Université Concordia University

CENTRE FOR CONTINUING EDUCATION

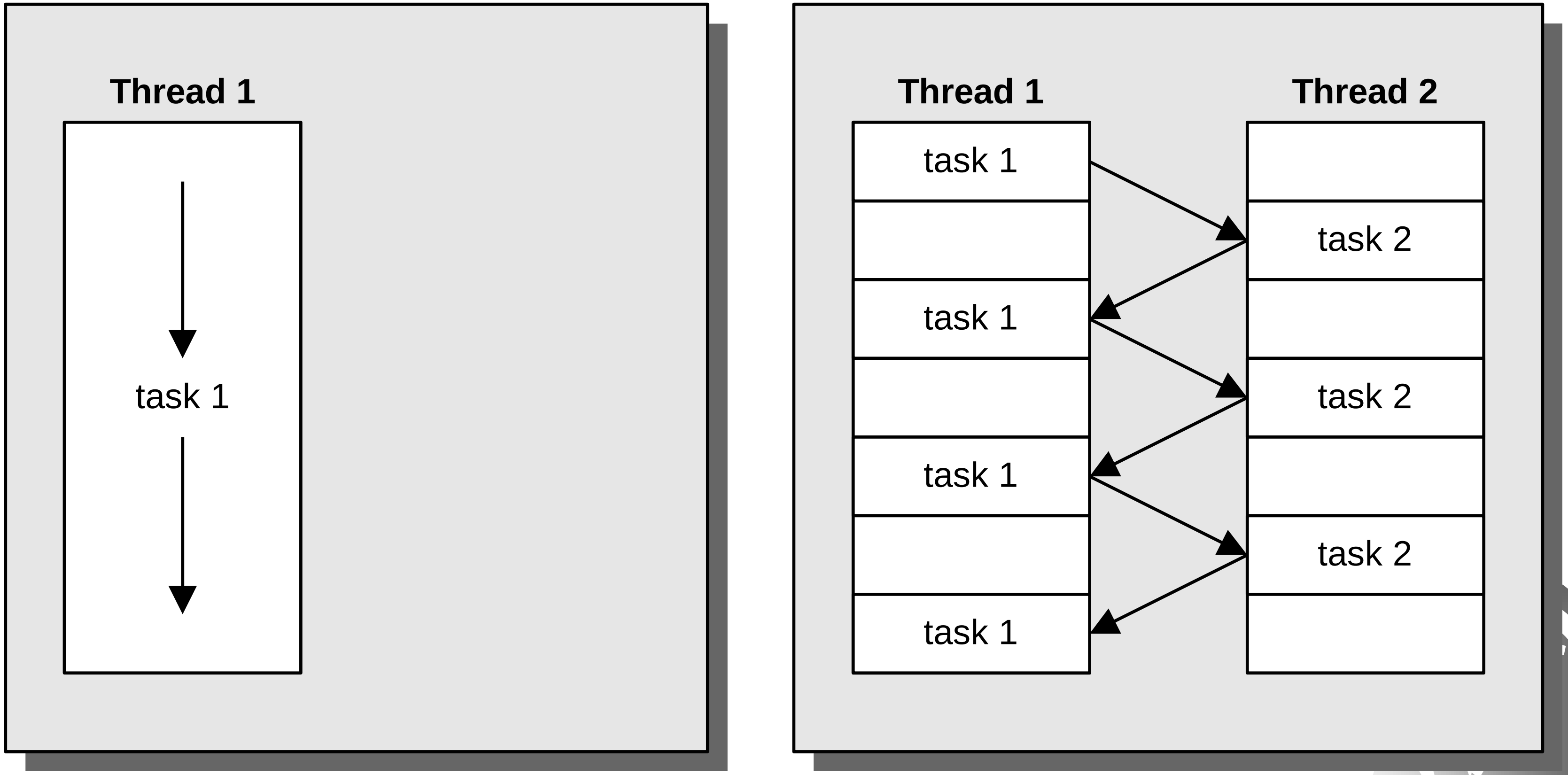
DESKTOP APPLICATION DEVELOPMENT WITH JAVA – CEJV569

Lecture #9

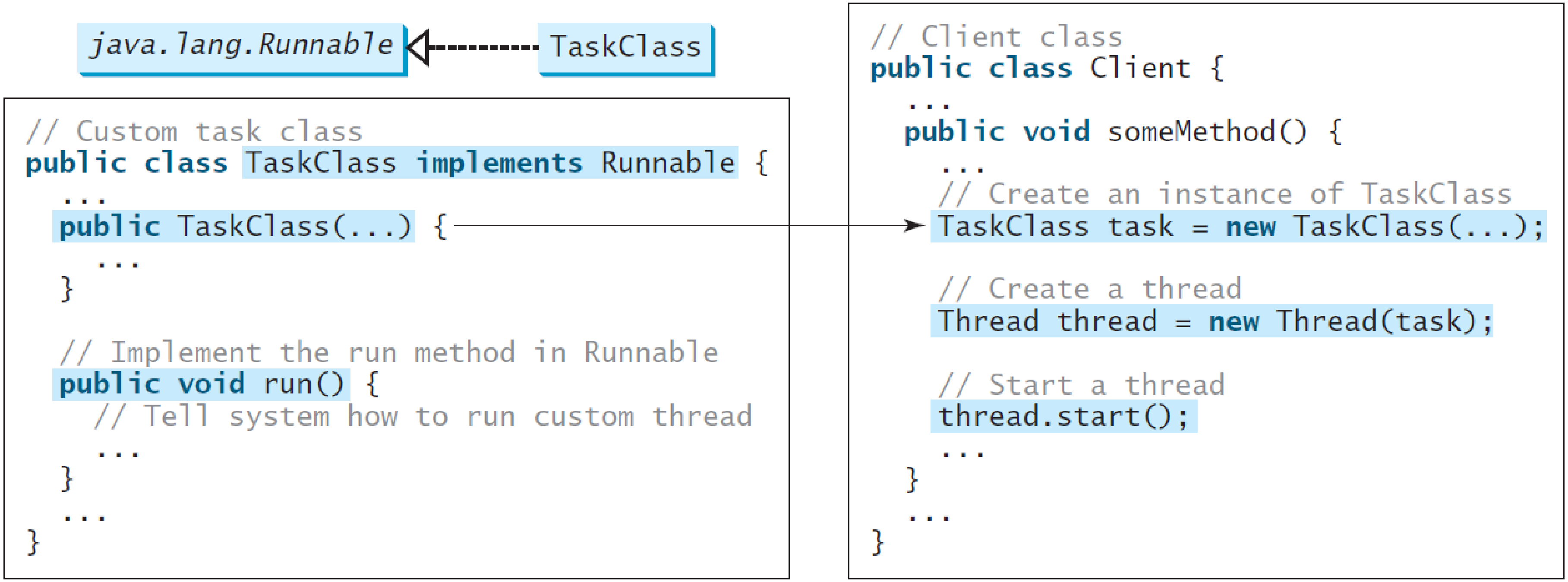
Threads

Design Patterns

How using threads can improve performance



Creating Tasks and Threads



```
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

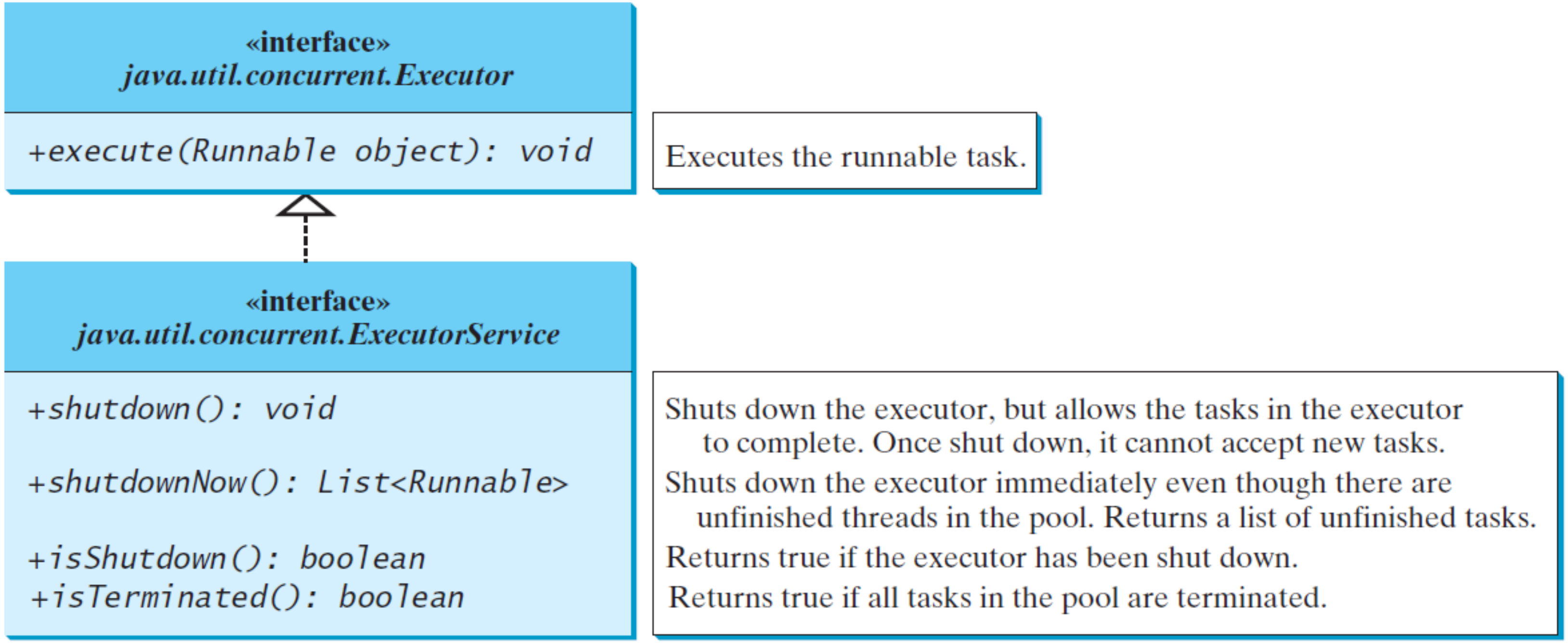
(a)

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

(b)

Thread Pools

- Starting a new thread for each task could limit throughput and cause poor performance.
- A thread pool is ideal to manage the number of tasks executing concurrently.



«interface»
java.util.concurrent.Executor

+execute(Runnable object): void

Executes the runnable task.

«interface»
java.util.concurrent.ExecutorService

+shutdown(): void

+shutdownNow(): List<Runnable>

+isShutdown(): boolean
+isTerminated(): boolean

Shuts down the executor, but allows the tasks in the executor to complete. Once shut down, it cannot accept new tasks.

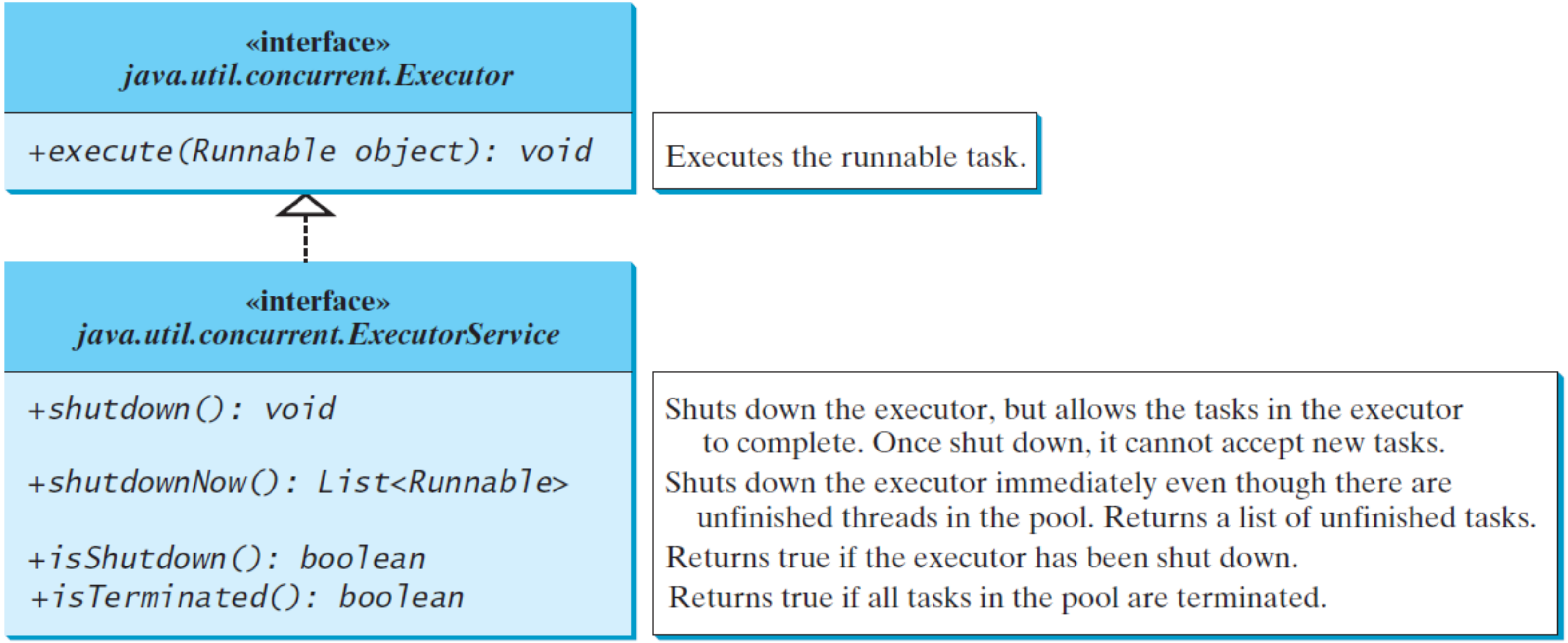
Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.

Returns true if the executor has been shut down.

Returns true if all tasks in the pool are terminated.

Thread Pools

- JDK 1.5 uses the Executor interface for executing tasks in a thread pool and the ExecutorService interface for managing and controlling tasks.
- ExecutorService is a subinterface of Executor.



«interface»
java.util.concurrent.Executor

`+execute(Runnable object): void`

Executes the runnable task.

«interface»
java.util.concurrent.ExecutorService

`+shutdown(): void`

Shuts down the executor, but allows the tasks in the executor to complete. Once shut down, it cannot accept new tasks.

`+shutdownNow(): List<Runnable>`

Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.

`+isShutdown(): boolean`

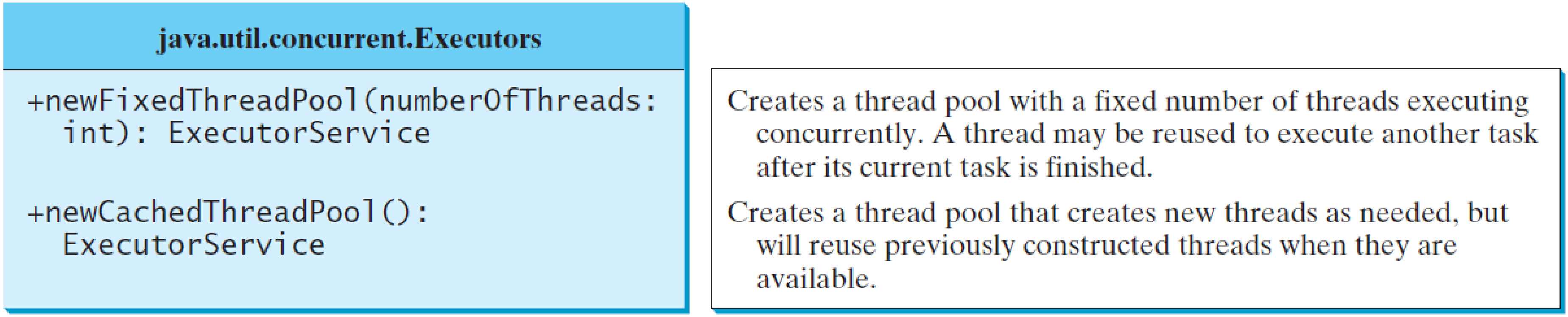
Returns true if the executor has been shut down.

`+isTerminated(): boolean`

Returns true if all tasks in the pool are terminated.

Creating Executors

To create an Executor object, use the static methods in the Executors class.



java.util.concurrent.Executors

```
+newFixedThreadPool(numberOfThreads:
   int): ExecutorService

+newCachedThreadPool():
   ExecutorService
```

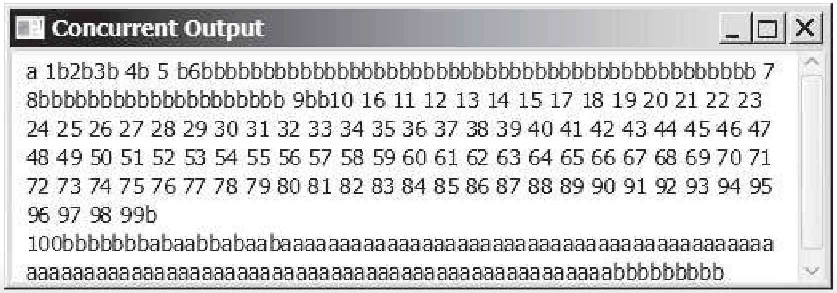
Creates a thread pool with a fixed number of threads executing concurrently. A thread may be reused to execute another task after its current task is finished.

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

ExecutorDemo

Exercise 29

- Create and run three threads:
  - The first thread prints the letter *a* 100 times.
  - The second thread prints the letter *b* 100 times.
  - The third thread prints the integers 1 through 100.
  - After Texting your code, display the output in a text area, as shown in the following Figure:



```
a 1b2b3b 4b 5 b6bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb 7
8bbbbbbbbbbbbbbbbbbbb 9bb10 16 11 12 13 14 15 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
96 97 98 99b
100bbbbbbbabaabbabaabaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbb
```

Sharing limited resources

- A single-threaded program moves around through the problem space as a single entity doing one thing at a time
- Never have to think about the problem of two entities trying to use the same resource at the same time
- Problems like:
  - two people trying to park in the same space
  - walk through a door at the same time
  - talk at the same time

Sharing limited resources

- With multithreading there is the possibility of two or more threads trying to use the same limited resource at once
- Colliding over a resource must be prevented
- Otherwise two threads can try to:
  - access the same bank account at the same time
  - print to the same printer
  - adjust the same valve

Thread Synchronization

- A shared resource may be corrupted if it is accessed simultaneously by multiple threads.
- For example, two unsynchronized threads accessing the same bank account may cause conflict.

Step	balance	thread[i]	thread[j]
1	0	newBalance = bank.getBalance() + 1;	
2	0		newBalance = bank.getBalance() + 1;
3	1	bank.setBalance(newBalance);	
4	1		bank.setBalance(newBalance);

Race Condition

What, then, caused the error in the example? Here is a possible scenario:

Step	balance	Task 1	Task 2
1	0	newBalance = balance + 1;	
2	0		newBalance = balance + 1;
3	1	balance = newBalance;	
4	1		balance = newBalance;

- The effect of this scenario is that Task 1 did nothing, because in Step 4 Task 2 overrides Task 1's result.
- Obviously, the problem is that Task 1 and Task 2 are accessing a common resource in a way that causes conflict.
- This is a common problem known as a *race condition* in multithreaded programs.

Race Condition

What, then, caused the error in the example? Here is a possible scenario:

Step	balance	Task 1	Task 2
1	0	newBalance = balance + 1;	
2	0		newBalance = balance + 1;
3	1	balance = newBalance;	
4	1		balance = newBalance;

- A class is said to be *thread-safe* if an object of the class does not cause a race condition in the presence of multiple threads.
- As demonstrated in the preceding example, the Account class is not thread-safe.

Resolving shared resource contention

- Virtually all multithreading schemes *serialize access to shared resources*.
- Only one thread at a time is allowed to access the shared resource.
- Put a locked clause around a piece of code
  - only one thread at a time may pass through that piece of code
- Produces *mutual exclusion*
- Called a *mutex*

Resolving shared resource contention

- Java prevents collisions over resources with the **synchronized** keyword
- When a thread wishes to execute a piece of code guarded by the **synchronized** keyword, it
  - checks to see if the resource is available
  - acquires it
  - executes the code
  - releases it
- **synchronized** is built into the language

Resolving shared resource contention

- Shared resource may be
  - a piece of memory in the form of an object
  - a file
  - an I/O port
  - something like a printer.
- To control access to a shared resource, put it inside an object
- Any method that accesses that resource can be made **synchronized**.

Resolving shared resource contention

- Typically data elements of a class are **private**
- Access that memory only through methods
- Prevent collisions by making methods **synchronized**

```
synchronized void f() { /* ... */
}
synchronized void g(){ /* ... */ }
```

Resolving shared resource contention

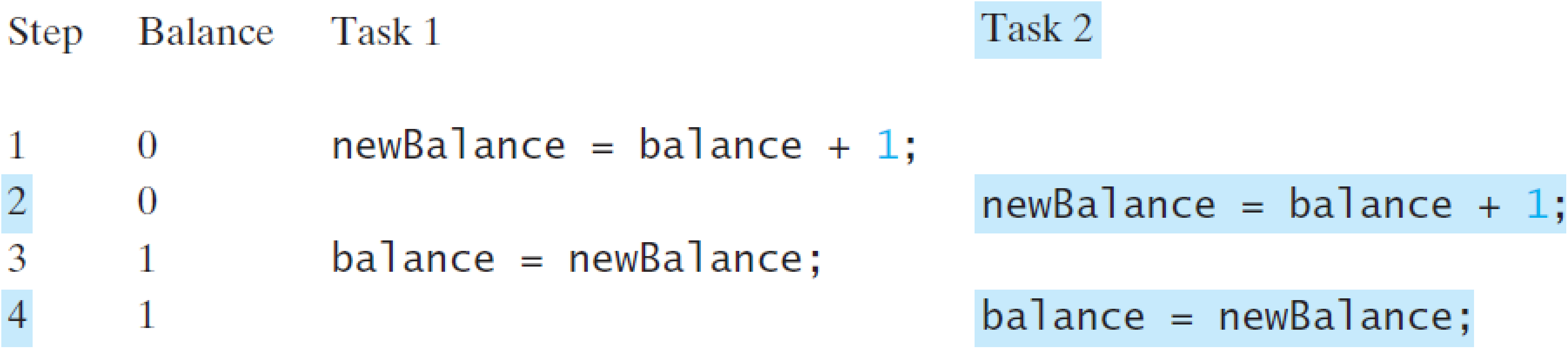
- Each object contains a single lock (*monitor*) that is part of the object
- When a **synchronized** method is called:
  - object is locked
  - no other **synchronized** method of that object can be called
- A single lock is shared by all the **synchronized** methods of an object
- Prevents common memory from being written by more than one thread at a time

Synchronizing Instance Methods and Static Methods

- A synchronized method acquires a lock before it executes.
- In the case of an instance method, the lock is on the object for which the method was invoked.
- In the case of a static method, the lock is on the class.
- If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired first, then the method is executed, and finally the lock is released.
- Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

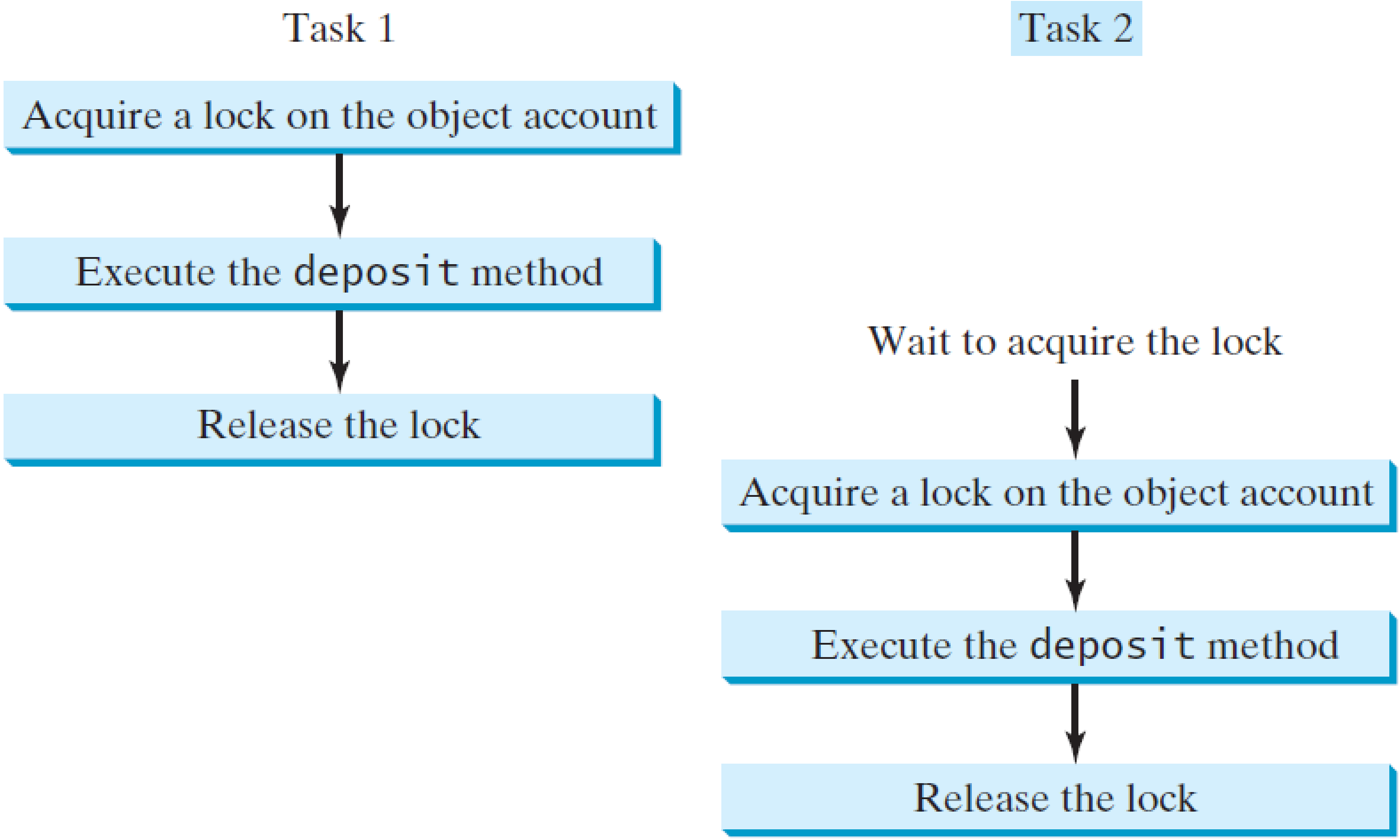
Synchronizing Instance Methods and Static Methods

With the deposit method synchronized, the preceding scenario cannot happen. If Task 2 starts to enter the method, and Task 1 is already in the method, Task 2 is blocked until Task 1 finishes the method.



Step	Balance	Task 1	Task 2
1	0	`newBalance = balance + 1;`	
2	0		`newBalance = balance + 1;`
3	1	`balance = newBalance;`	
4	1		`balance = newBalance;`

Synchronizing Tasks



Task 1

Acquire a lock on the object account

Execute the `deposit` method

Release the lock

Task 2

Wait to acquire the lock

Acquire a lock on the object account

Execute the `deposit` method

Release the lock

Synchronizing Statements vs. Methods

- Any synchronized instance method can be converted into a synchronized statement.
- Suppose that the following is a synchronized instance method:

```
public synchronized void xMethod() {
  // method body
}
```

This method is equivalent to

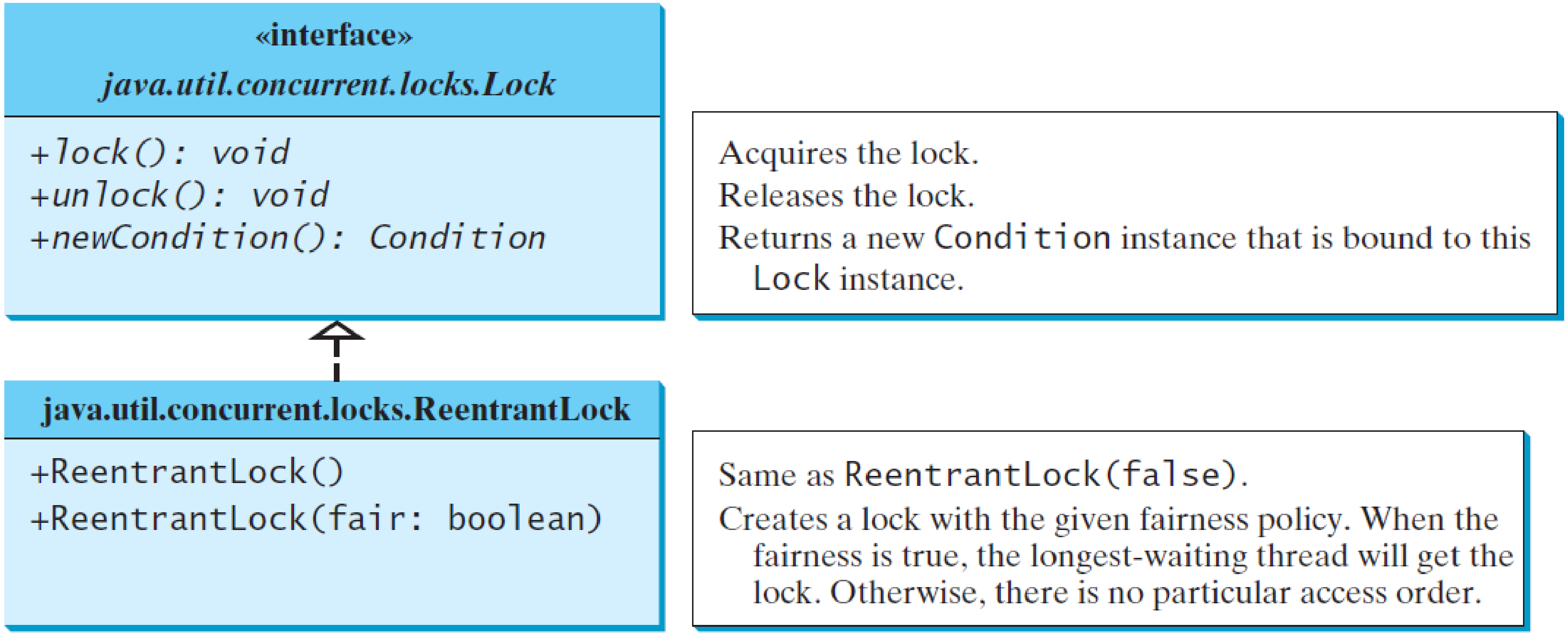
```
public void xMethod() {
  synchronized (this) {
    // method body
  }
}
```

Synchronization Using Locks

- A synchronized instance method implicitly acquires a lock on the instance before it executes the method.

- JDK 1.5 enables you to use locks explicitly.

- The new locking features are flexible and give you more control for coordinating threads.



«interface»
java.util.concurrent.locks.Lock

```
+lock(): void
+unlock(): void
+newCondition(): Condition
```

Acquires the lock.
Releases the lock.
Returns a new `Condition` instance that is bound to this `Lock` instance.

java.util.concurrent.locks.ReentrantLock

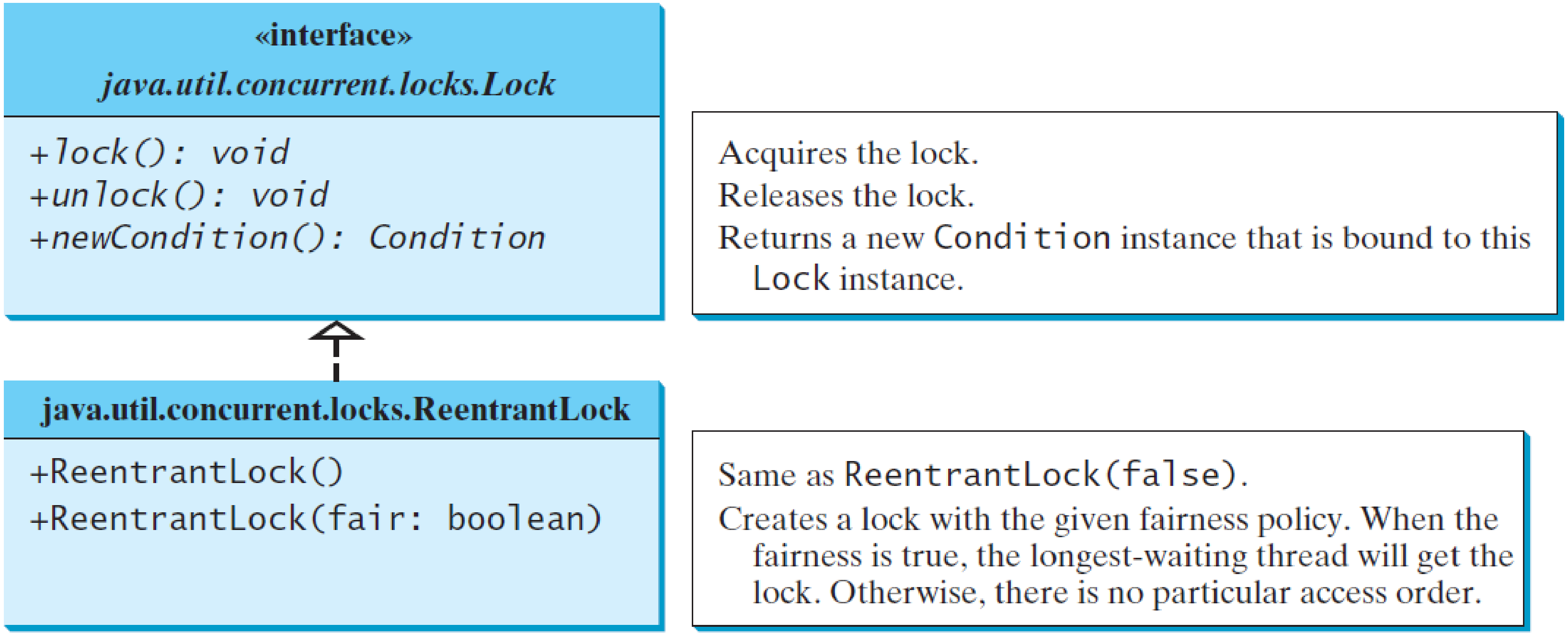
```
+ReentrantLock()
+ReentrantLock(fair: boolean)
```

Same as `ReentrantLock(false)`.
Creates a lock with the given fairness policy. When the fairness is true, the longest-waiting thread will get the lock. Otherwise, there is no particular access order.

Synchronization Using Locks

- A lock is an instance of the Lock interface, which declares the methods for acquiring and releasing locks, as shown in the following Figure.

- A lock may also use the newCondition() method to create any number of Condition objects, which can be used for thread communications.



«interface»
java.util.concurrent.locks.Lock

+lock(): void
+unlock(): void
+newCondition(): Condition

Acquires the lock.
Releases the lock.
Returns a new Condition instance that is bound to this Lock instance.

java.util.concurrent.locks.ReentrantLock

+ReentrantLock()
+ReentrantLock(fair: boolean)

Same as ReentrantLock(false).
Creates a lock with the given fairness policy. When the fairness is true, the longest-waiting thread will get the lock. Otherwise, there is no particular access order.

Fairness Policy

- ReentrantLock is a concrete implementation of Lock for creating mutual exclusive locks.

- You can create a lock with the specified fairness policy.

- True fairness policies guarantee the longest-wait thread to obtain the lock first.

- False fairness policies grant a lock to a waiting thread without any access order.

- Programs using fair locks accessed by many threads may have poor overall performance than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation.

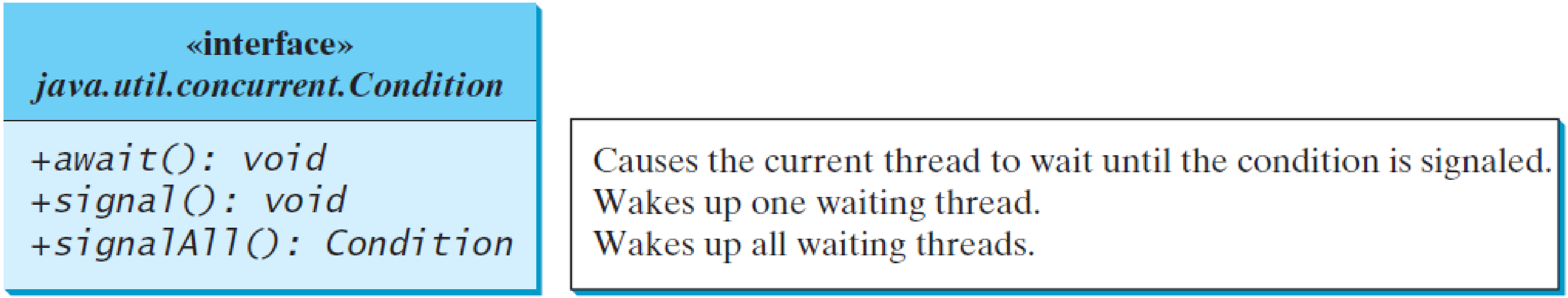
Example: Using Locks

This example revises Account without synchornization to synchronize the account modification using explicit locks.

AccountWithSyncUsingLock

Cooperation Among Threads

- The conditions can be used to facilitate communications among threads.
- A thread can specify what to do under a certain condition.
- Conditions are objects created by invoking the newCondition() method on a Lock object.
- Once a condition is created, you can use its await(), signal(), and signalAll() methods for thread communications, as shown in the following Figure.



«interface»
java.util.concurrent.Condition

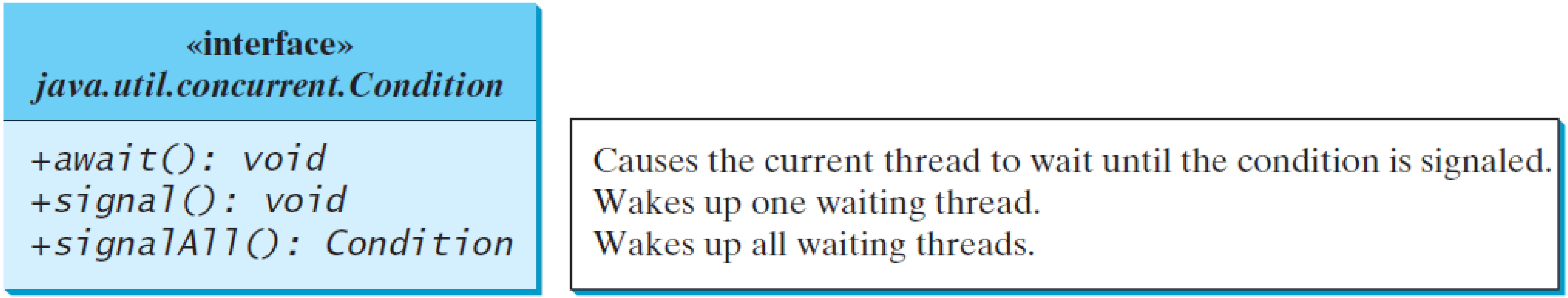
```
+await(): void
+signal(): void
+signalAll(): Condition
```

Causes the current thread to wait until the condition is signaled.
Wakes up one waiting thread.
Wakes up all waiting threads.

Cooperation Among Threads

- The await() method causes the current thread to wait until the condition is signaled.

- The signal() method wakes up one waiting thread, and the signalAll() method wakes all waiting threads.



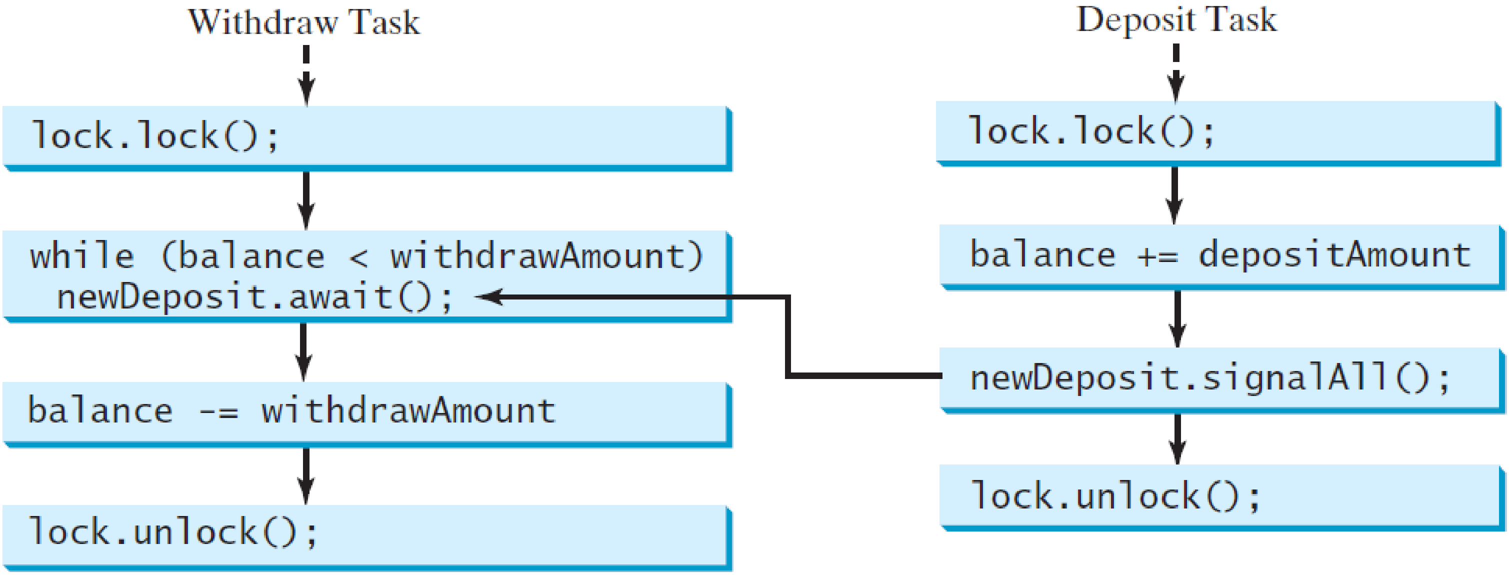
«interface»
java.util.concurrent.Condition

```
+await(): void
+signal(): void
+signalAll(): Condition
```

Causes the current thread to wait until the condition is signaled.
Wakes up one waiting thread.
Wakes up all waiting threads.

Cooperation Among Threads

- To synchronize the operations, use a lock with a condition: newDeposit (i.e., new deposit added to the account).
- If the balance is less than the amount to be withdrawn, the withdraw task will wait for the newDeposit condition.
- When the deposit task adds money to the account, the task signals the waiting withdraw task to try again.



Withdraw Task

```
lock.lock();

while (balance < withdrawAmount)
  newDeposit.await();

balance -= withdrawAmount

lock.unlock();
```

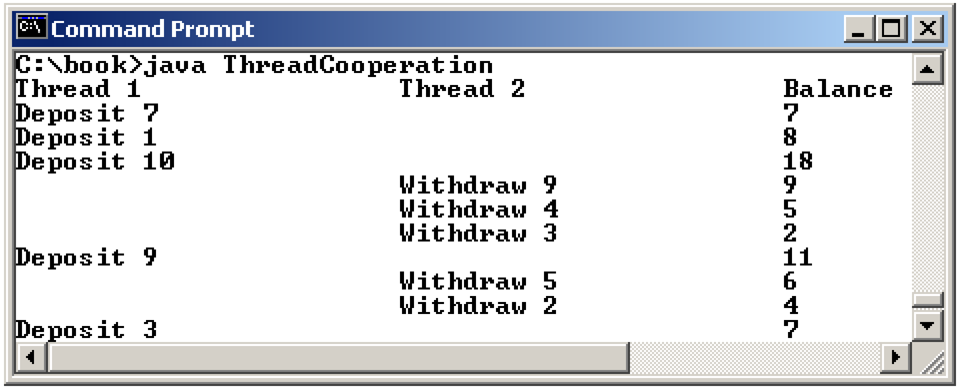
Deposit Task

```
lock.lock();

balance += depositAmount

newDeposit.signalAll();

lock.unlock();
```

Example: Thread Cooperation

- Write a program that demonstrates thread cooperation.
- Suppose that you create and launch two threads, one deposits to an account, and the other withdraws from the same account.
- The second thread has to wait if the amount to be withdrawn is more than the current balance in the account.
- Whenever new fund is deposited to the account, the first thread notifies the second thread to resume.
- If the amount is still not enough for a withdrawal, the second thread has to continue to wait for more fund in the account.
- Assume the initial balance is 0 and the amount to deposit and to withdraw is randomly generated.

Example: Thread Cooperation



```
C:\book>java ThreadCooperation
Thread 1                Thread 2                Balance
Deposit 7                                       7
Deposit 1                                       8
Deposit 10                                      18
                        Withdraw 9              9
                        Withdraw 4              5
                        Withdraw 3              2
Deposit 9                                       11
                        Withdraw 5              6
                        Withdraw 2              4
Deposit 3                                       7
```

ThreadCooperation

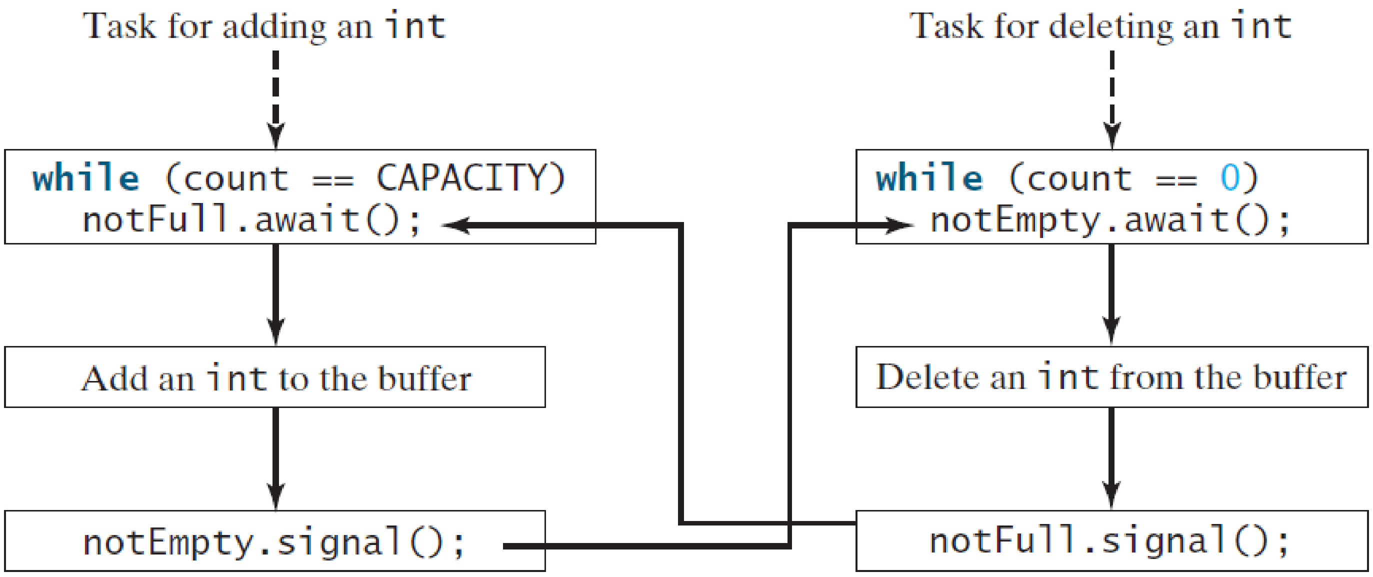
Case Study: Producer/Consumer

- Consider the classic Consumer/Producer example.
- Suppose you use a buffer to store integers.
  - The buffer size is limited.
  - The buffer provides the method write(int) to add an int value to the buffer and the method read() to read and delete an int value from the buffer.

- To synchronize the operations, use a lock with two conditions:
  - notEmpty (i.e., buffer is not empty)
  - notFull (i.e., buffer is not full).

- When a task adds an int to the buffer, if the buffer is full, the task will wait for the notFull condition.
- When a task deletes an int from the buffer, if the buffer is empty, the task will wait for the notEmpty condition.

Case Study: Producer/Consumer



Task for adding an int

```
while (count == CAPACITY)
  notFull.await();
```

Add an int to the buffer

```
notEmpty.signal();
```

Task for deleting an int

```
while (count == 0)
  notEmpty.await();
```

Delete an int from the buffer

```
notFull.signal();
```

Case Study: Producer/Consumer

- The program contains the Buffer class and two tasks for repeatedly producing and consuming numbers to and from the buffer.
  - The write(int) method adds an integer to the buffer.
  - The read() method deletes and returns an integer from the buffer.
- For simplicity, the buffer is implemented using a linked list.
- Two conditions notEmpty and notFull on the lock are created.
- The conditions are bound to a lock.
  - A lock must be acquired before a condition can be applied.

ConsumerProducer

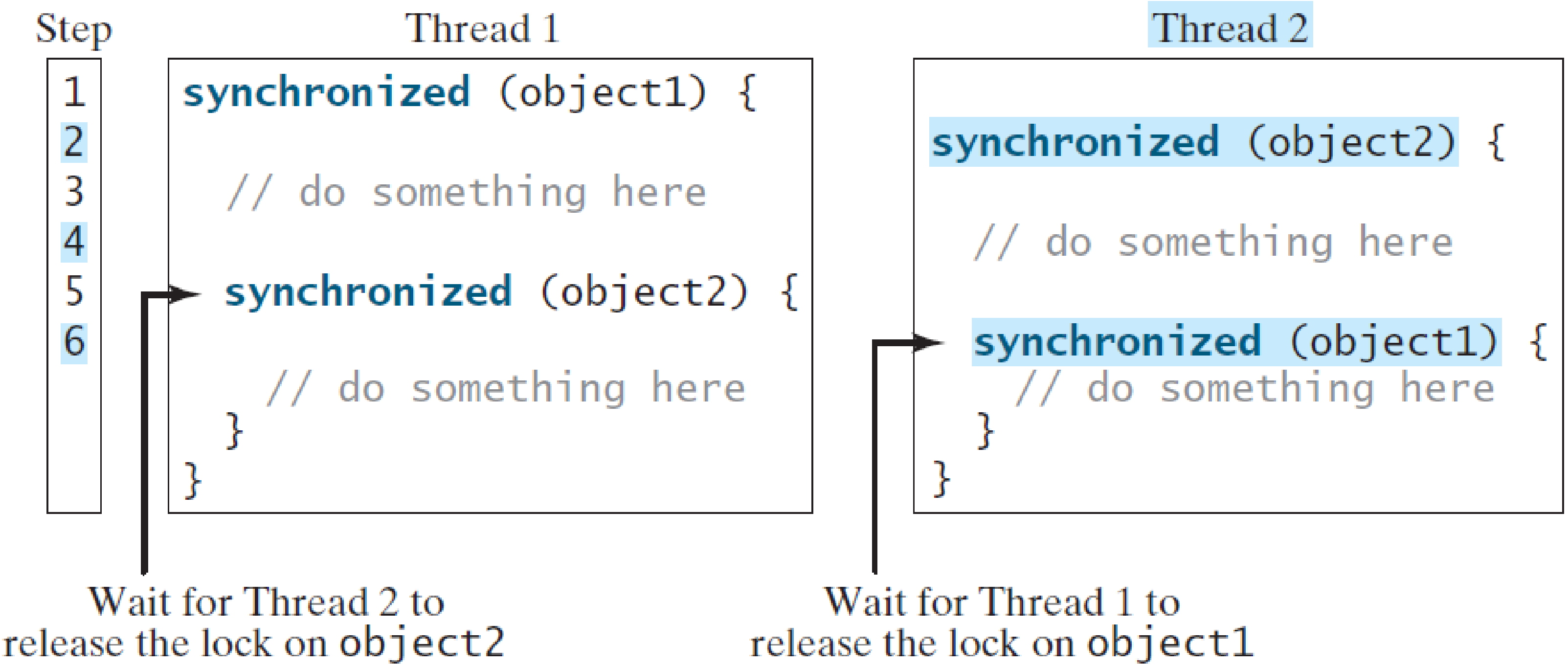
Exercise 30

☞ Modify ThreadCooperation.java:

- No usage of Lock
- Use Synchronization
- Use the object’s **wait()** and **notifyAll()** methods.

Deadlock

- Sometimes two or more threads need to acquire the locks on several shared objects.
- This could cause *deadlock*, in which each thread has the lock on one of the objects and is waiting for the lock on the other object.
- Consider the scenario with two threads and two objects, as shown in the following Figure.



Step

1
2
3
4
5
6

Thread 1

```
synchronized (object1) {

  // do something here

  synchronized (object2) {

    // do something here
  }
}
```

Thread 2

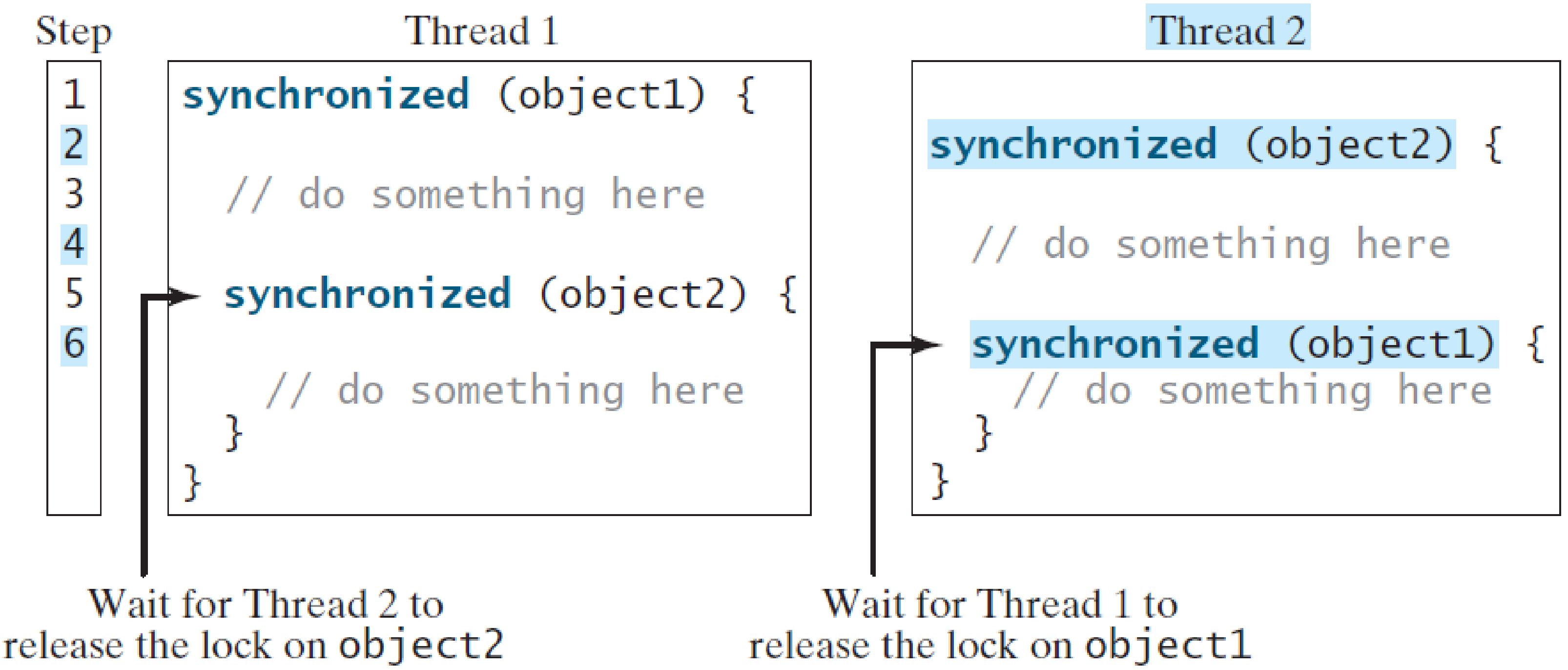
```

synchronized (object2) {

  // do something here

  synchronized (object1) {
    // do something here
  }
}
```

Wait for Thread 2 to
release the lock on object2

Wait for Thread 1 to
release the lock on object1

Deadlock

- Thread 1 acquired a lock on object1 and Thread 2 acquired a lock on object2.
- Now Thread 1 is waiting for the lock on object2 and Thread 2 for the lock on object1.
- The two threads wait for each other to release the in order to get the lock, and neither can continue to run.



```
synchronized (object1) {

  // do something here

  synchronized (object2) {

    // do something here
  }
}
```

```

synchronized (object2) {

  // do something here

  synchronized (object1) {
    // do something here
  }
}
```

Preventing Deadlock

- Deadlock can be easily avoided by using a simple technique known as resource ordering.
- With this technique, you assign an order on all the objects whose locks must be acquired and ensure that each thread acquires the locks in that order.
- For the previous example, suppose the objects are ordered as object1 and object2.
- Using the resource ordering technique, Thread 2 must acquire a lock on object1 first, then on object2.
- Once Thread 1 acquired a lock on object1, Thread 2 has to wait for a lock on object1.
- So Thread 1 will be able to acquire a lock on object2 and no deadlock would occur.

What is a Design Pattern

- If a problem occurs over and over again, a solution to that problem has been developed
- That solution is described as a pattern
- Design patterns are language-independent strategies for solving common object-oriented design problems
- Design patterns are not idioms or algorithms or components

What is a Design Pattern?

- Design patterns represent the best practices used by experienced object-oriented software developers.
- Design patterns are solutions to general problems that software developers faced during software development.
- These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Types of Design Patterns

☞ As per the design pattern reference book **Design Patterns - Elements of Reusable Object-Oriented Software** , there are many design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns.

Types of Design Patterns

S.N	Pattern & Description
1	**Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.
2	**Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3	**Behavioral Patterns** These design patterns are specifically concerned with communication between objects.

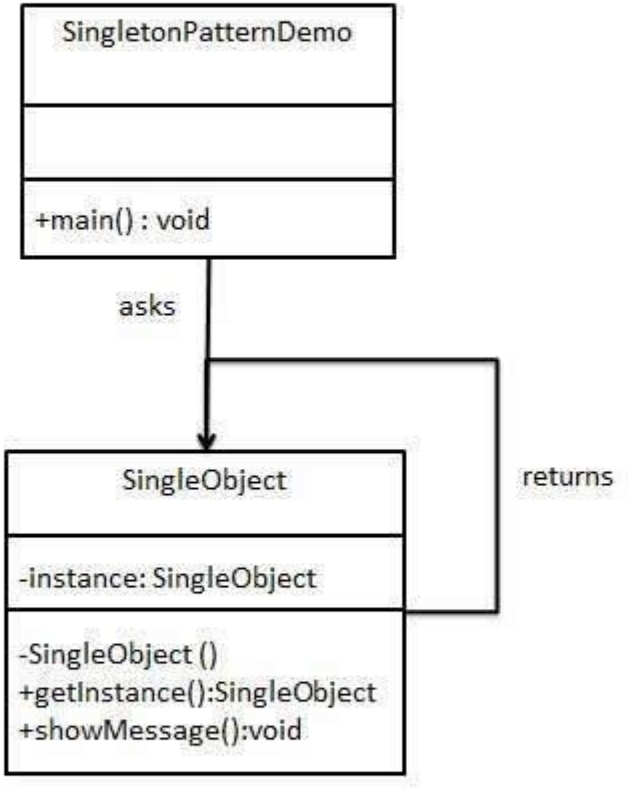
Singleton Pattern - Creational

- *Ensure a class has only one instance and provide a global point of access to it.*
- Used in the design of logger classes.
- Provides a global logging access point in all the application components

Singleton Pattern - Creational

- Singleton pattern is one of the simplest design patterns in Java.
- 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 a single class which is responsible to create an object while making sure that only single object gets created.
- This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

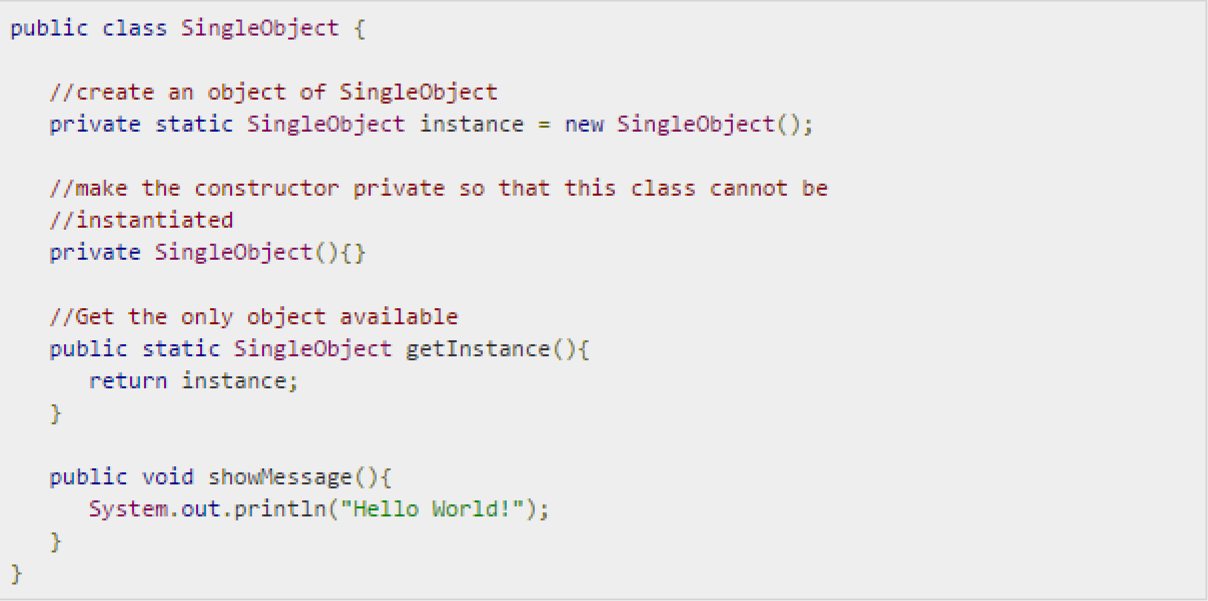
Singleton Pattern - Creational



Step 1

- Create a Singleton Class.

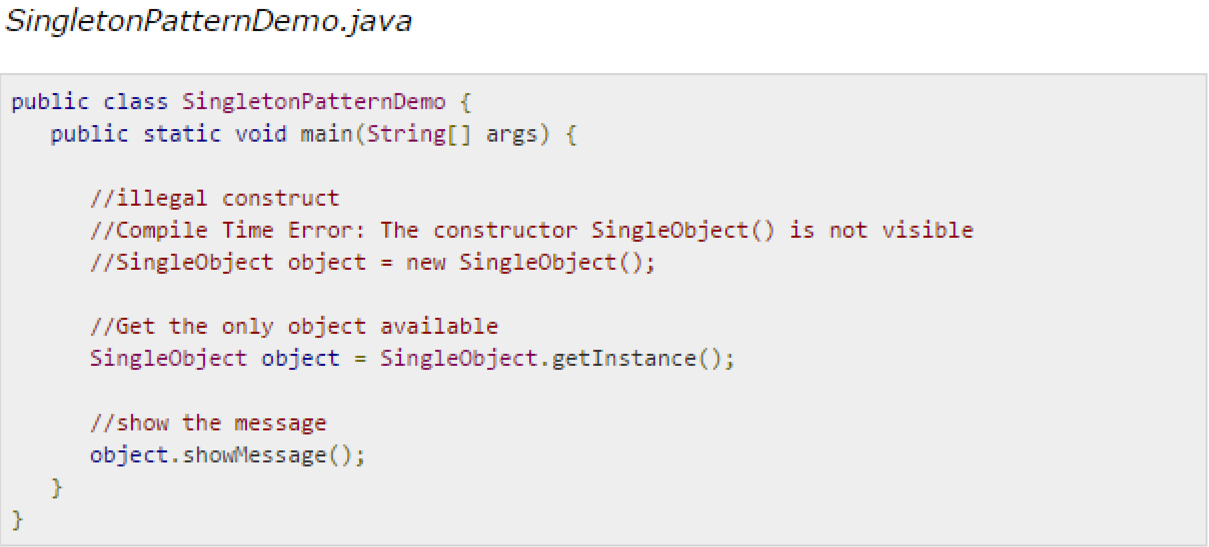
SingleObject.java



```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Step 2

☞ Get the only object from the singleton class.



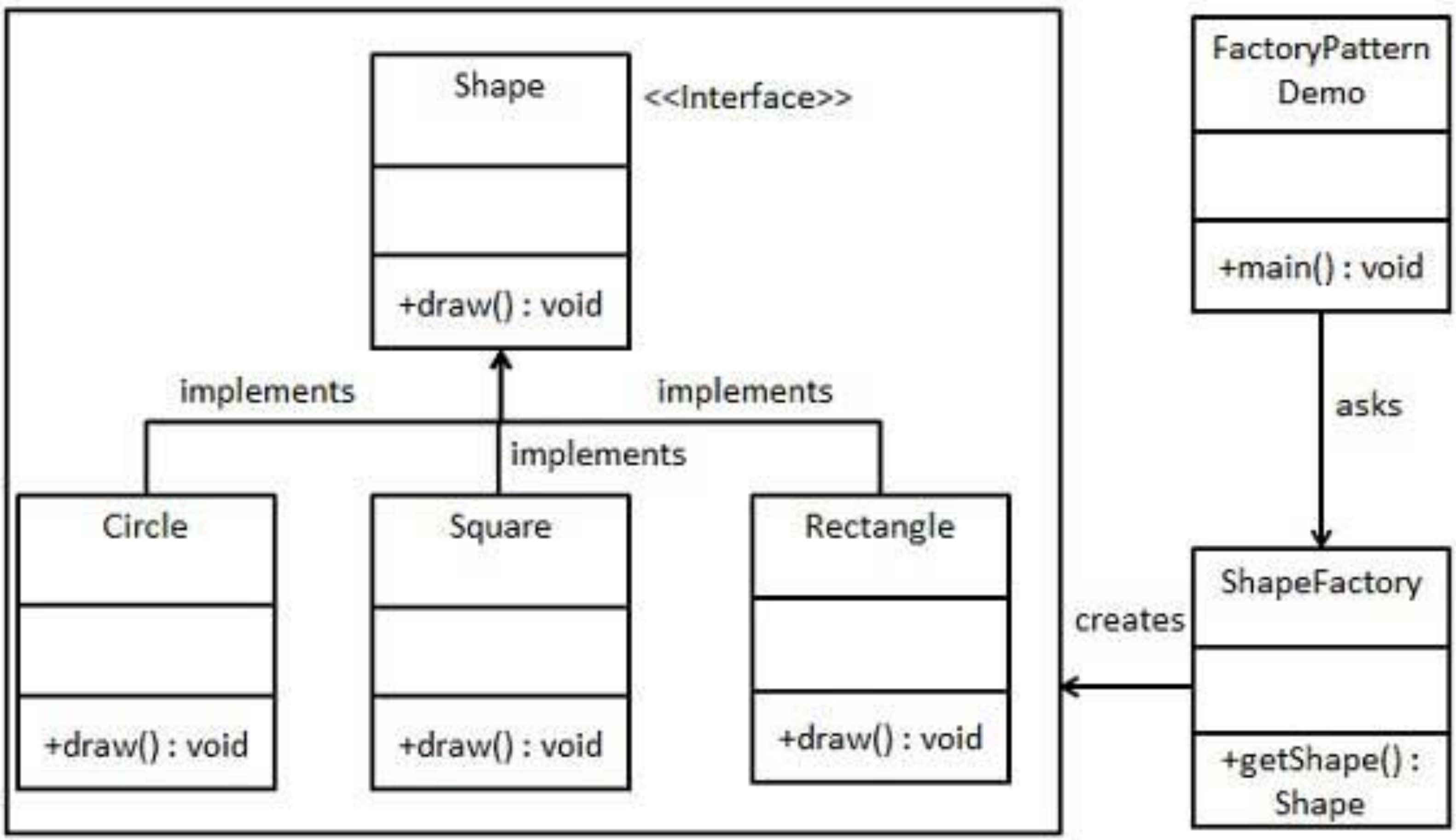
SingletonPatternDemo.java

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

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Factory Method Pattern - Creational

- *Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses*
- A factory is somewhere that items get produced.
- Uses a single method of an object to create the appropriate object
- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Factory Method Pattern - Creational



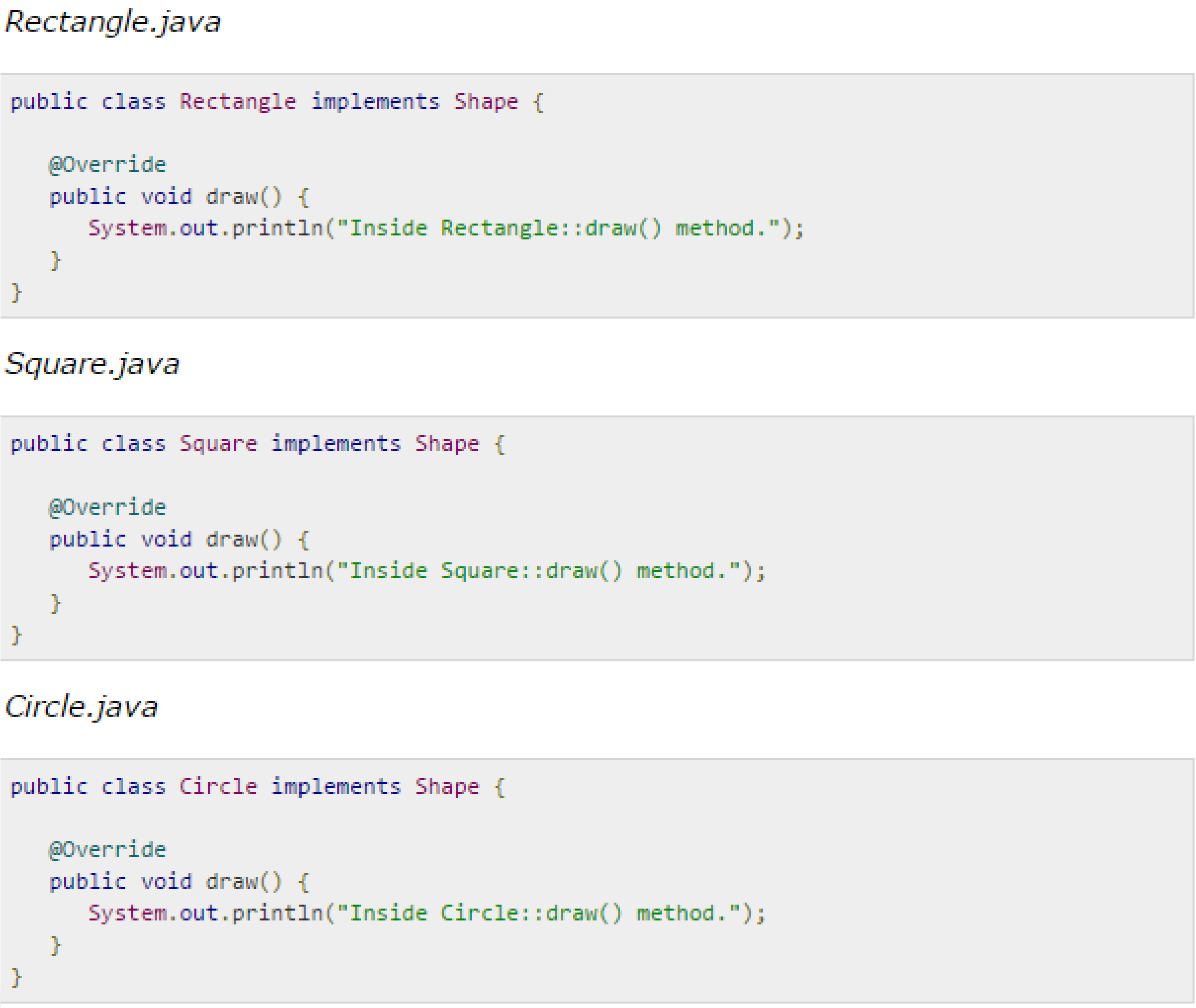
Step 1

- ☞ Create an interface.
  - – *Shape.java*

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

Step 2

- Create concrete classes implementing the same interface.



Rectangle.java

```java
public class Rectangle implements Shape {

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

Square.java

```java
public class Square implements Shape {

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

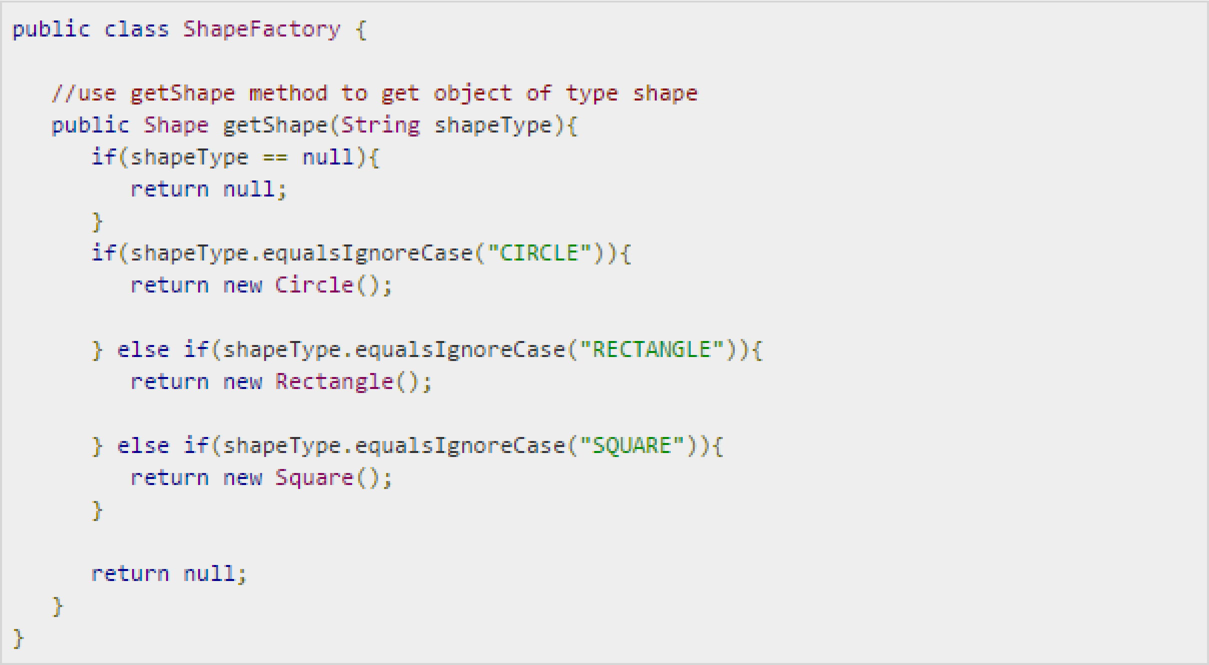
Circle.java

```java
public class Circle implements Shape {

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

Step 3

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

ShapeFactory.java

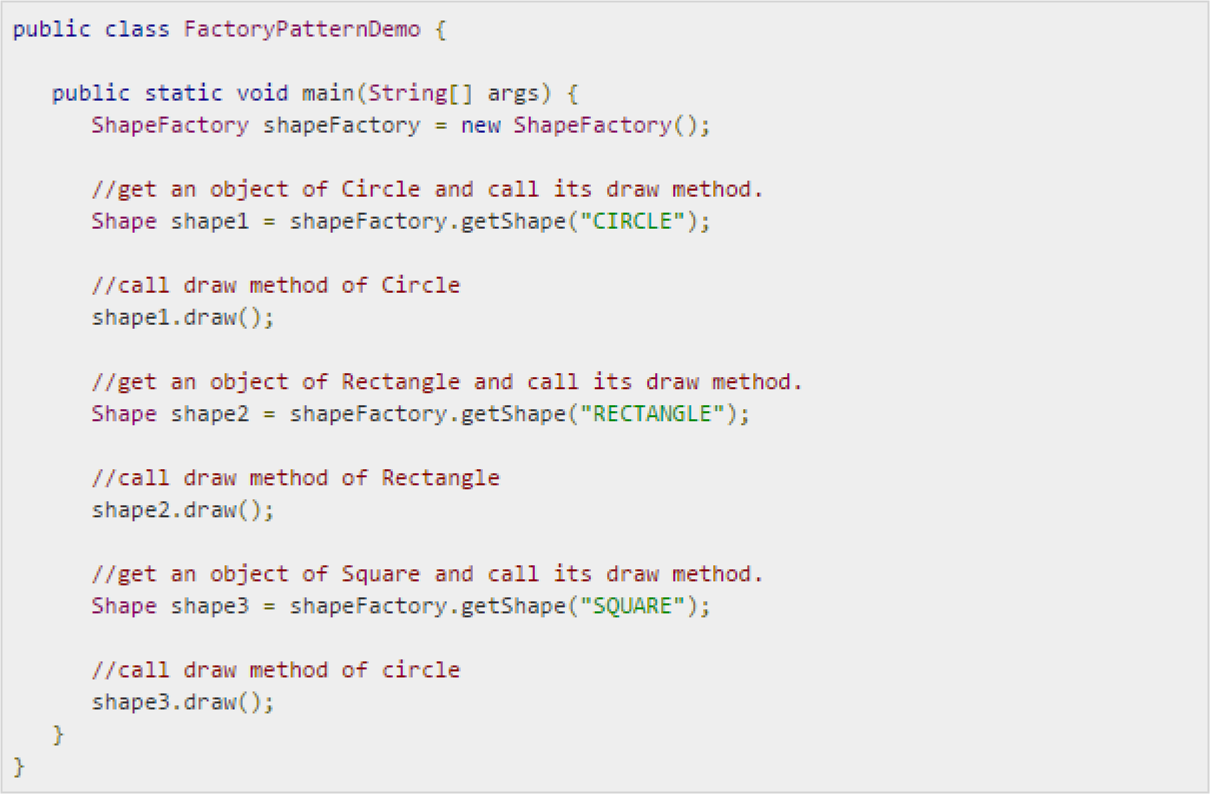


```java
public class ShapeFactory {

   //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();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```

Step 4

- 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();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of circle
      shape3.draw();
   }
}
```

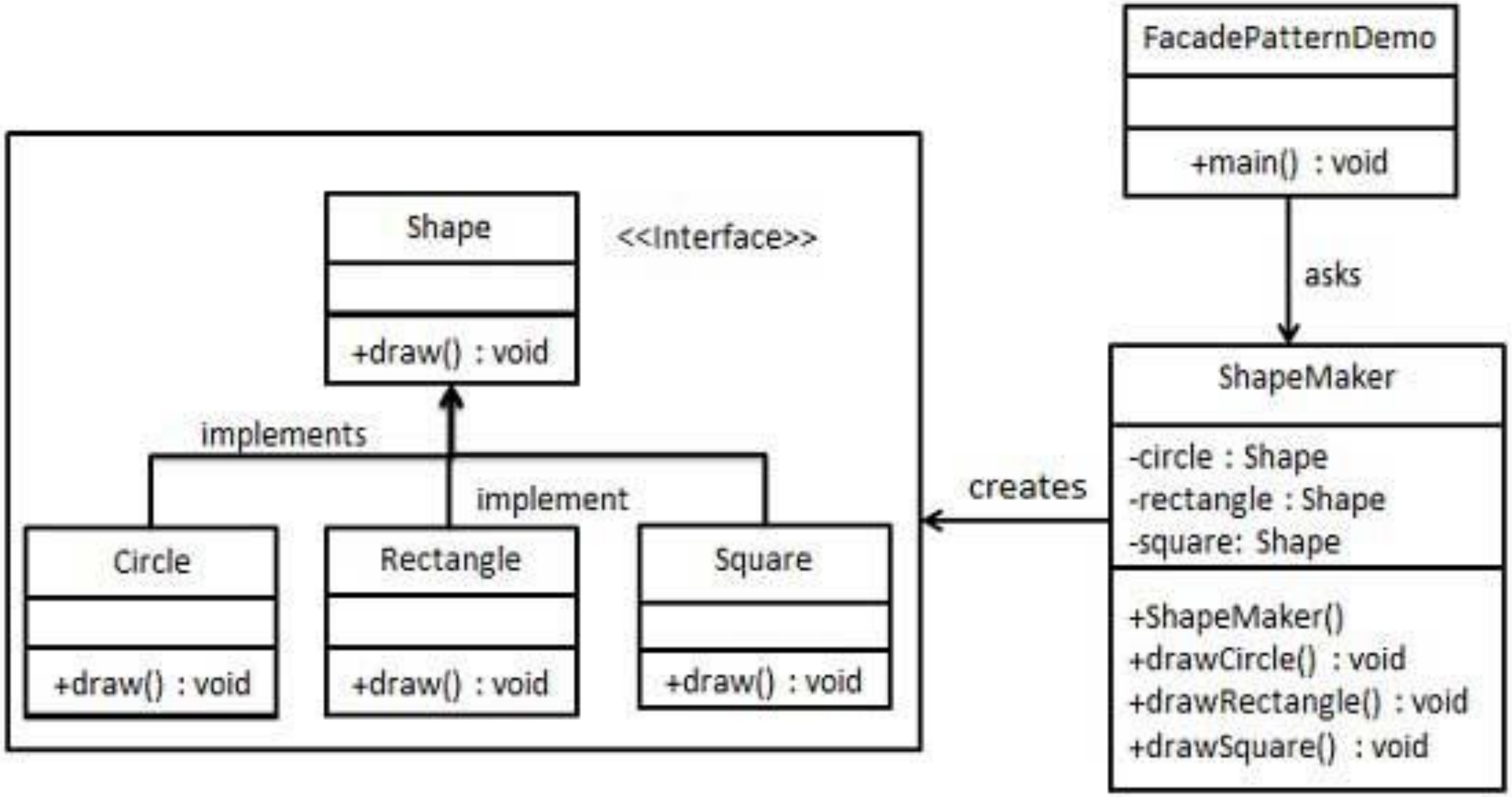
Façade Pattern - Structural

- *Provide a unified interface to a set of interfaces in a subsystem.*
- Façade defines a higher-level interface that makes the subsystem easier to use.
- The Operating System is a façade over the inner workings of the computer

Façade Pattern - Structural

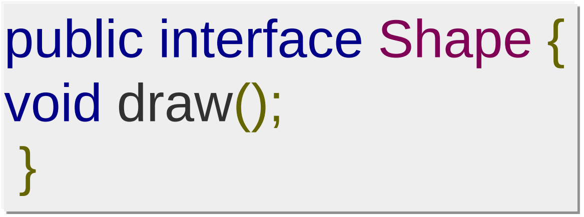
- Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system.
- This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.
- This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

Façade Pattern - Structural



Step 1

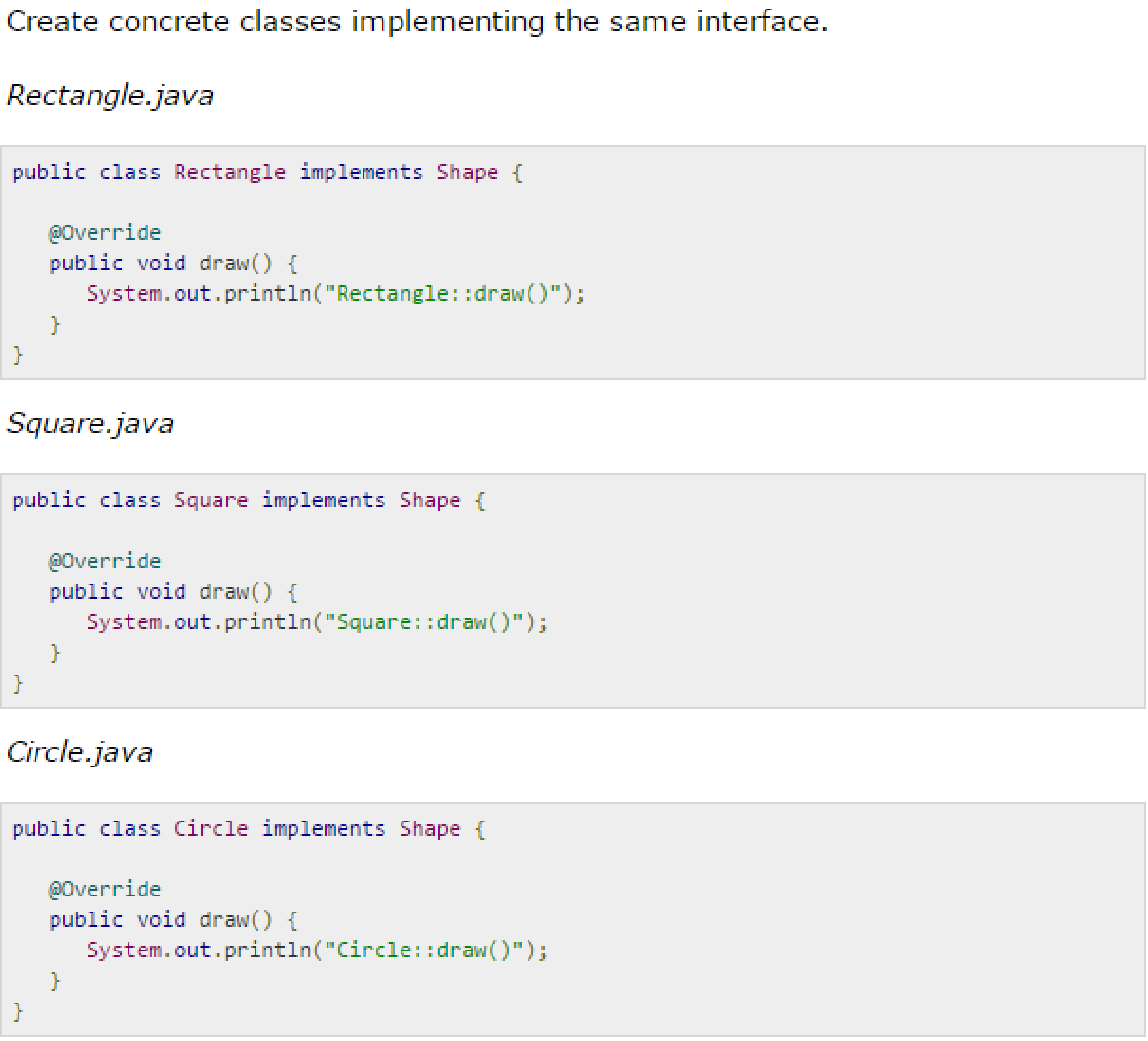
- ☞ Create an interface.
  - – *Shape.java*



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

Step 2

☞ Create concrete classes implementing the same interface.



Create concrete classes implementing the same interface.

Rectangle.java

```java
public class Rectangle implements Shape {

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

Square.java

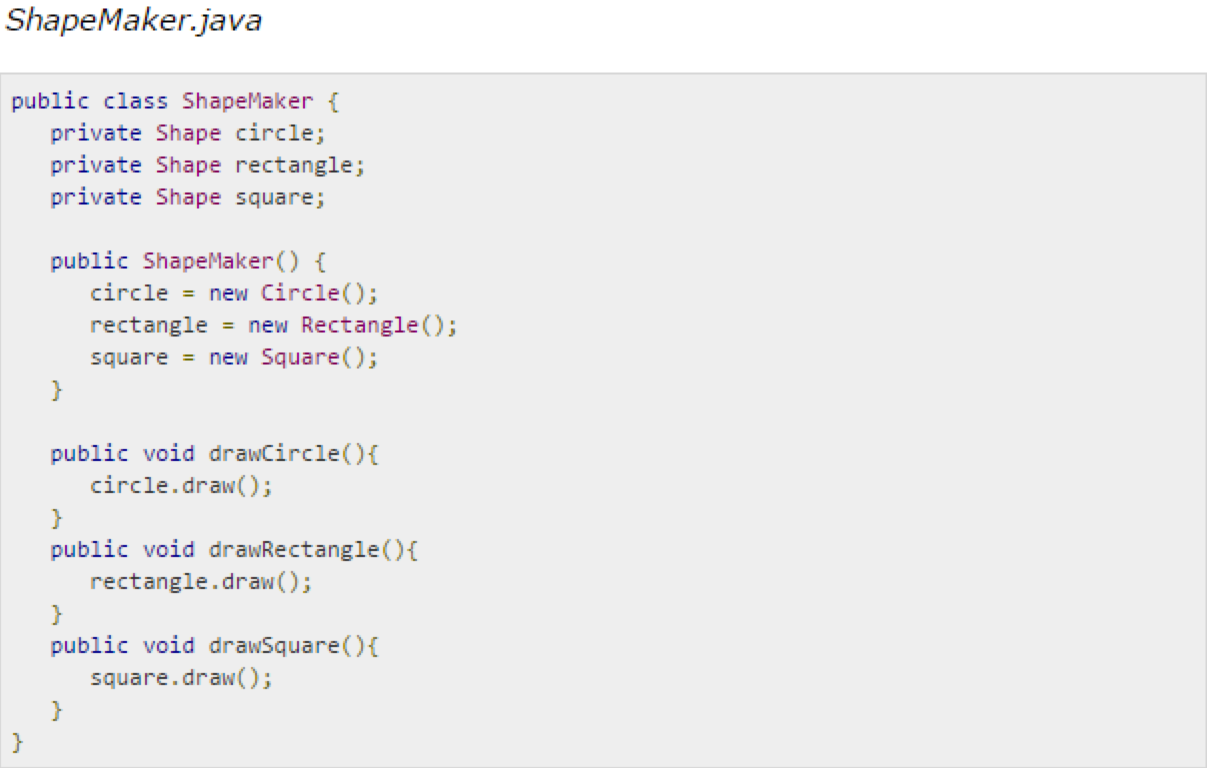
```java
public class Square implements Shape {

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

Circle.java

```java
public class Circle implements Shape {

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

Step 3

- ☞ Create a facade class.

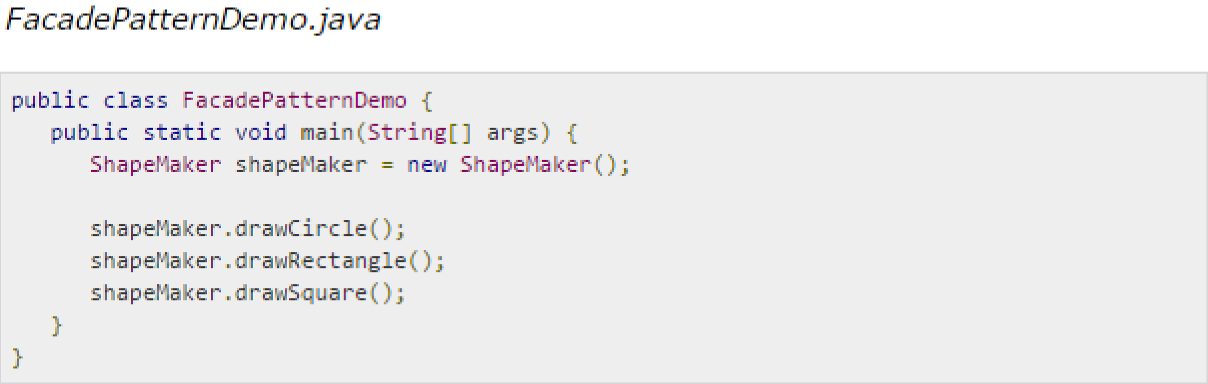


ShapeMaker.java

```java
public class ShapeMaker {
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }

   public void drawCircle(){
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
}
```

Step 4

- Use the facade to draw various types of shapes.



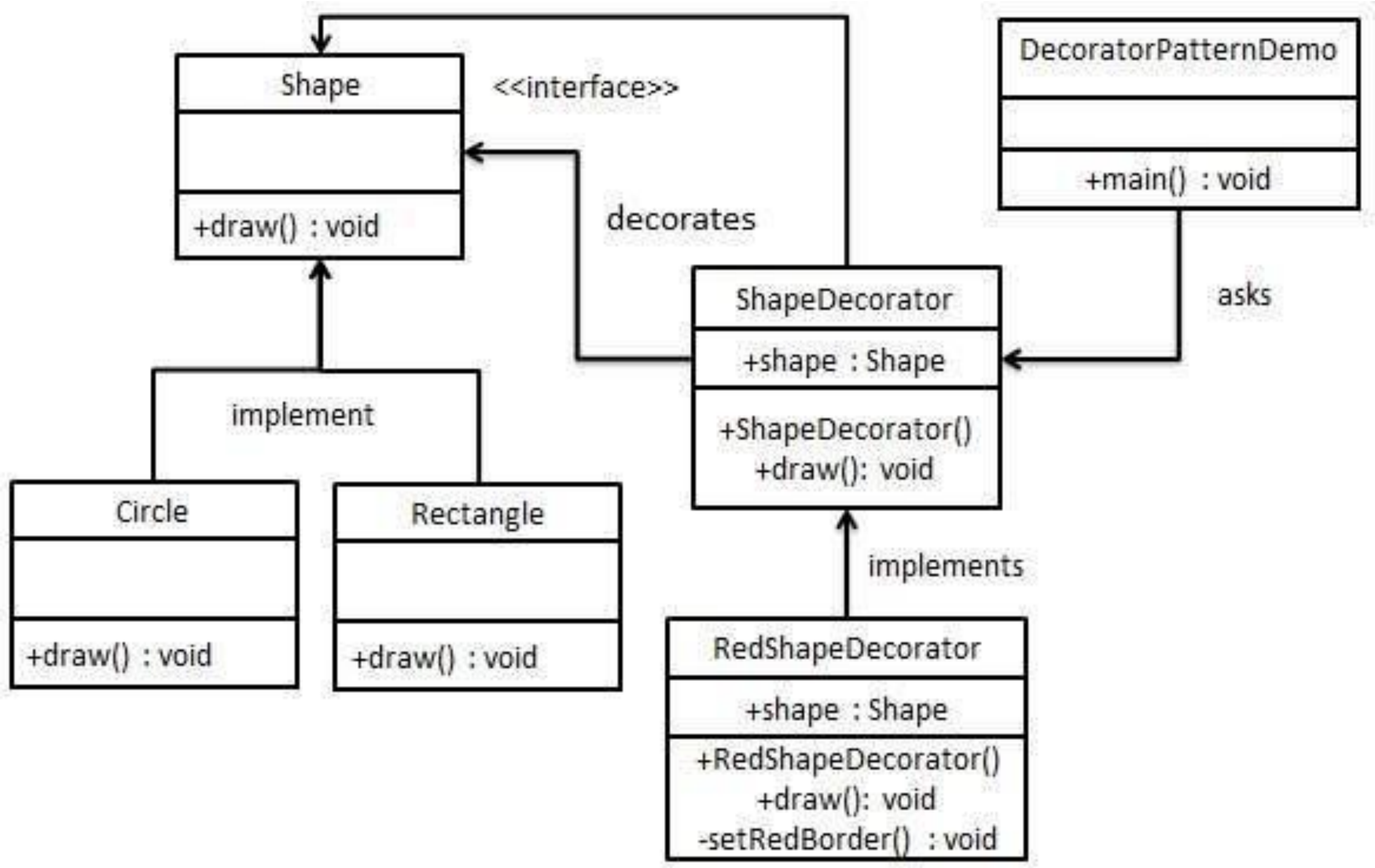
FacadePatternDemo.java

```
public class FacadePatternDemo {
   public static void main(String[] args) {
      ShapeMaker shapeMaker = new ShapeMaker();

      shapeMaker.drawCircle();
      shapeMaker.drawRectangle();
      shapeMaker.drawSquare();
   }
}
```

Decorator Pattern - Structural

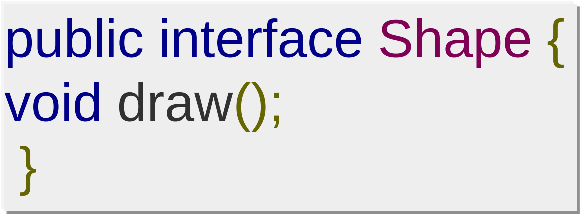
- *Allows for the dynamic wrapping of objects in order to modify their existing responsibilities and behaviours*
- New state and behaviour can be added to an existing class
- Decorator pattern allows a user to add new functionality to an existing object without altering its structure.

Decorator Pattern - Structural



Step 1

- Create an interface.
  - *Shape.java*

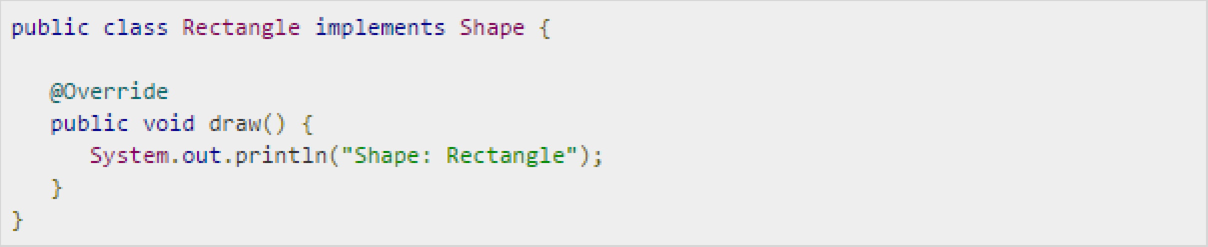


```
public interface Shape {
void draw();
 }
```

Step 2

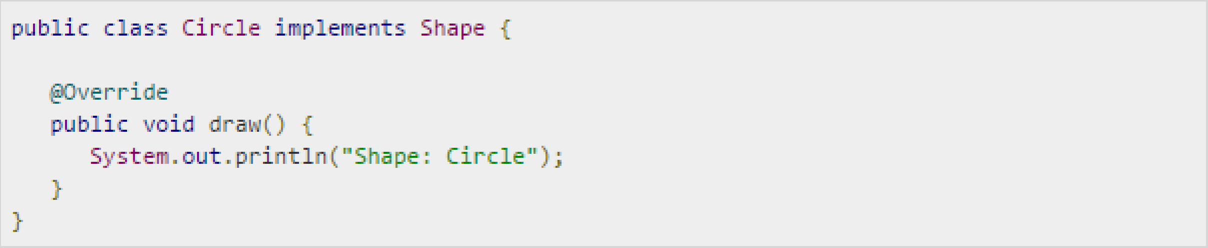
- Create concrete classes implementing the same interface.

Rectangle.java



```
public class Rectangle implements Shape {

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

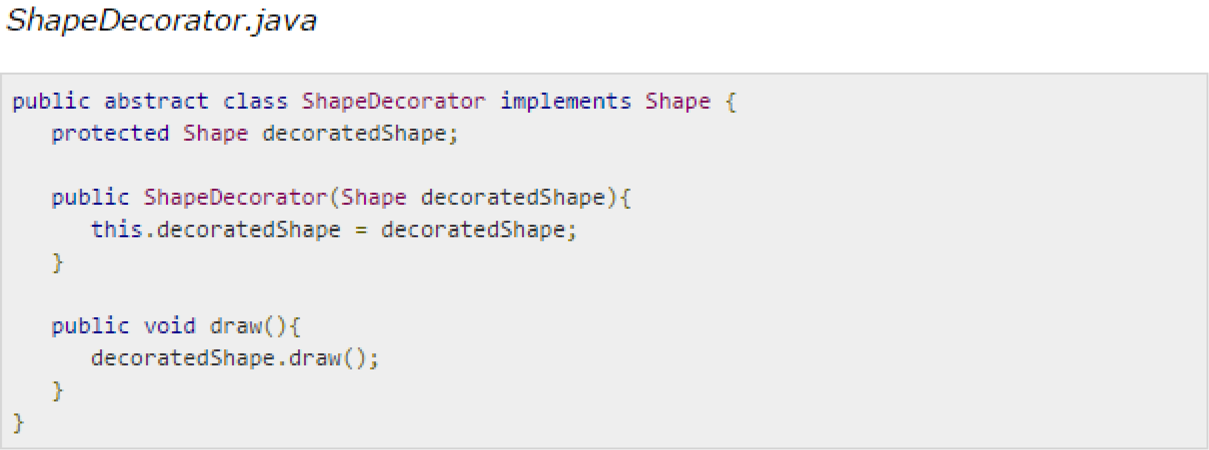
Circle.java



```
public class Circle implements Shape {

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

Step 3

- ☞ Create abstract decorator class implementing the *Shape* interface.

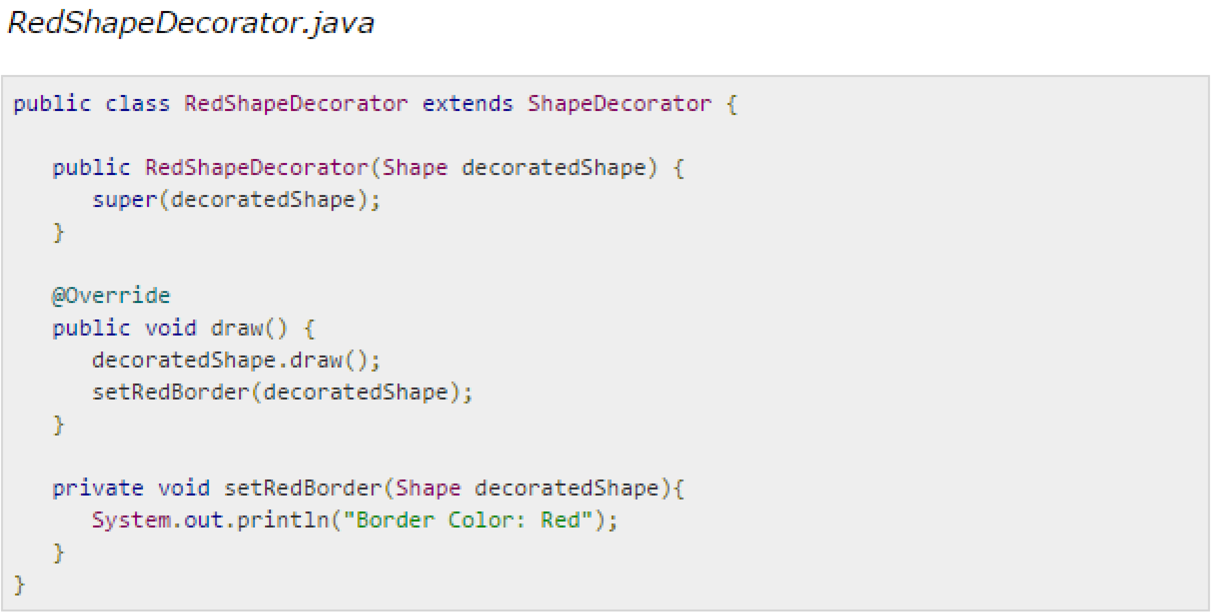


ShapeDecorator.java

```java
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

Step 4

- Create concrete decorator class extending the *ShapeDecorator* class.

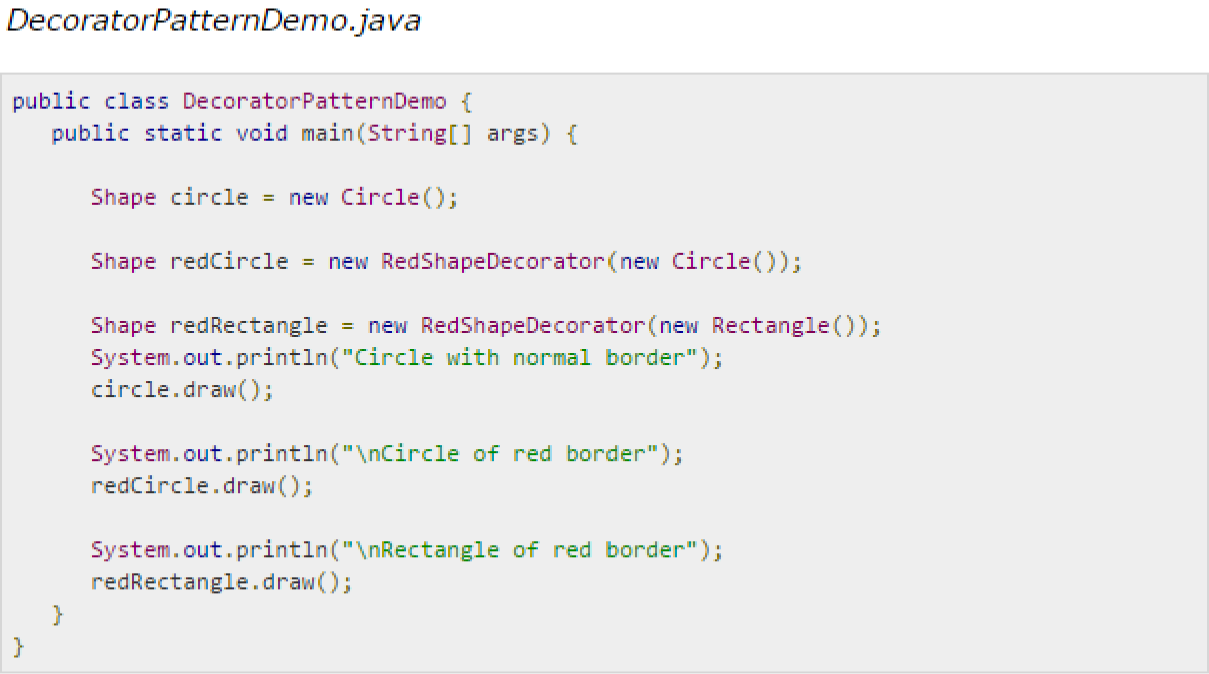


RedShapeDecorator.java

```
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

Step 5

- ☞ Use the *RedShapeDecorator* to decorate *Shape* objects.



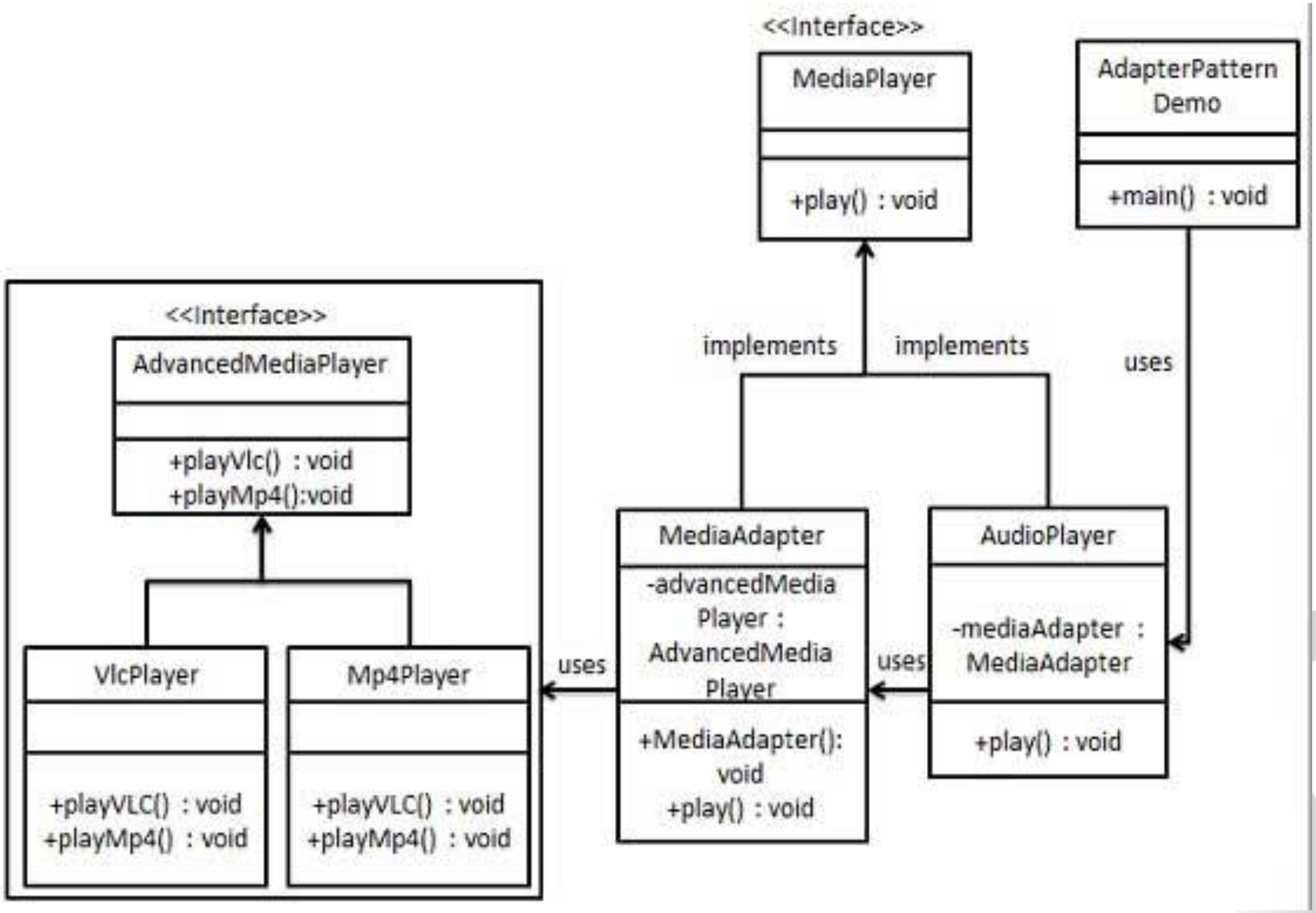
DecoratorPatternDemo.java

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

      Shape circle = new Circle();

      Shape redCircle = new RedShapeDecorator(new Circle());

      Shape redRectangle = new RedShapeDecorator(new Rectangle());
      System.out.println("Circle with normal border");
      circle.draw();

      System.out.println("\nCircle of red border");
      redCircle.draw();

      System.out.println("\nRectangle of red border");
      redRectangle.draw();
   }
}
```

Adapter Pattern - Structural

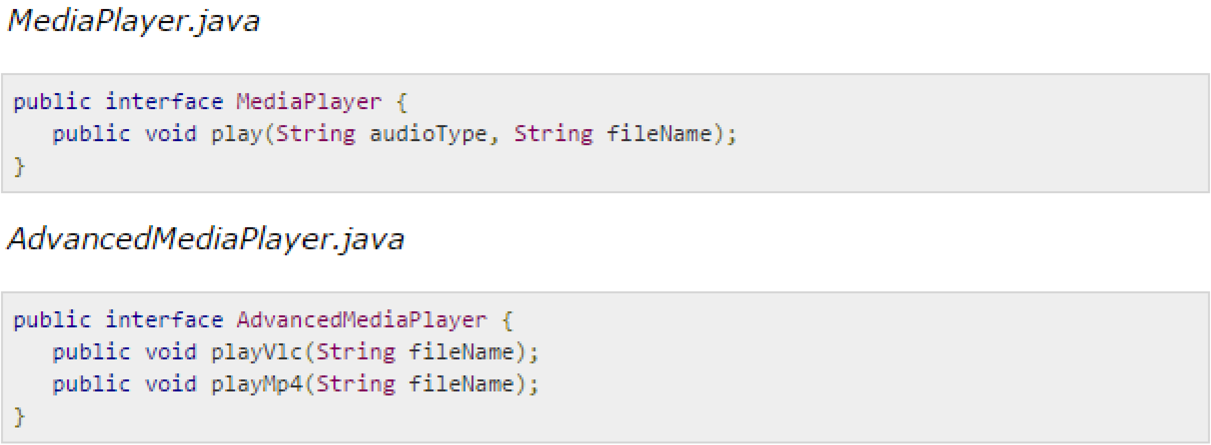
- *Convert the interface of a class into another interface clients expect.*
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- When you visit Europe with North American electrical devices you need an adapter

Adapter Pattern - Structural



Step 1

- Create interfaces for Media Player and Advanced Media Player.



MediaPlayer.java

```
public interface MediaPlayer {
   public void play(String audioType, String fileName);
}
```

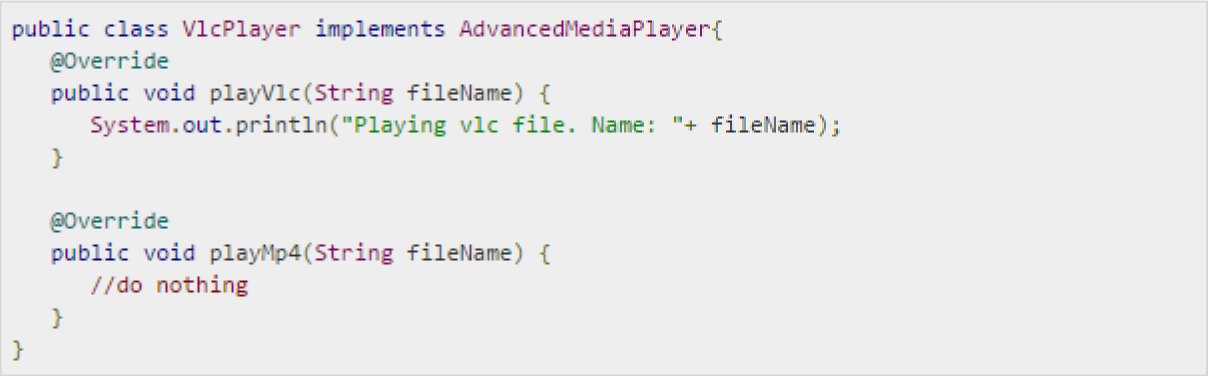
AdvancedMediaPlayer.java

```
public interface AdvancedMediaPlayer {
   public void playVlc(String fileName);
   public void playMp4(String fileName);
}
```

Step 2

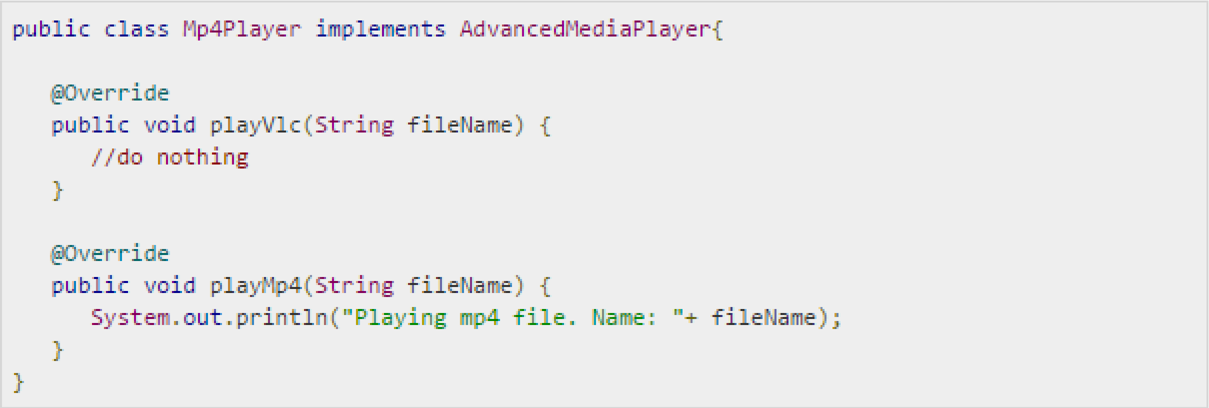
- Create concrete classes implementing the *AdvancedMediaPlayer* interface.

VlcPlayer.java



```
public class VlcPlayer implements AdvancedMediaPlayer{
   @Override
   public void playVlc(String fileName) {
      System.out.println("Playing vlc file. Name: "+ fileName);
   }

   @Override
   public void playMp4(String fileName) {
      //do nothing
   }
}
```

Mp4Player.java

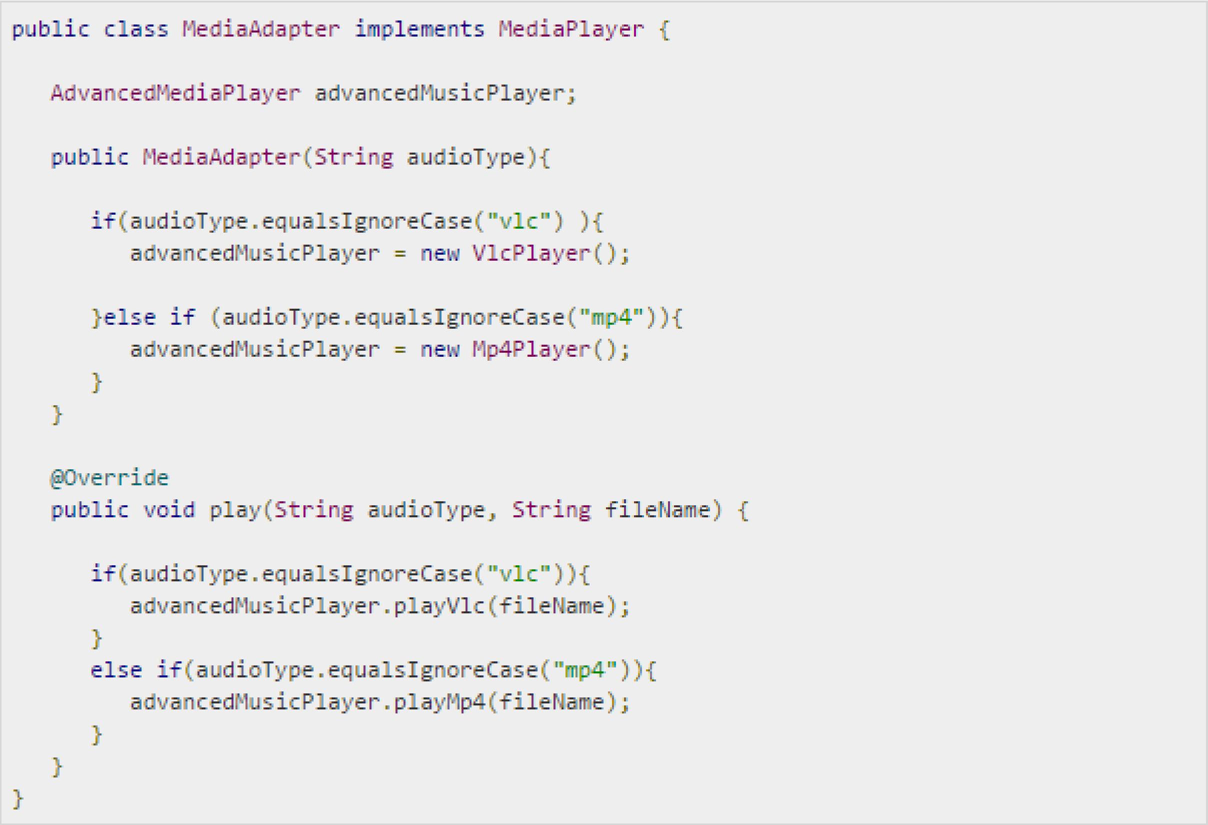


```
public class Mp4Player implements AdvancedMediaPlayer{

   @Override
   public void playVlc(String fileName) {
      //do nothing
   }

   @Override
   public void playMp4(String fileName) {
      System.out.println("Playing mp4 file. Name: "+ fileName);
   }
}
```

Step 3

- Create adapter class implementing the *MediaPlayer* interface.

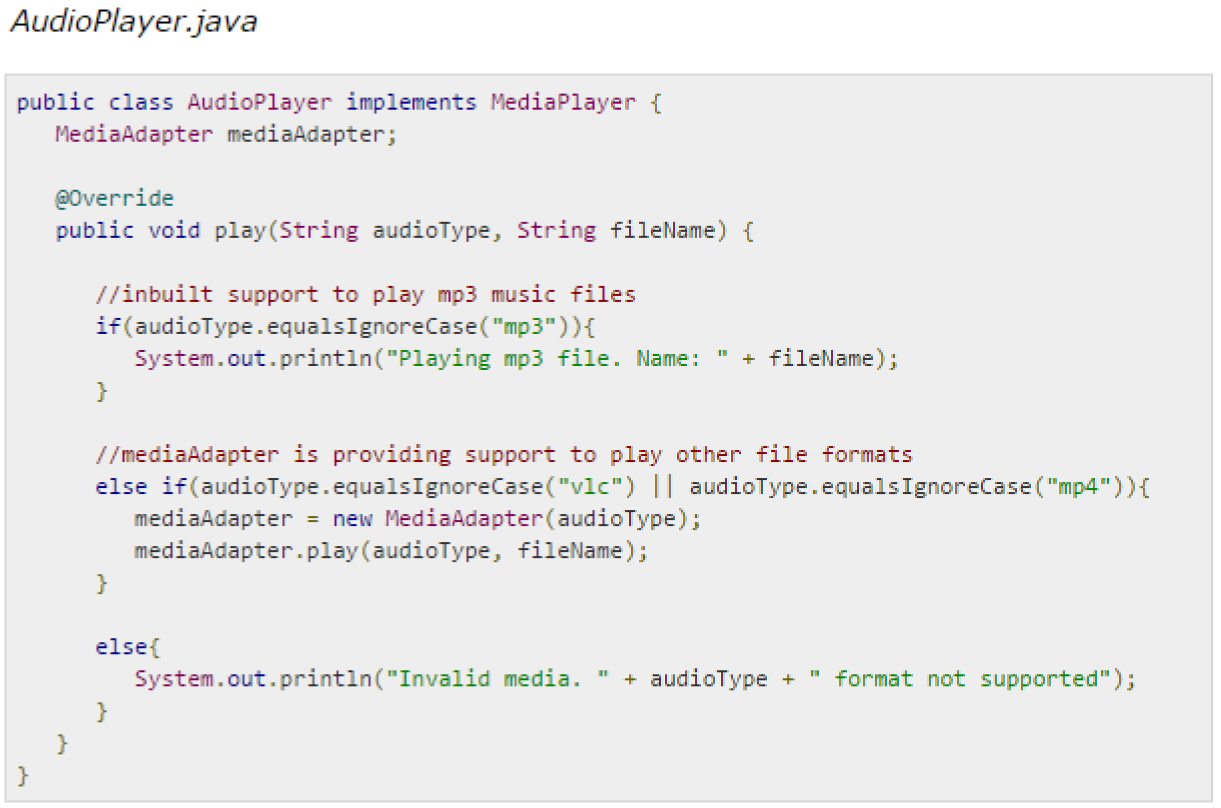
MediaAdapter.java



```
public class MediaAdapter implements MediaPlayer {

   AdvancedMediaPlayer advancedMusicPlayer;

   public MediaAdapter(String audioType){

      if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();

      }else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
      }
   }

   @Override
   public void play(String audioType, String fileName) {

      if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      }
      else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
   }
}
```

Step 4

☞ Create concrete class implementing the *MediaPlayer* interface.

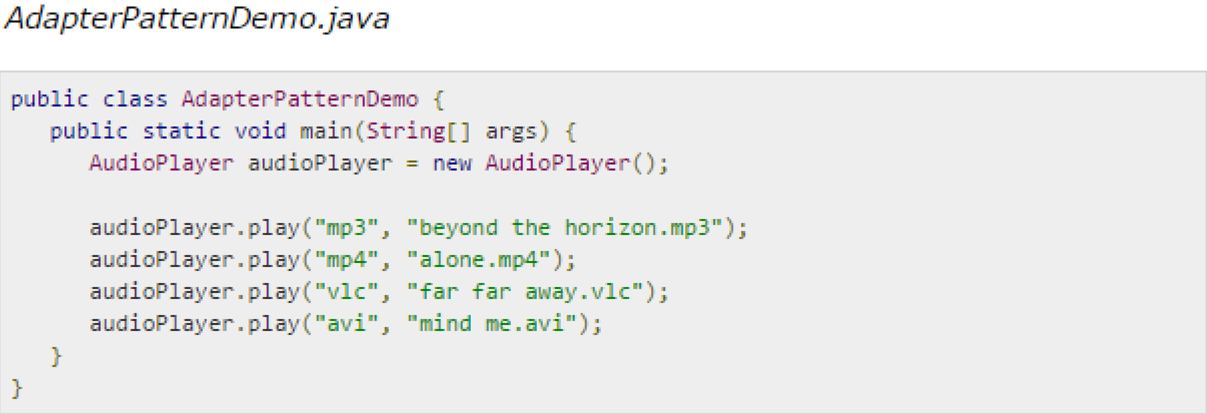


AudioPlayer.java

```
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;

   @Override
   public void play(String audioType, String fileName) {

      //inbuilt support to play mp3 music files
      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: " + fileName);
      }

      //mediaAdapter is providing support to play other file formats
      else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter.play(audioType, fileName);
      }

      else{
         System.out.println("Invalid media. " + audioType + " format not supported");
      }
   }
}
```

Step 5

☞ Use the AudioPlayer to play different types of audio formats.



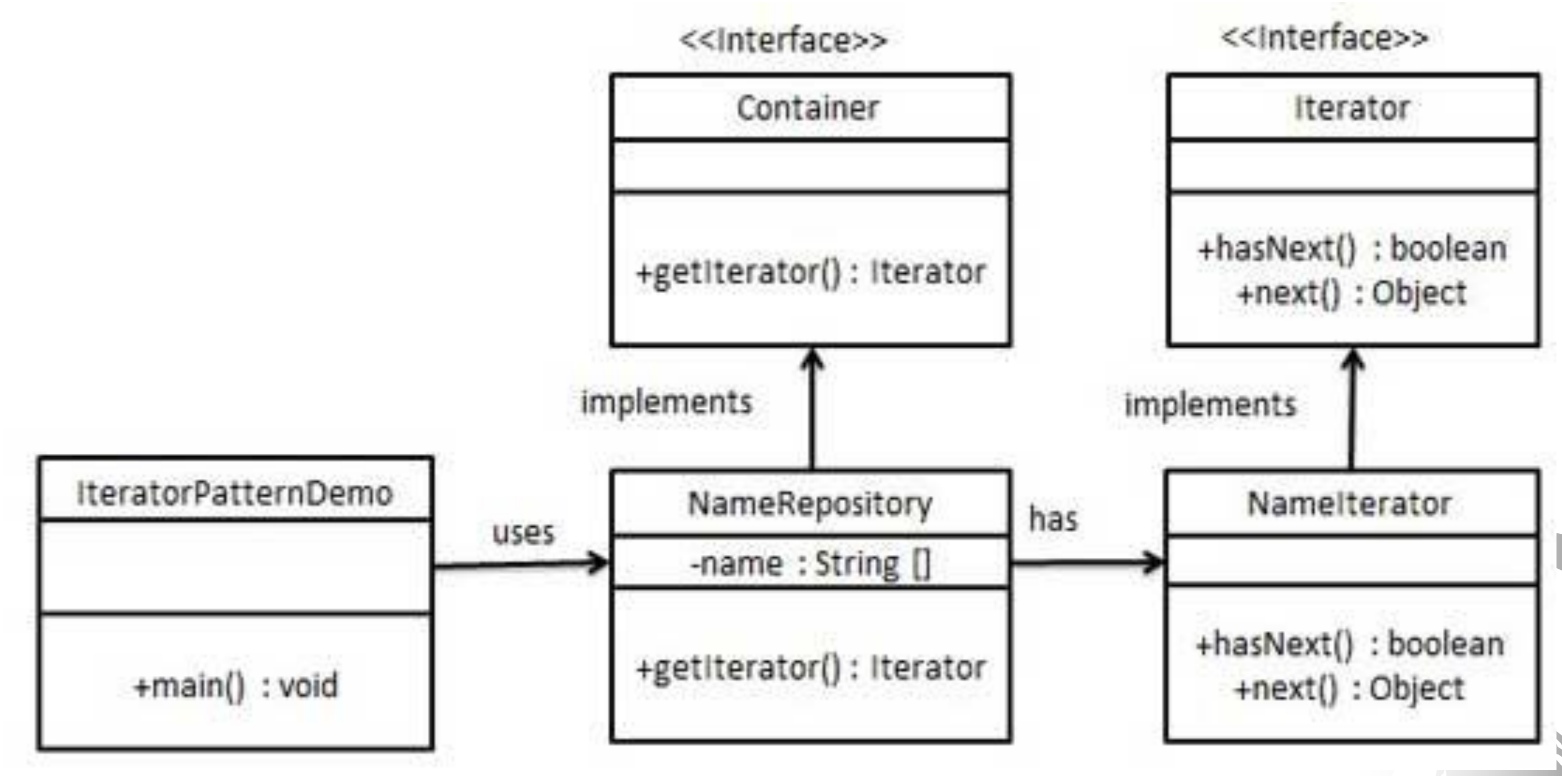
AdapterPatternDemo.java

```
public class AdapterPatternDemo {
   public static void main(String[] args) {
      AudioPlayer audioPlayer = new AudioPlayer();

      audioPlayer.play("mp3", "beyond the horizon.mp3");
      audioPlayer.play("mp4", "alone.mp4");
      audioPlayer.play("vlc", "far far away.vlc");
      audioPlayer.play("avi", "mind me.avi");
   }
}
```

Iterator Pattern - Behavioural

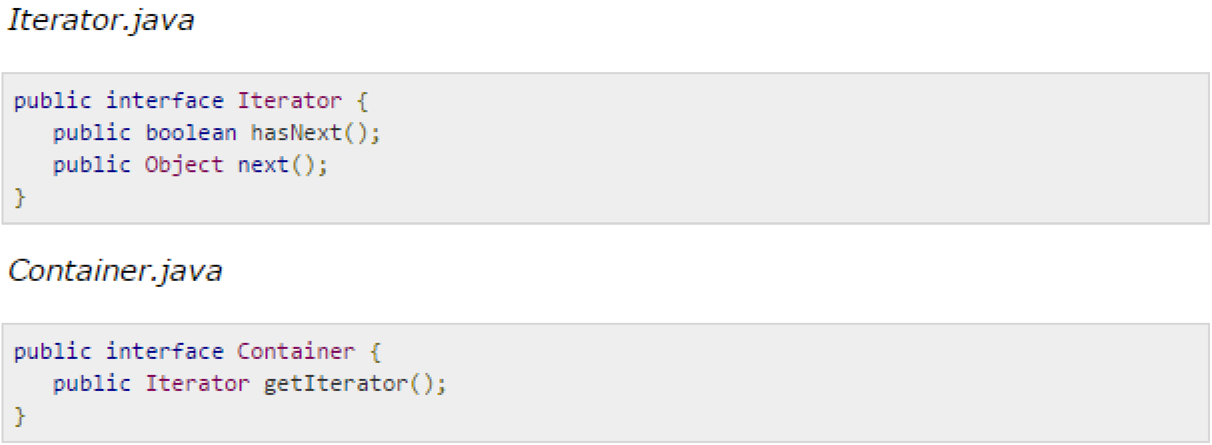
- *Provides a way to access the elements of an aggregate object without exposing its underlying representation.*
- Collections can create iterator objects
- These objects allow you to move through the collection without concern for how the collection is coded

Iterator Pattern - Behavioural



Step 1

☞ Create interfaces.



Iterator.java

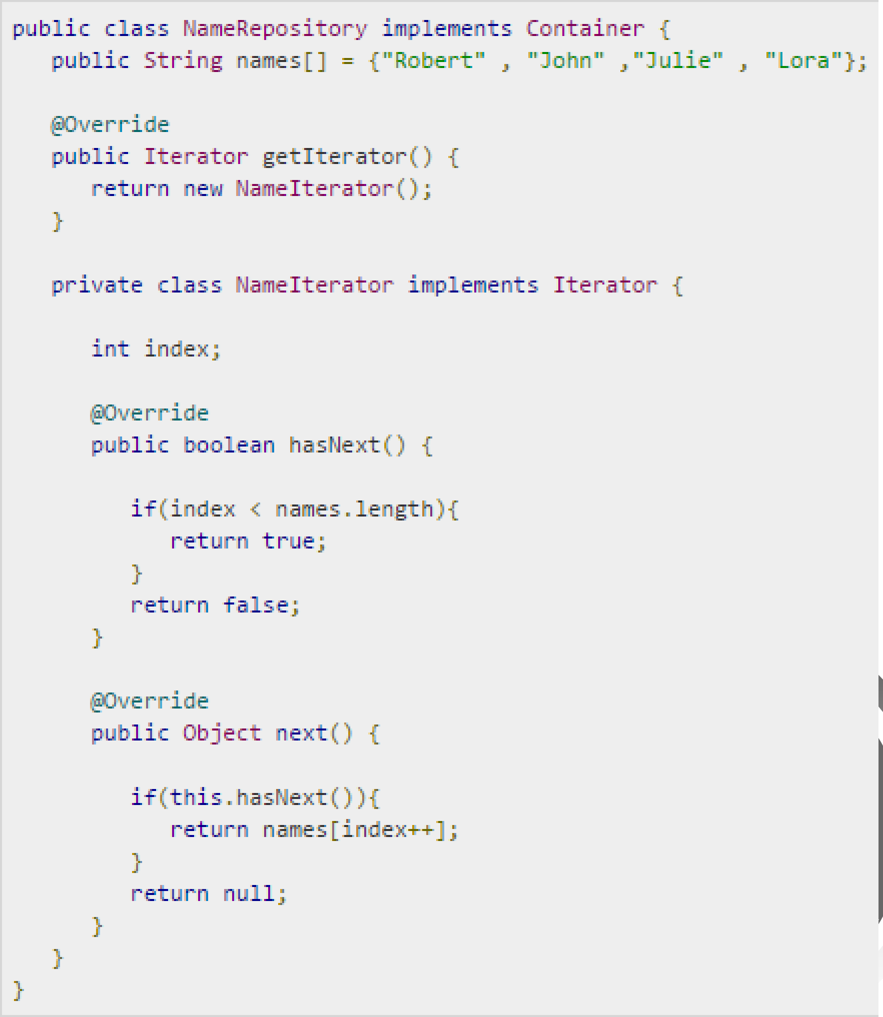
```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}
```

Container.java

```
public interface Container {
   public Iterator getIterator();
}
```

Step 2

- Create concrete class implementing the *Container* interface. This class has inner class *NameIterator* implementing the *Iterator* interface.

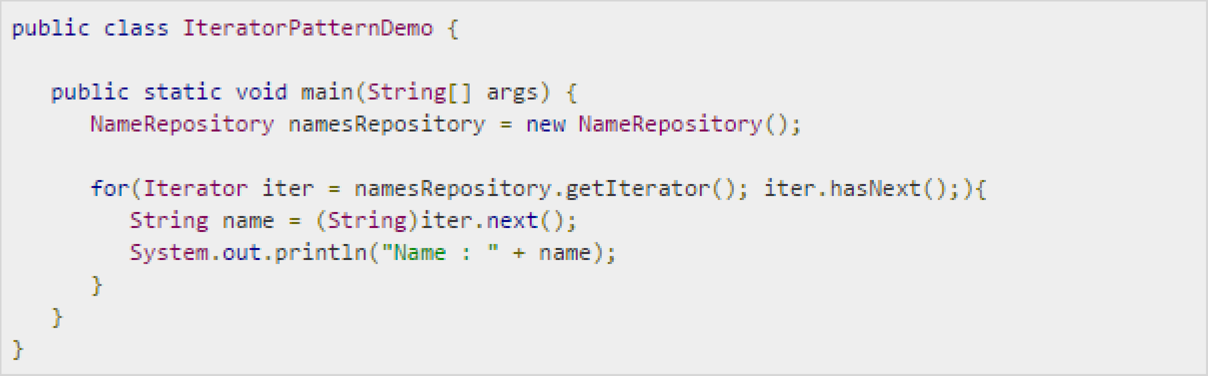


```java
public class NameRepository implements Container {
   public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

   @Override
   public Iterator getIterator() {
      return new NameIterator();
   }

   private class NameIterator implements Iterator {

      int index;

      @Override
      public boolean hasNext() {

         if(index < names.length){
            return true;
         }
         return false;
      }

      @Override
      public Object next() {

         if(this.hasNext()){
            return names[index++];
         }
         return null;
      }
   }
}
```

Step 3

- Use the *NameRepository* to get iterator and print names.

IteratorPatternDemo.java



```
public class IteratorPatternDemo {

   public static void main(String[] args) {
      NameRepository namesRepository = new NameRepository();

      for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
         String name = (String)iter.next();
         System.out.println("Name : " + name);
      }
   }
}
```

Bibliography

- http://www.tutorialspoint.com/design_pattern/index.htm
- http://java.dzone.com/articles/design-patterns-overview
- http://refcardz.dzone.com/refcardz/design-patterns