inner Class

Display 13.9 Class with an Inner Class *(Part 1 of 2)* (continued)

```
        public void addIn(Money secondAmount)
        {
            abortOnNull(secondAmount);
            int newCents = (cents + secondAmount.cents)%100;
            long carry = (cents + secondAmount.cents)/100;
            cents = newCents;
            dollars = dollars + secondAmount.dollars + carry;
        }

       private void abortOnNull(Object o)
       {
           if (o == null)
           {
               System.out.println("Unexpected null argument.");
               System.exit(0);
           }
       }
}
```

The definition of the inner class ends here, but the definition of the outer class continues in Part 2 of this display.

Class with an Inner Class



Display 13.9 Class with an Inner Class *(Part 2 of 2)*

```
    private Money balance;

    public BankAccount()
    {
        balance = new Money("0.00");
    }

    public String getBalance()
    {
        return balance.getAmount();
    }

    public void makeDeposit(String depositAmount)
    {
        balance.addIn(new Money(depositAmount));
    }

    public void closeAccount()
    {
        balance.dollars = 0;
        balance.cents = 0;
    }
}
```

To invoke a nonstatic method of the inner class outside of the inner class, you need to create an object of the inner class.

This invocation of the inner class method `getAmount()` would be allowed even if the method `getAmount()` were marked as `private`.

Notice that the outer class has access to the private instance variables of the inner class.

This class would normally have more methods, but we have only included the methods we need to illustrate the points covered here.

Inner Class

```java
public class DataStructure {
    // create an array
    private final static int SIZE = 15;
    private int[] arrayOfInts = new int[SIZE];

    public DataStructure() {
        // fill the array with ascending integer values
        for (int i = 0; i < SIZE; i++) {
            arrayOfInts[i] = i;
        }
    }

    public void printEven() {
        // print out values of even indices of the array
        InnerEvenIterator iterator = this.new InnerEvenIterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.getNext() + " ");
        }
    }

    // inner class implements the Iterator pattern
    private class InnerEvenIterator {
        // start stepping through the array from the beginning
        private int next = 0;

        public boolean hasNext() {
            // check if a current element is the last in the array
            return (next <= SIZE - 1);
        }

        public int getNext() {
            // record a value of an even index of the array
            int retValue = arrayOfInts[next];
            //get the next even element
            next += 2;
            return retValue;
        }
    }

    public static void main(String s[]) {
        // fill the array with integer values and print out only
        // values of even indices
        DataStructure ds = new DataStructure();
        ds.printEven();
    }
}
```

Output

```
0
2
4
6
8
10
12
14
```

Static Inner Classes

- **A normal inner class has a connection between its objects and the outer class object that created the inner class object**
  - This allows an inner class definition to reference an instance variable, or invoke a method of the outer class
- **There are certain situations, however, when an inner class must be static**
  - If an object of the inner class is created within a static method of the outer class. {Builder design pattern}
  - If the inner class must have static members

☐ Instance variables of the outer class cannot be referenced.
☐ Non-static methods of the outer class cannot be i nvoked.

Public Inner Classes

- **If an inner class is marked public, then it can be used outside of the outer class**
  - Create an object using an instance of the outer class {non-static}
  - Use the Outer class' name to access the inner class {static}

```
BankAccount account = new BankAccount();
BankAccount.Money amount = account.new Money("41.99");
```

```
OuterClass.InnerClass innerObject = new OuterClass.InnerClass();
```

Nested Inner Classes

- It is possible to nest an inner class within another inner class
- This however makes the notation for accessing the nested inner class longer.

```
public class A{
   ---
   public class B{
      ---
      public class C{

      }

   }

}
```

```
A aObject = new A();
A.B bObject = aObject.new B();
A.B.C cObject = bObject.new C();
```

Inheriting Inner Classes

- **It is possible to inherit an outer class that has an inner class**
- **If you derive (inherit) an outer class then the inner class is automatically inherited.**
  - The inner class cannot be overridden.

Anonymous Classes

- **If an object is to be created, but there is no need to name the object's class, then an anonymous class definition can be used**
  - The class definition is embedded inside the expression with the **new** operator
- **Anonymous classes are sometimes used when they are to be assigned to a variable of another type**

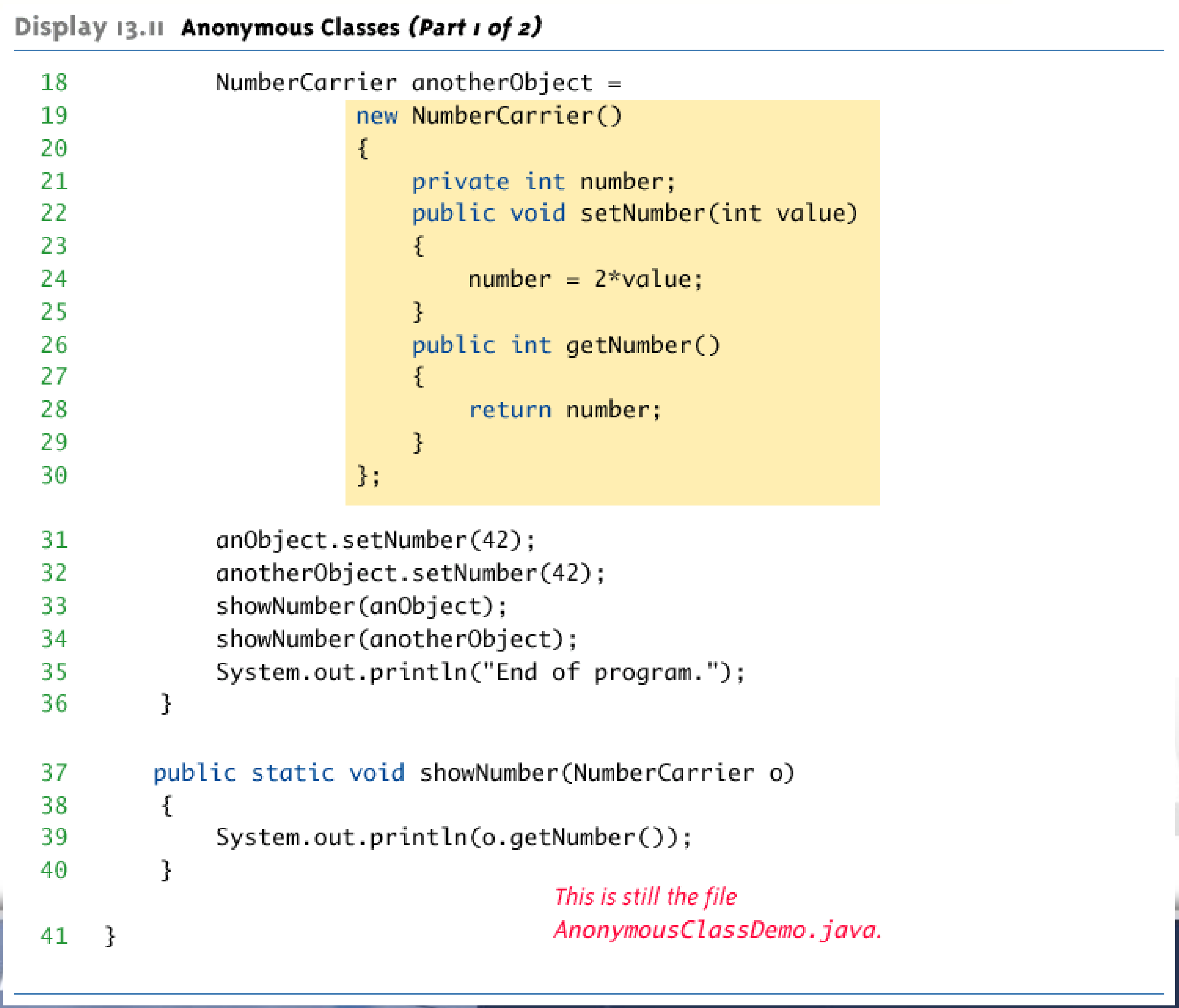
Anonymous Classes

Display 13.11 Anonymous Classes (*Part 1 of 2*)

This is just a toy example to demonstrate the Java syntax for anonymous classes.

```
public class AnonymousClassDemo
{
    public static void main(String[] args)
    {
        NumberCarrier anObject =
                      new NumberCarrier()
                      {
                          private int number;
                          public void setNumber(int value)
                          {
                              number = value;
                          }
                          public int getNumber()
                          {
                             return number;
                          }
                       };
```

Anonymous Classes



Display 13.11 Anonymous Classes *(Part 1 of 2)*

```
    NumberCarrier anotherObject =
             new NumberCarrier()
             {
               private int number;
               public void setNumber(int value)
               {
                 number = 2*value;
               }
               public int getNumber()
               {
                 return number;
               }
             };

    anObject.setNumber(42);
    anotherObject.setNumber(42);
    showNumber(anObject);
    showNumber(anotherObject);
    System.out.println("End of program.");
  }

  public static void showNumber(NumberCarrier o)
  {
    System.out.println(o.getNumber());
  }

}
```

This is still the file AnonymousClassDemo.java.

Anonymous Classes

Display 13.11 **Anonymous Classes *(Part 2 of 2)***

SAMPLE DIALOGUE

```
42
84
End of program.
```

```java
public interface NumberCarrier
{
    public void setNumber(int value);
    public int getNumber();
}
```

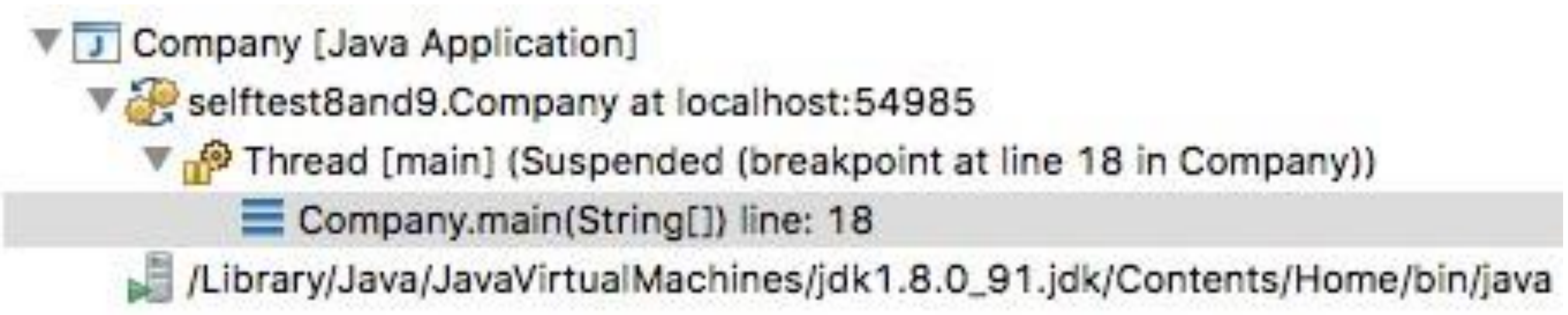
This is the file NumberCarrier.java.

Threads & Multithreading

- **A thread is a separate computation process.**
  - Threads can be thought of as computations that execute in parallel.
  - A thread is a single sequential flow of control within a program.
  - Thread does not have its own address space but uses the memory and other resources of the process in which it executes.
  - There may be several threads in one process
- **Characteristics of threads**
  - Threads are lightweight processes as the overhead of switching between threads is less
  - The can be easily spawned

Threads & Multithreading

- **We have experienced threads**
  - The Java Virtual Machine spawns a thread when your program is run called the Main Thread



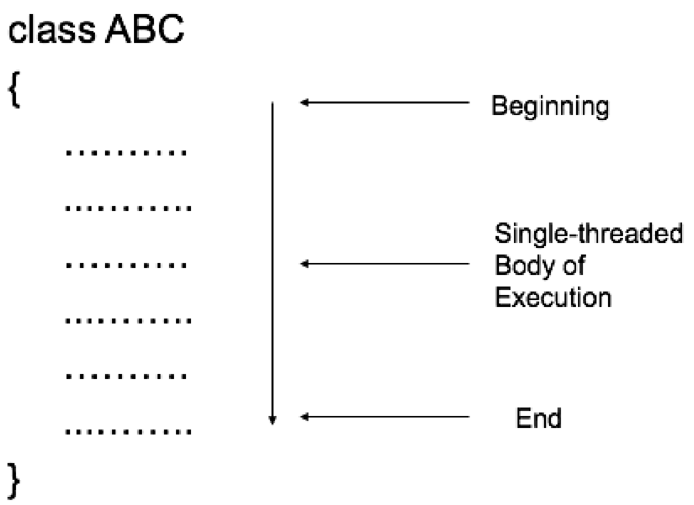
```
▼ Company [Java Application]
  ▼ selftest8and9.Company at localhost:54985
    ▼ Thread [main] (Suspended (breakpoint at line 18 in Company))
        Company.main(String[]) line: 18
  /Library/Java/JavaVirtualMachines/jdk1.8.0_91.jdk/Contents/Home/bin/java
```

Threads & Multithreading

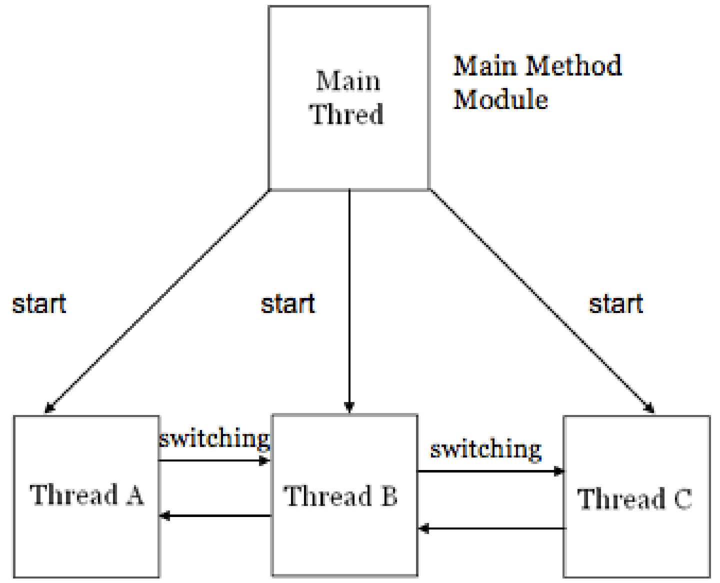
- **Multithreading is a programming concept where a program (process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel**
- **Why do we need threads?**
  - To enhance parallel processing
  - To increase response to the user
  - To utilize the idle time of the CPU
  - Prioritize your work depending on priority
- **E.g.**
  - Video Games
  - Web Server
    - The web server listens for request and serves it

Threads & Multithreading

Single Thread



MultiThread



Threads in Java

- **Java contains a Thread class and a Runnable interface.**
  - There are 2 ways we can use threads
    - extending the class **Thread**.

      public class MyThreadClass extends Thread{.. }
    - Implementing **Runnable** interface.

      public class MyThreadClass implements Runnable{.. }

Note: It is usually more preferred to implement the Runnable Interface so that we can extend properties from other classes

- **The Thread class and Runnable interface contains the method run().**
  - The **public void run()** method acts like the main method of a traditional sequential program.
  - We must **override** or **implement** the **run()** method so that our thread can perform the actions we need.

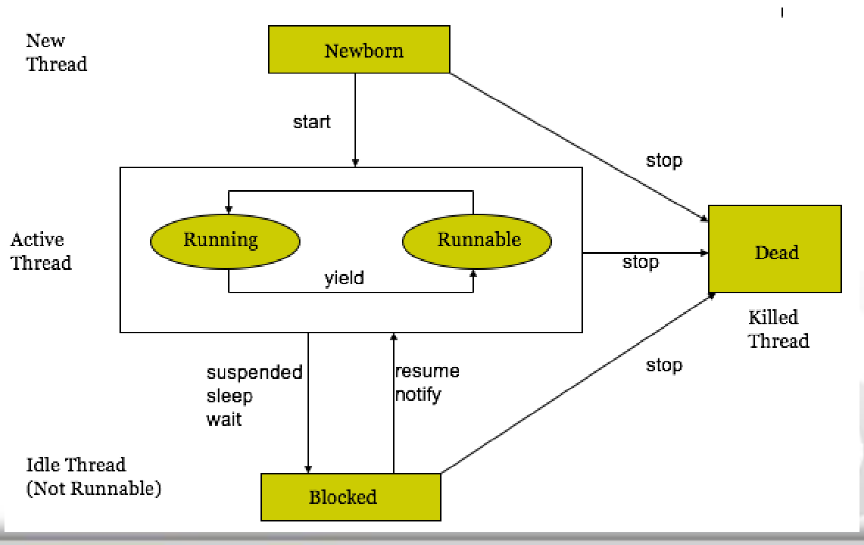
Thread Lifecycle

When threads are created they may go through the following states.

- **New** (Newborn)
- **Runnable**
- **Running**
- **Blocked**
- **Terminated** (Dead)

The states of threads may be controlled by these methods.

- **start()**
- **yield()**
- **suspend(), sleep(), wait()**
- **resume(), notify()**
- **stop()**



Thread Lifecycle

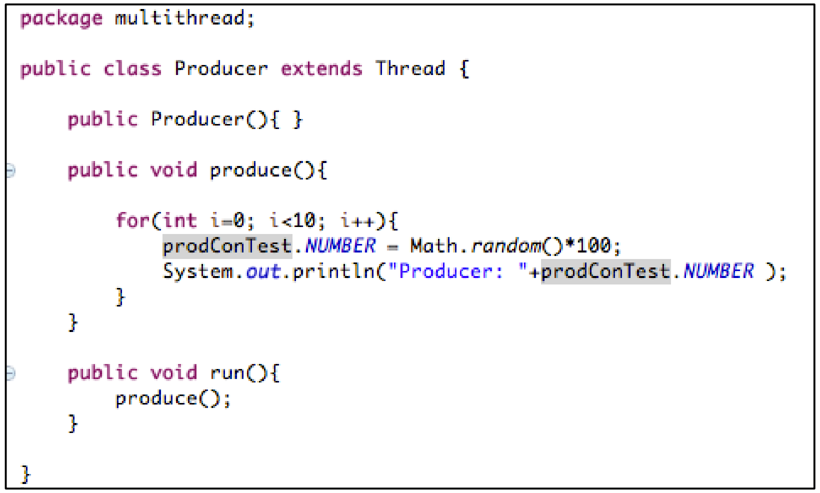
- **Example**

```
class mythread implements Runnable{
        public void run(){
                System.out.println("Thread Started");
        }
}

class mainclass {
        public static void main(String args[]){
                Thread  t = new Thread(new mythread()); // This is the way to instantiate a
                                                  thread implementing runnable interface
                t.start(); // starts the thread by running the run method
                }
}
```

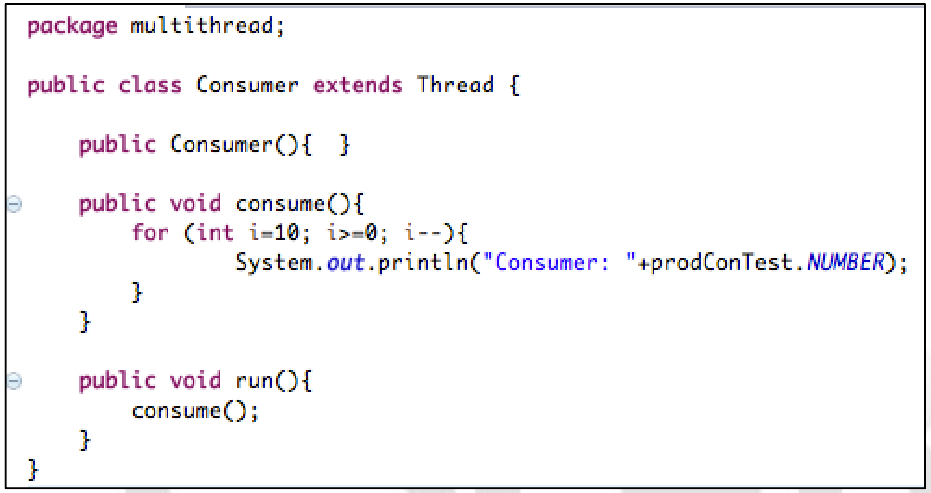
- **Calling t.run() does not start a thread, it is just a simple method call.**
- **Creating an object does not create a thread, calling start() method creates the thread.**

Producer Consumer

- **Let us examine a simple producer consumer example**
- **In this example**
  - We have a producer thread that creates a random number.
  - We also have a consumer thread that consumes the random number.



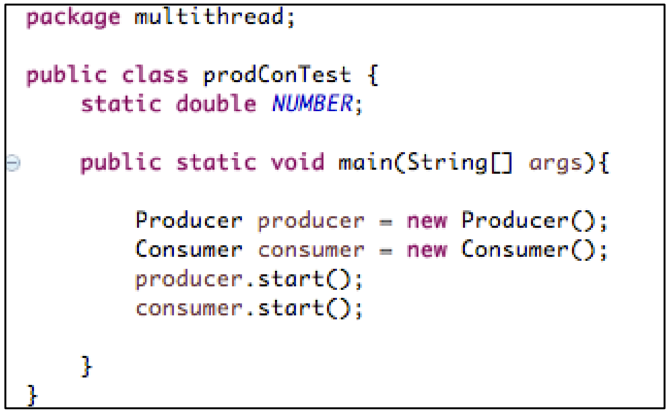
```java
package multithread;

public class Producer extends Thread {

    public Producer(){ }

    public void produce(){

        for(int i=0; i<10; i++){
            prodConTest.NUMBER = Math.random()*100;
            System.out.println("Producer: "+prodConTest.NUMBER );
        }
    }

    public void run(){
        produce();
    }

}
```



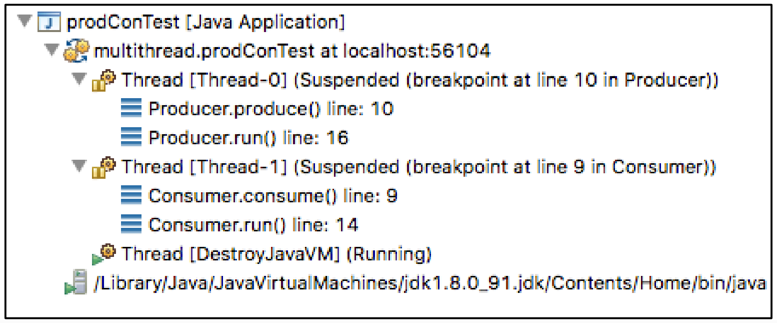
```java
package multithread;

public class Consumer extends Thread {

    public Consumer(){  }

    public void consume(){
        for (int i=10; i>=0; i--){
                System.out.println("Consumer: "+prodConTest.NUMBER);
        }
    }

    public void run(){
        consume();
    }
}
```

Producer Consumer

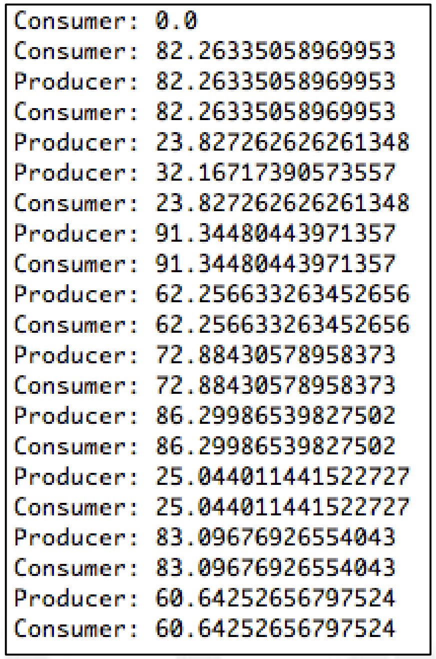
- **The main method of the producer consumer creates two threads.**
  - One producer thread
  - One consumer thread



```
package multithread;

public class prodConTest {
    static double NUMBER;

    public static void main(String[] args){

        Producer producer = new Producer();
        Consumer consumer = new Consumer();
        producer.start();
        consumer.start();

    }
}
```



```
▼ prodConTest [Java Application]
  ▼ multithread.prodConTest at localhost:56104
    ▼ Thread [Thread-0] (Suspended (breakpoint at line 10 in Producer))
        Producer.produce() line: 10
        Producer.run() line: 16
    ▼ Thread [Thread-1] (Suspended (breakpoint at line 9 in Consumer))
        Consumer.consume() line: 9
        Consumer.run() line: 14
      Thread [DestroyJavaVM] (Running)
  /Library/Java/JavaVirtualMachines/jdk1.8.0_91.jdk/Contents/Home/bin/java
```



```
Consumer: 0.0
Consumer: 82.26335058969953
Producer: 82.26335058969953
Consumer: 82.26335058969953
Producer: 23.827262626261348
Producer: 32.16717390573557
Consumer: 23.827262626261348
Producer: 91.34480443971357
Consumer: 91.34480443971357
Producer: 62.256633263452656
Consumer: 62.256633263452656
Producer: 72.88430578958373
Consumer: 72.88430578958373
Producer: 86.29986539827502
Consumer: 86.29986539827502
Producer: 25.044011441522727
Consumer: 25.044011441522727
Producer: 83.09676926554043
Consumer: 83.09676926554043
Producer: 60.64252656797524
Consumer: 60.64252656797524
```

Something's fishy with our results…

Synchronization

- **Multithreading introduces an asynchronous behavior to your programs.**
- **We must force synchronicity if needed.**
- **This is done by using the keyword synchronized**

- **Synchronization in java is the capability of controlling the access of a shared resource by multiple threads.**

- **Why do we use synchronization?**
  - To prevent thread interference
  - To prevent consistency problems
  - To prevent data corruption

Synchronization

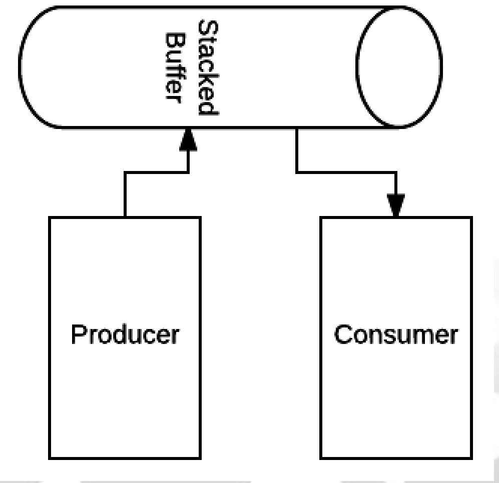
- **How it works.**
  - When a thread begins to execute a synchronized method it automatically acquires a lock on that object.
  - The lock is relinquished when the method ends.
  - Only one thread can have the lock at a particular time
  - Therefore only one thread can execute the synchronized instance method of the same object at a particular time.

  - Synchronization allows only one thread to perform an operation on a object at a time.
  - If multiple threads require an access to an object, synchronization helps in maintaining consistency.

Synchronized Producer Consumer

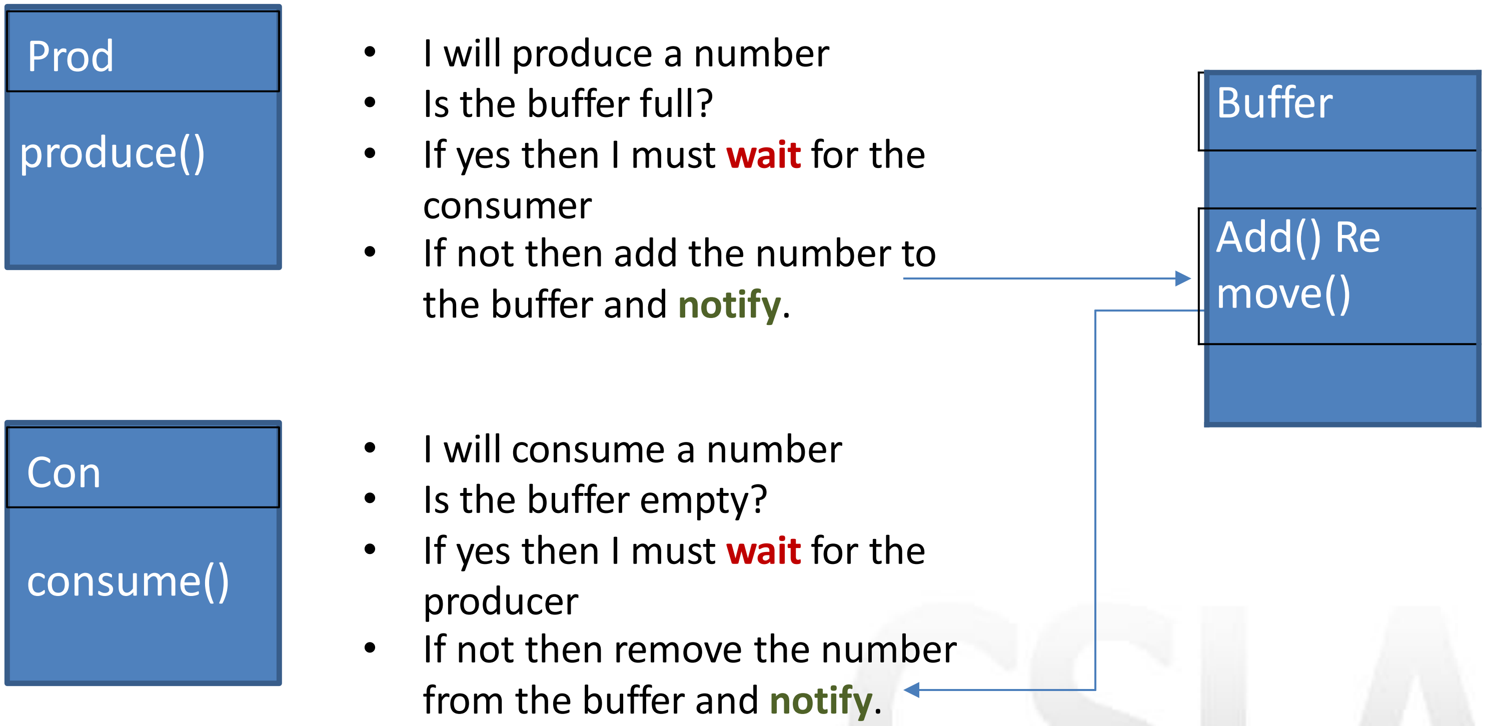
- This version of the producer consumer has a single stacked buffer
- The producer adds data to the buffer and waits if it is full
- The consumer takes data from the buffer and waits if empty

- To fully synchronize this multithreaded program we need
- Synchronized
  - Makes a method synchronous
- Wait()
  - Forces the thread to go into the blocked state
- Notify() / NotifyAll()
  - Forces the thread back into the active state.



Synchronized Producer Consumer

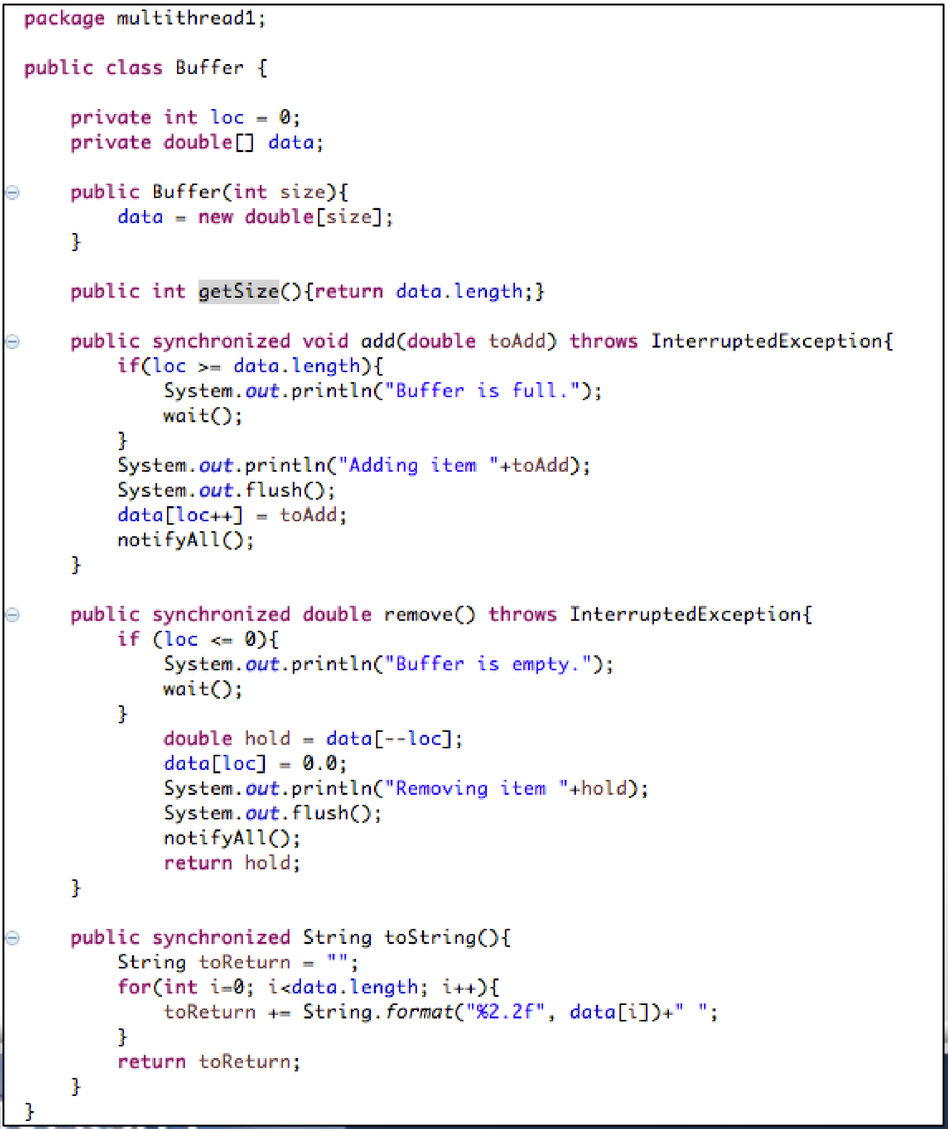
- **How the program works**



- I will produce a number
- Is the buffer full?
- If yes then I must **wait** for the consumer
- If not then add the number to the buffer and **notify**.

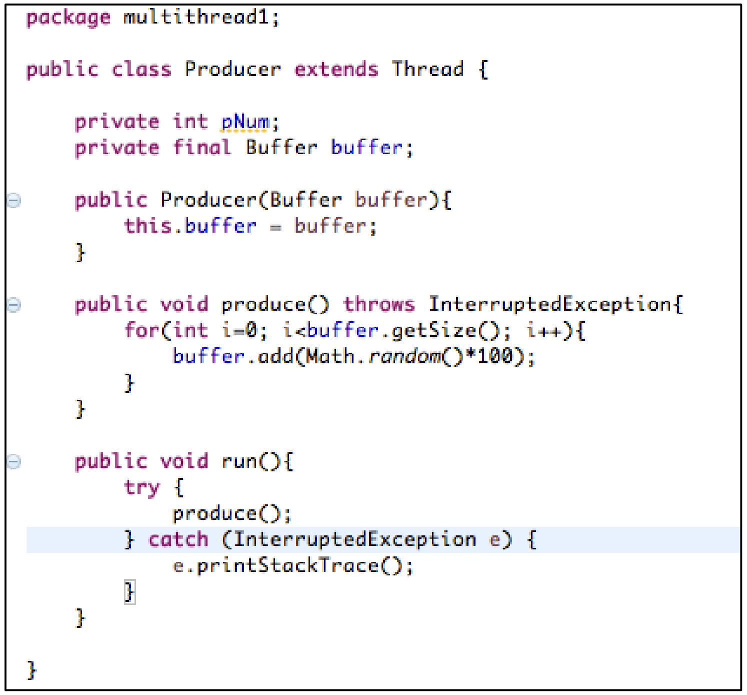
- I will consume a number
- Is the buffer empty?
- If yes then I must **wait** for the producer
- If not then remove the number from the buffer and **notify**.

Synchronized Producer Consumer

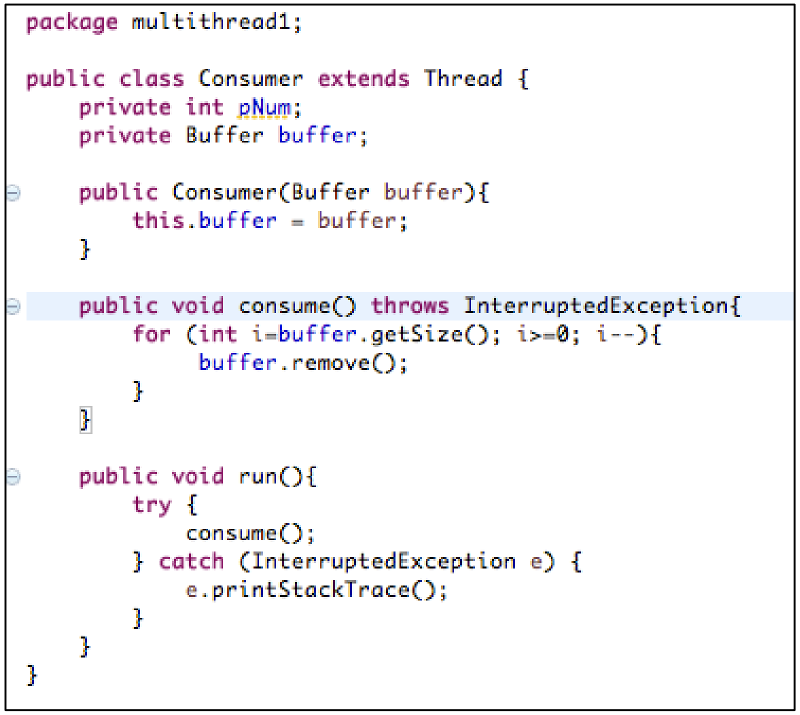


```
package multithread1;

public class Buffer {

    private int loc = 0;
    private double[] data;

    public Buffer(int size){
        data = new double[size];
    }

    public int getSize(){return data.length;}

    public synchronized void add(double toAdd) throws InterruptedException{
        if(loc >= data.length){
            System.out.println("Buffer is full.");
            wait();
        }
        System.out.println("Adding item "+toAdd);
        System.out.flush();
        data[loc++] = toAdd;
        notifyAll();
    }

    public synchronized double remove() throws InterruptedException{
        if (loc <= 0){
            System.out.println("Buffer is empty.");
            wait();
        }
            double hold = data[--loc];
            data[loc] = 0.0;
            System.out.println("Removing item "+hold);
            System.out.flush();
            notifyAll();
            return hold;
    }

    public synchronized String toString(){
        String toReturn = "";
        for(int i=0; i<data.length; i++){
            toReturn += String.format("%2.2f", data[i])+" ";
        }
        return toReturn;
    }
}
```

Synchronized Producer Consumer

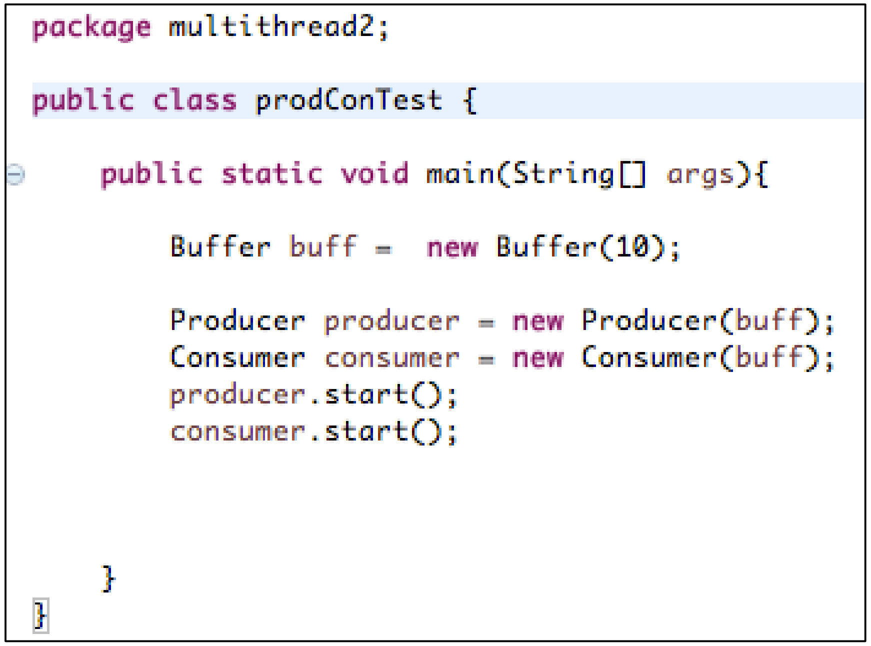


```java
package multithread1;

public class Producer extends Thread {

    private int pNum;
    private final Buffer buffer;

    public Producer(Buffer buffer){
        this.buffer = buffer;
    }

    public void produce() throws InterruptedException{
        for(int i=0; i<buffer.getSize(); i++){
            buffer.add(Math.random()*100);
        }
    }

    public void run(){
        try {
            produce();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

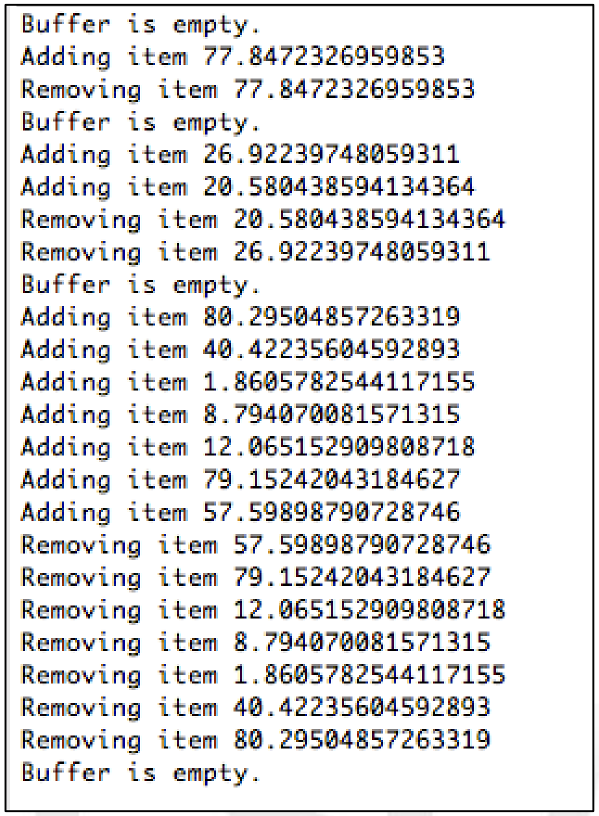


```java
package multithread1;

public class Consumer extends Thread {
    private int pNum;
    private Buffer buffer;

    public Consumer(Buffer buffer){
        this.buffer = buffer;
    }

    public void consume() throws InterruptedException{
        for (int i=buffer.getSize(); i>=0; i--){
             buffer.remove();
        }
    }

    public void run(){
        try {
            consume();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Synchronized Producer Consumer



```
package multithread2;

public class prodConTest {

    public static void main(String[] args){

        Buffer buff =  new Buffer(10);

        Producer producer = new Producer(buff);
        Consumer consumer = new Consumer(buff);
        producer.start();
        consumer.start();



    }
}
```



```
Buffer is empty.
Adding item 77.8472326959853
Removing item 77.8472326959853
Buffer is empty.
Adding item 26.92239748059311
Adding item 20.580438594134364
Removing item 20.580438594134364
Removing item 26.92239748059311
Buffer is empty.
Adding item 80.29504857263319
Adding item 40.42235604592893
Adding item 1.8605782544117155
Adding item 8.794070081571315
Adding item 12.065152909808718
Adding item 79.15242043184627
Adding item 57.59898790728746
Removing item 57.59898790728746
Removing item 79.15242043184627
Removing item 12.065152909808718
Removing item 8.794070081571315
Removing item 1.8605782544117155
Removing item 40.42235604592893
Removing item 80.29504857263319
Buffer is empty.
```

Other Methods

Method	Meaning
Thread currentThread()	returns a reference to the current thread
Void sleep(long msec)	causes the current thread to wait for msec milliseconds
String getName()	returns the name of the thread.
Int getPriority()	returns the priority of the thread
Boolean isAlive()	returns true if this thread has been started and has not Yet died. Otherwise, returns false.
Void join()	causes the caller to wait until this thread dies.
Void run()	comprises the body of the thread. This method is overridden by subclasses.
Void setName(String s)	sets the name of this thread to s.
Void setPriority(int p)	sets the priority of this thread to p.

Thread Priorities

- In java, each thread is assigned a priority, which affects the order in which it is scheduled for running.
- Java permits us to set the priority of a thread using the setPriority() method as follows:
  ThreadName.setPriority(intNumber);
- The intNumber may assume one of these constants or any value between 1 and 10.
- The intNumber is an integer value to which the thread's priority is set. The Thread class defines several priority constants:
  - MIN_PRIORITY : 1
  - NORM_PRIORITY : 5
  - MAX_PRIORITY : 10
- The default setting is NORM_PRIORITY.

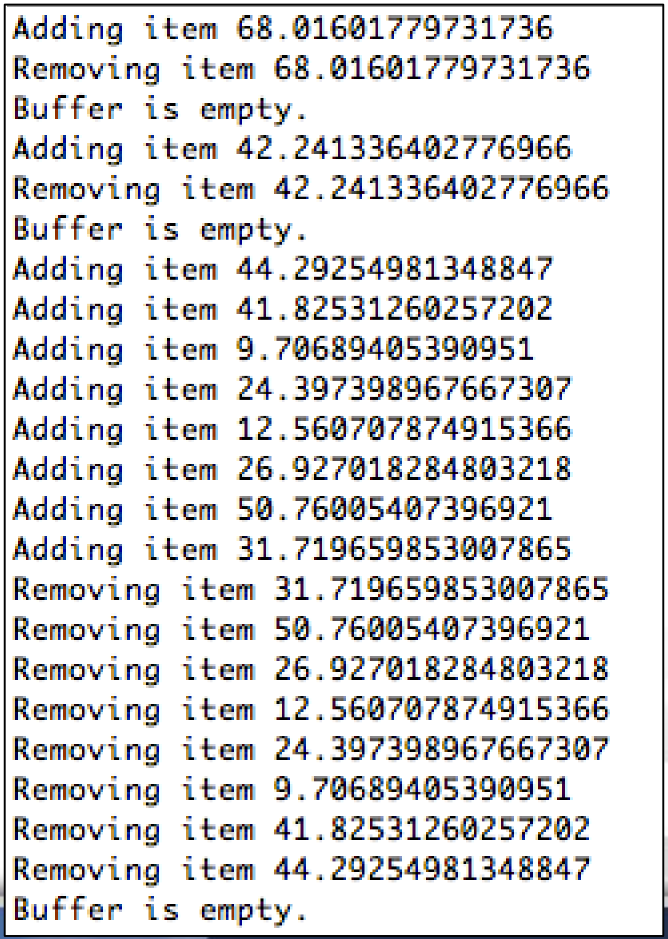
Self-Test

- **이전 슬라이드의 Producer Consumer 예제를 사용하여 진행한다.**
- **Producer Consumer 예제를 Nested Class로 변환할 것이다.**

- **Self-Test에서 비어 있는 ProdConSelfTest Class에서 수행해야하는 작업은 아래와 같다.**
  - 클래스에는 아래의 3개 인스턴스 변수가 있어야한다.
    - Private Buffer buffer
    - Private Producer producer
    - Private Consumer consumer
  - 아래와 같이 인자가 없는 생성자를 생성한다.
    - 크기가 15인 새로운 buffer.
    - 새로운 producer.
    - 새로운 consumer.
  - Producer와 Consumer Class를 private inner class로 추가한다.
  - produce와 consumer의 Thread를 시작하도록 하는 startThreads() 메소드의 비어 있는 부분을 작성한다.

Self-Test

- 아래처럼 Console 창에 출력되는지 확인한다.
  (출력되는 모양은 정확하게 일치하지 않을 수 있다..)



```
Adding item 68.01601779731736
Removing item 68.01601779731736
Buffer is empty.
Adding item 42.241336402776966
Removing item 42.241336402776966
Buffer is empty.
Adding item 44.29254981348847
Adding item 41.82531260257202
Adding item 9.70689405390951
Adding item 24.397398967667307
Adding item 12.560707874915366
Adding item 26.927018284803218
Adding item 50.76005407396921
Adding item 31.719659853007865
Removing item 31.719659853007865
Removing item 50.76005407396921
Removing item 26.927018284803218
Removing item 12.560707874915366
Removing item 24.397398967667307
Removing item 9.70689405390951
Removing item 41.82531260257202
Removing item 44.29254981348847
Buffer is empty.
```