

COMP2911 Notes

Object-Oriented Design

Object Oriented Programming and Abstract Data Types:

OOP:

- Processing is more distributed - control given to the objects themselves.
 - Harder to prove correctness.
- Allows for multiple class implementations.
- Can extend classes by inheritance.
- Easier to add new operations.

ADT:

- One flow of control.
 - Easier to prove correctness.
- Harder to add new operations.

Classes and Objects

- A class is a template for an object.
- An object is an “instance” of a class.

Encapsulation

- Encapsulation is the hiding of the internal implementation of an object.
- This is done by accessing the object through its methods.

Abstraction

- Abstraction is the focussing of the essential characteristic of an object, which enables it to be distinguished from other kinds of objects.
- Ignore the irrelevant.

Inheritance

- Inheritance is when a class “inherits” the properties of another class, which it can then extend on.
- The class that inherits the properties is known as the “subclass”
- The class that the subclass inherits from is known as the “superclass”
- In java:
 - A subclass is made using the “extends” keyword.
 - To refer to the superclass methods, the “super” keyword is used.
 - ```
public String toString()
{ return super.toString() + “[hireDate = “ + hireDate + ”]” ; }
```

## Dynamic Binding

Dynamic binding means that a block of code executed with reference to a method call is determined at run time. It is generally used when multiple classes contain different implementations of the same method.

```
1 /**
2 * Human Class
3 */
4 class Human {
5 public void walk() {
6 System.out.println("Human walks");
7 }
8 }
9
10 /**
11 * Boy - Subclass of Human
12 */
13 class Boy extends Human {
14 public void walk() {
15 System.out.println("Boy walks");
16 }
17 }
18
19 /**
20 * Main Class that
21 */
22 public class DynamicBinding {
23
24 public static void main(String args[]) {
25 // Human = "Human walks"
26 // Boy = "Boy walks"
27 // What is printed below ?
28
29 Human myobj = new Boy();
30 myobj.walk();
31 }
32 }
33
```

The output of the above program is: "Boy Walks"

## Liskov Substitution Principle (LSP)

- A subclass must honour the contract made by its parent class.
- Java version: A method of class t, "should work", when called on an object, that is a subclass of t.
- This principle came about to help prove object oriented programs were valid.
- E.g. Can 'Square' extend 'Rectangle'?
  - No, 'setHeight' and 'setWidth' work differently for a square.

## Mutable and Immutable Objects

- Mutable objects have fields that can be changed after creation.
- Immutable objects have no fields that can be changed after creation.

## Method - toString

- Due to inheritance, a subclass may have a different name/field to print. Therefore, it is made more general by using expressions like:
  - getClass.getName()

## Method - equals

- Subclasses can cause confusion with checking if objects are equal. Therefore, always start equality checks, by checking the class itself.

```
public boolean equals(Object other) {

 if (getClass() != other.getClass()) return false;
 // Cast object to correct type
 // Check fields are equal
 return true;
}
```

- '==' compares object references, not contents.
- '.equals' compares object contents.

## Method - clone

- The class must implement 'Cloneable'
- A try-catch block is used to catch the 'CloneNotSupportedException'
- Calling super.clone() saves the hassle of having to create new objects and setting the fields.

```
public Class clone() {
 try {
 Class cloned = (Class) super.clone();
 return cloned;
 }

 catch (CloneNotSupportedException e) {
 return null;
 }
}
```

- For a subclass:
  - Call super.clone()
  - For all fields not set by the superclass, set the fields to be clones of the original object.
    - E.g. cloned.hireDate = (Calendar) hireDate.clone()

## Object-Oriented Design Process

### Use Cases:

- A sequence of steps (done by the user or system) that describe an interaction with a system.
- Enrolment system example:
  1. System shows list of course
  2. User selects a course
  3. User asks the system to enrol in course
  4. System checks the prerequisites are met for the course
  5. System allocates sessions to the user
  6. System displays sessions to the user.

CRC Cards:

- Stands for: Class-Responsibility-Collaborator
- Class:
  - The classes that make up the system.
- Responsibility:
  - Explains what the class will have to do.
  - Language doesn't imply methods.
- Collaborator:
  - What class interactions are needed to perform responsibilities.
- Enrolment system example (draft):

| EnrolmentSystem (Class)                                                                                                                               |                                                                                                          |
|-------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|
| <b>Responsibilities</b> <ul style="list-style-type: none"><li>• Maintains a list of courses on offer</li><li>• Maintains a list of students</li></ul> | <b>Collaborators</b> <ul style="list-style-type: none"><li>• Course Offering</li><li>• Student</li></ul> |

| CourseOffering (Class)                                                                                                                              |                                                                                                                     |
|-----------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| <b>Responsibilities</b> <ul style="list-style-type: none"><li>• Enrol students for this particular course offering in different sessions.</li></ul> | <b>Collaborators</b> <ul style="list-style-type: none"><li>• Course</li><li>• Students</li><li>• Sessions</li></ul> |

| Course (Class)                                                                                                                                                       |                                                                               |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| <b>Responsibilities</b> <ul style="list-style-type: none"><li>• Manages course details - name, description, etc.</li><li>• Manages prerequisites of course</li></ul> | <b>Collaborators</b> <ul style="list-style-type: none"><li>• Course</li></ul> |

| Student (Class)                                                                                    |                                                                                  |
|----------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------|
| <b>Responsibilities</b> <ul style="list-style-type: none"><li>• Manages their enrolments</li></ul> | <b>Collaborators</b> <ul style="list-style-type: none"><li>• Enrolment</li></ul> |

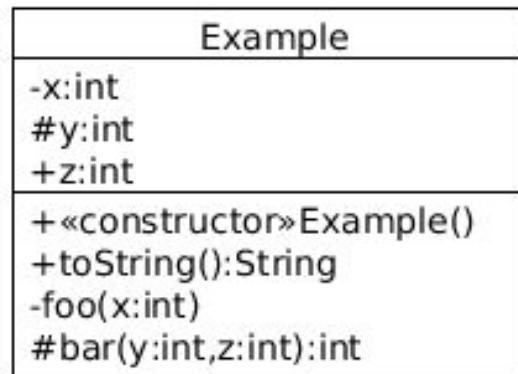
| Enrolment (Class)                                                                                                                 |                                                                                 |
|-----------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------|
| <b>Responsibilities</b> <ul style="list-style-type: none"><li>• Maintains a list of offerings a students is enrolled in</li></ul> | <b>Collaborators</b> <ul style="list-style-type: none"><li>• Offering</li></ul> |

### Walkthrough:


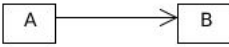

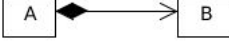
- Tries to work out how the classes will interact.
- Enrollment system example:
  1. 'Enrolment System' displays a list of 'Course Offerings'
  2. User selects a 'Course Offering'
  3. 'Enrolment System' asks 'Course Offering' to enrol a 'Student'
  4. 'Course Offering' gets prerequisites from 'Course'
  5. 'Course Offering' asks 'Student' if prerequisites are passed
  6. 'Course Offering' asks 'Sessions' to add 'Student'
  7. 'Student' creates new 'Enrolment' object

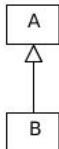
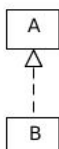
### UML Class Diagrams:

- How classes are represented:

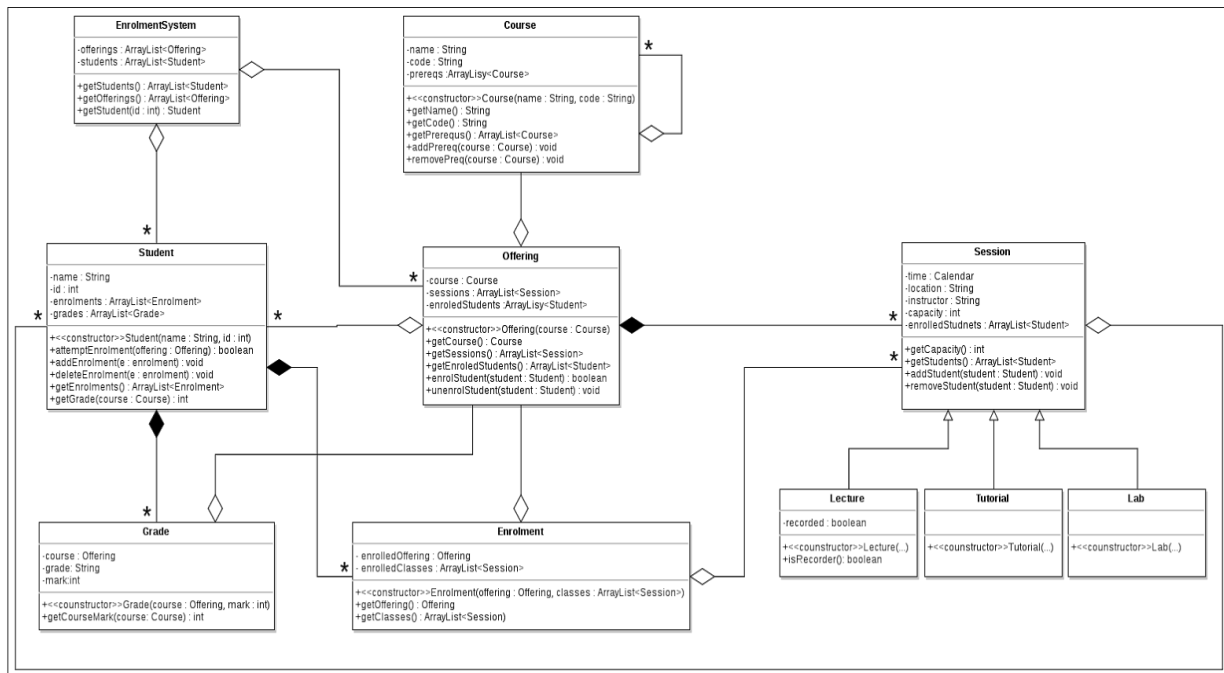


- Relationships between classes:

| Relationship | Depiction                                                                           | Interpretation                                                                                                                                                                                                     |
|--------------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Dependency   |  | A depends on B<br>This is a very loose relationship and so I rarely use it, but it's good to recognize and be able to read it.                                                                                     |
| Association  |  | An A sends messages to a B<br>Associations imply a direct communication path. In programming terms, it means instances of A can call methods of instances of B, for example, if a B is passed to a method of an A. |
| Aggregation  |  | An A is made up of B<br>This is a part-to-whole relationship, where A is the whole and B is the part. In code, this essentially implies A has fields of type B.                                                    |
| Composition  |  | An A is made up of B with lifetime dependency<br>That is, A aggregates B, and if the A is destroyed, its B are destroyed as well.                                                                                  |

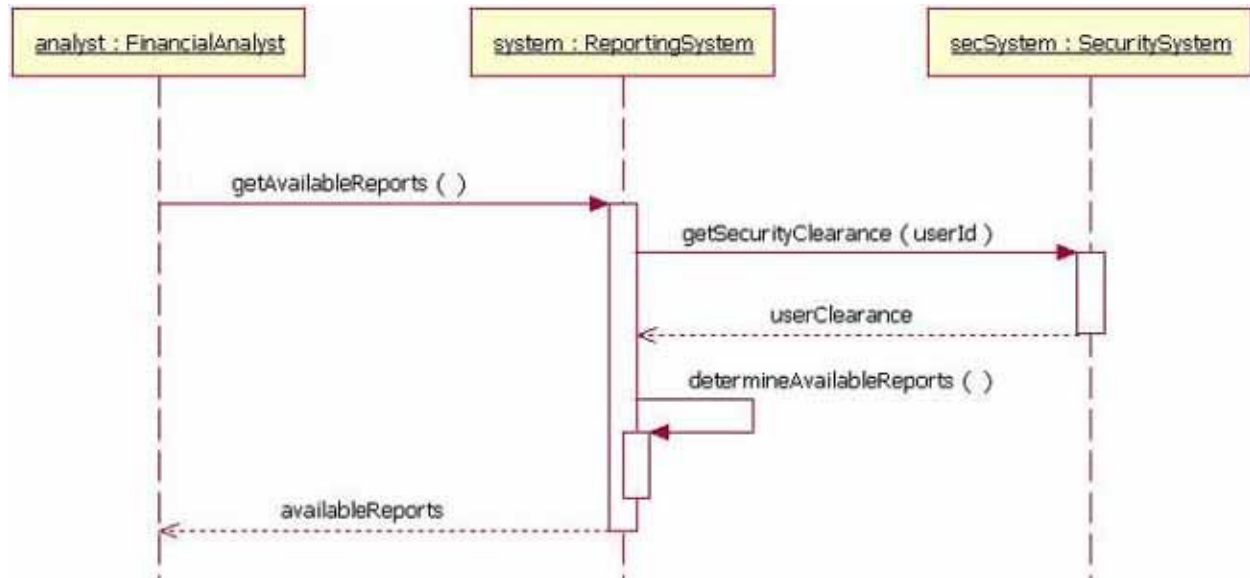
| Relationship   | Depiction                                                                         | Interpretation                                                                                                                                                                                                                                  |
|----------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Generalization |  | <p>A generalizes B<br/>Equivalently, B is a subclass of A. In Java, this is extends.</p>                                                                                                                                                        |
| Realization    |  | <p>B realizes (the interface defined in) A<br/>As the parenthetical name implies, this is used to show that a class realizes an interface. In Java, this is implements, and so it would be common for A to have the «interface» stereotype.</p> |

- Multiplicity explains how many of each object is involved in a relationship. When multiplicity is not stated, “1” is assumed. Otherwise the values are explicitly defined as:
  - A constant - ‘1’
  - Unbounded (zero or more) - ‘\*’
  - A range - ‘2..\*’
- Law of Demeter - Objects should assume as little as possible about the structure or properties of anything else. Basically, avoid triangles with dependencies in UML diagrams and multiple ‘dots’ in code. E.g. o.get(name).get(thing).remove(node)
- Enrolment system example:



## UML Sequence Diagrams:

- It shows the interactions between classes with method calls, helping to validate use cases and walkthroughs.
- Example:



## Programming By Contract

### Preconditions

- Conditions that a user ensures are met before a method call.
- The user must be able to check the precondition themselves.
  - Other methods might be needed like getters.
- The programmer should not check the preconditions.

### Postconditions

- Conditions that a method guarantees to be true, if the preconditions are satisfied.

### Invariants

- Conditions on an object state.
- These conditions hold after constructor use, before each method call and after each method call. They may not hold during method execution.
- We can prove the invariant is true using preconditions and postconditions.

## Covariance

- A subclass should not have weaker postconditions than its parent class.

## Contravariance

- A subclass should not have stronger preconditions than its parent class.

## Example

```
/**
 * Simple class to implement a bank account
 * @invariant balance >= 0
 */
public class BankAccount {

 private int balance;

 /**
 * Constructor
 */
 public BankAccount(){
 this.balance = 0;
 }

 /**
 * Deposit amount into bank account
 * @param amount
 * @pre amount > 0
 * @post newBalance > oldBalance
 */
 public void deposit(int amount){
 balance += amount;
 }

 /**
 * Withdraw amount from bank account
 * @param amount
 * @pre (amount > 0) and (balance >= amount)
 */
 public void withdraw(int amount){
 balance -= amount;
 }

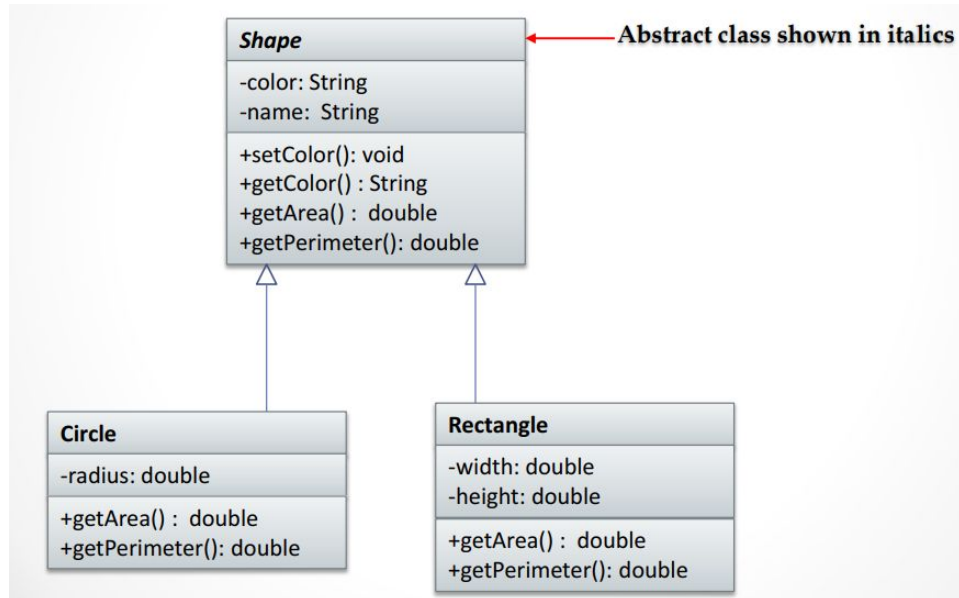
 /**
 * Returns account balance
 * @return balance
 */
 public int getBalance(){
 return balance;
 }
}
```



# Generic Types and Polymorphism

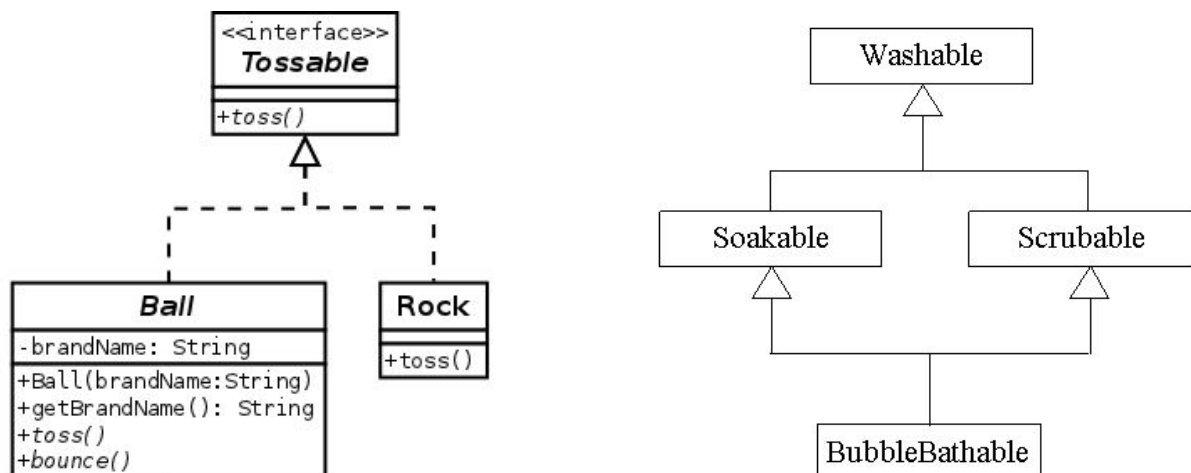
## Abstract Class

- A class that cannot be instantiated and serves as a base type for subclasses.
- *Note: This is the opposite of concrete subclasses, which have instances*
- Example:



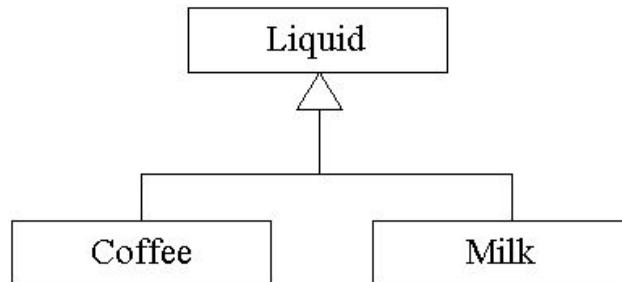
## Interface

- A collection of abstract methods, i.e. defined methods without a body.
- A class implements an interface - provides implementation for all interface methods.
- A class can implement multiple interfaces (a sort of multiple inheritance).
- Note that interfaces can also extend each other (right).



## Polymorphism

- Polymorphism is the ability to appear in many forms.
- In Java - the ability to treat a subclass object, as if it were the object of the base class. The behaviour is determined at runtime by dynamic binding.
- So although the code might think it is sending a message to a base class, it could be sending the message to any of its subclasses.
- Simple example:



```
Liquid favouriteDrink = new Coffee();
Liquid secondFavouriteDrink = new Milk();

favouriteDrink.drink();
secondFavouriteDrink.pour();
```

- The syntax 'instanceof', can be used to check which liquid the object refers to:

```
if (favouriteDrink instanceof Coffee) {
 System.out.println("Favourite drink is coffee");
} else if (favouriteDrink instanceof Milk) {
 System.out.println("Favourite drink is milk");
}
```

- Redefining an instance method in a subclass is called “overriding” the instance method.
- If a new method is added to a subclass, it cannot be called if the subclass is treated as the superclass. In order for this to work, the 'instanceof' could be used to distinguish if the subclass is of the appropriate type and then downcasting used:

```
Liquid liquid = new Tea();

if (liquid instanceof Tea) {
 Tea tea = (Tea) liquid;
 tea.readFutureFromLeaves();
}
```

## Array and List Covariance

- Arrays are covariant - An array of type T may contain elements of type T or any subtype of T.
  - `Fruit[] f = new Apple[10];` No Problems
  - `f[0] = new Orange();` `ArrayStoreException`
- Lists are not covariant - A list of type T cannot contain elements that are not T, including subtypes of T. This is due to type parameters being discarded after compilation and so type information is not available during runtime.
  - `ArrayList<Fruit> f;`  
`ArrayList<Apple> a;`  
`f = a;` `Compilation error`  
`a = f;` `Compilation error`  
`f.add(new Apple());` No Problems

## Generics

- Generics provide compile-time type checking and avoid the risk of `ClassCastException`.
- A generic type is a class or interface that is parameterised over types. Angle brackets <> are used to specify the type parameter.
- Types are usually given the generic parameters of E, T, etc.
- The '?' type is the wildcard type. This allows:
  - `ArrayList<Apple> a = new ArrayList<Apple>();`  
`ArrayList<? extends Fruit> f = a;` No Problems  
`f.add(new Orange());` `Compilation error`
- Set example (not everything shown):

```
import java.util.*;

/**
 * @invariant all elements of the set are distinct
 * @param <E>
 */
public class ArrayListSet<E> implements Set<E> {

 //===== Fields =====//
 private ArrayList<E> setElements;

 //===== Constructors =====//
 public ArrayListSet(){
 setElements = new ArrayList<E>();
 }

 private ArrayListSet(ArrayListSet<E> set) {
 setElements = new ArrayList<E>(set.setElements);
 }
}
```

```

//===== Methods =====//
public Set<E> union(Set<E> secondSet){
 Set<E> newSet = this.clone();
 for (E element : secondSet) {
 newSet.add(element);
 }
 return newSet;
}

public Set<E> intersection(Set<E> secondSet) {
 Set<E> newSet = new ArrayListSet<E>();
 for (E element : secondSet) {
 if (this.contains(element)) {
 newSet.add(element);
 }
 }
 return newSet;
}

public boolean subset(Set<E> secondSet) {
 for (E element : secondSet) {
 if (!this.contains(element)) {
 return false;
 }
 }
 return true;
}

public boolean equals(Object o) {

 // Base cases
 if (this == o) return true;
 if (o == null) return false;

 // Check o is a set
 if (!(o instanceof Set)) return false;

 // Turn o into a set of something
 Set<?> other = (Set<?>) o;

 // Check sets are the same size
 if (size() != other.size()){
 return false;
 }

 // Check they have the same elements
 for (E element : setElements) {
 if (!other.contains(element)) {
 return false;
 }
 }
 return true;
}

```

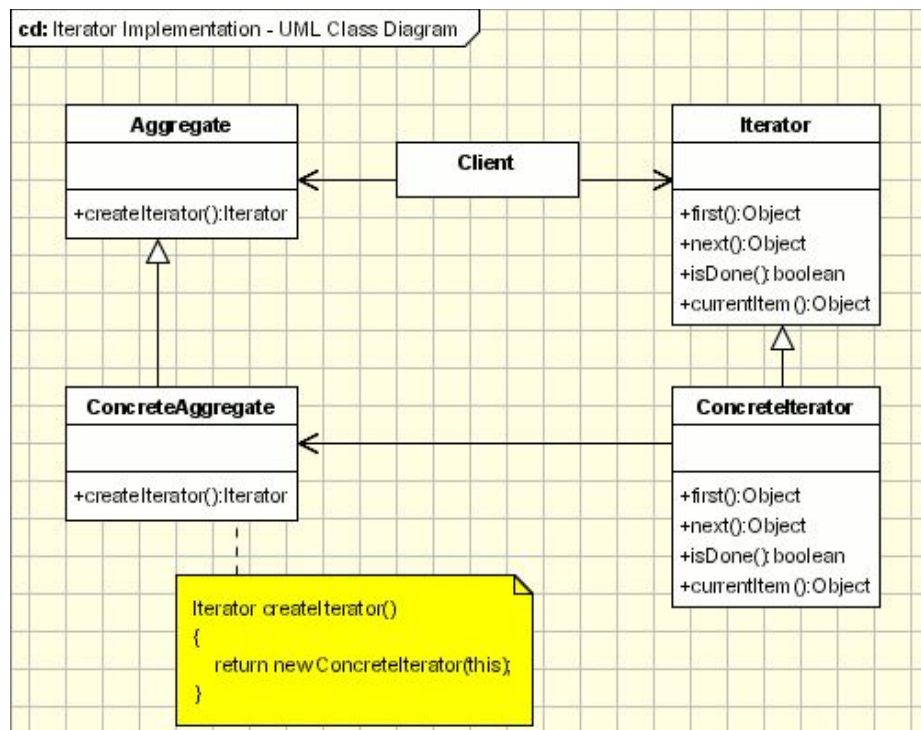
# Design Patterns

## General Idea

- Historically - A design pattern is a solution to a problem that is commonly occurring.
- Pattern definition:
  - Motivation - General problem scenario (when to use pattern)
  - Intent - Description of solution to problem
  - Implementation - UML class design

## Iterator Pattern

- Motivation - Need to traverse items of diverse sorts of collection in a uniform manner, without exposing the internal structure.
- Intent - Create an iterator object that maintains a reference to one item and methods for advancing through the collection.
- Implementation:
  - Client - Some class that contains a collection.
  - Aggregate - The collection.
  - Concrete Aggregate - The implementation of the collection'. It has a "CreateIterator" function, which returns a new ConcreteIterator.
  - Iterator - The class that the client uses to traverse through collection.
  - Concrete Iterator - The implementation of the iterator.



- The client uses the aggregate directly through its public interface. This allows the client to get an iterator. The iterator's interface is then used by the client to traverse the collection.
- Iterating over a book collection example:
  - Aggregate - Container
  - Concrete Aggregate - BooksCollection
  - Iterator - Iterator
  - Concrete Iterator - BookIterator

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

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

public class BooksCollection implements Container {

 private String books[]={"A","B","C","D"};

 public Iterator createIterator() {
 return new BookIterator();
 }

 public class BookIterator implements Iterator {

 private int currentPosition;

 public boolean hasNext() {
 if (currentPosition < books.length) {
 return true;
 } else {
 return false;
 }
 }

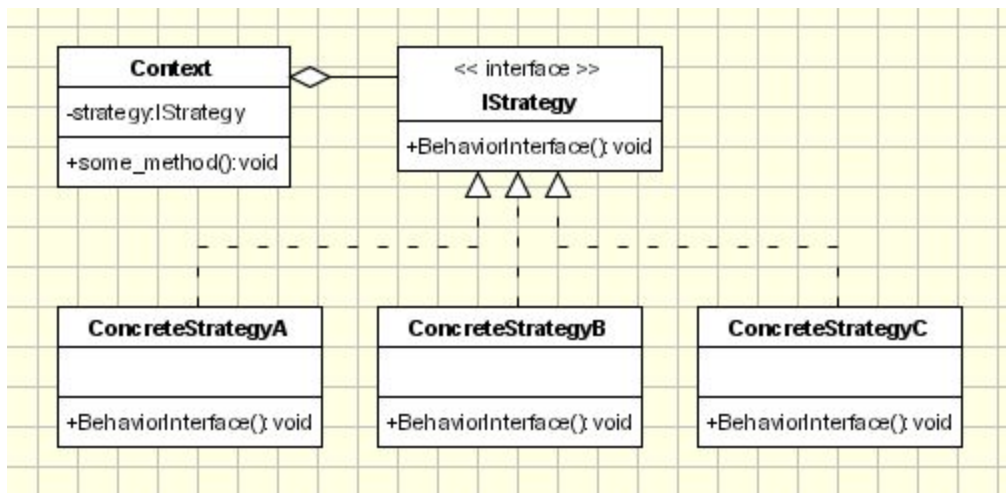
 public Object next() {
 if (this.hasNext()) {
 return books[currentPosition++];
 } else {
 return null;
 }
 }
 }
}
```

## Strategy Pattern

- Motivation - Need a way to adapt the behaviour of an algorithm at runtime by varying one part of the procedure (giving algorithm variants).



- Intent - Define an interface that abstracts the varying functionality, have the client use that interface to supply a concrete variant.
- Implementation:
  - Strategy - An interface common to all supported algorithms
  - Concrete Strategy - Implementations of the algorithm
  - Context - What uses the interface to call the algorithm defined by a concrete strategy.



- The context object receives requests from the client and delegates them to the strategy object. Usually the concrete strategy is created by the client and passed to the context. From this point the clients interacts only with the context.
- Simple in-built java example:

```

// Create List
ArrayList<Integer> list = new ArrayList<Integer>();

// Add items to the list
// ...
// ...

// Sort using given comparison method
Collections.sort(list, new Comparator<Integer>(){
 public int compare(Integer a, Integer b) {
 return a - b;
 }
});

```

- From scratch robot behaviour example:
  - Strategy - Behaviour
  - Concrete Strategy A/B - Aggressive/Defensive Behaviour
  - Context - Robot

```
public interface Behaviour {
 public void makeAction();
}
```

```
public class AgressiveBehaviour implements Behaviour{

 public void makeAction() {
 System.out.println("ATTACKING!");
 }
}
```

```
public class DefensiveBehaviour implements Behaviour {

 public void makeAction() {
 System.out.println("DEFENDING!");
 }
}
```

```
public class Robot {

 Behaviour behaviour;

 public void setBehaviour(Behaviour behaviour) {
 this.behaviour = behaviour;
 }

 public void makeAction() {
 behaviour.makeAction();
 }

 public static void main(String[] args) {
 Robot C3P0 = new Robot();

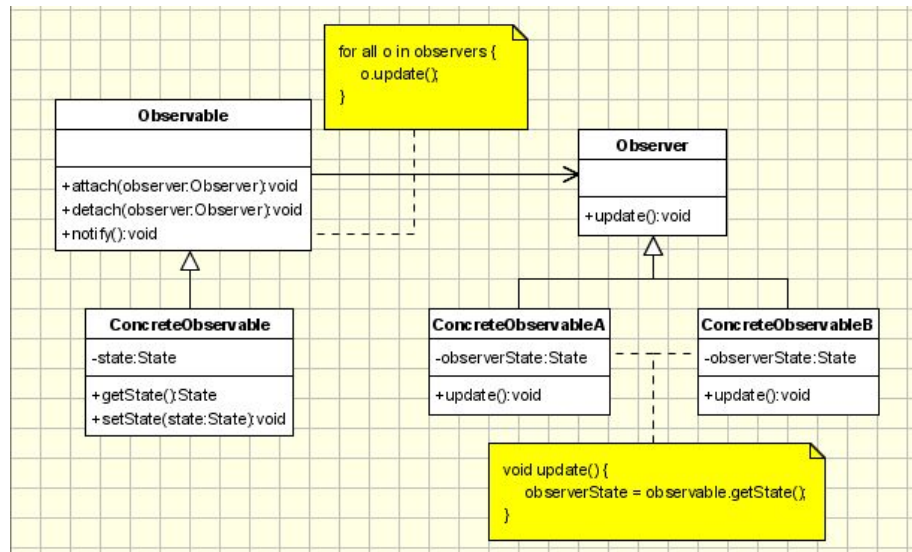
 C3P0.setBehaviour(new DefensiveBehaviour());
 C3P0.makeAction();

 C3P0.setBehaviour(new AgressiveBehaviour());
 C3P0.makeAction();
 }
}
```

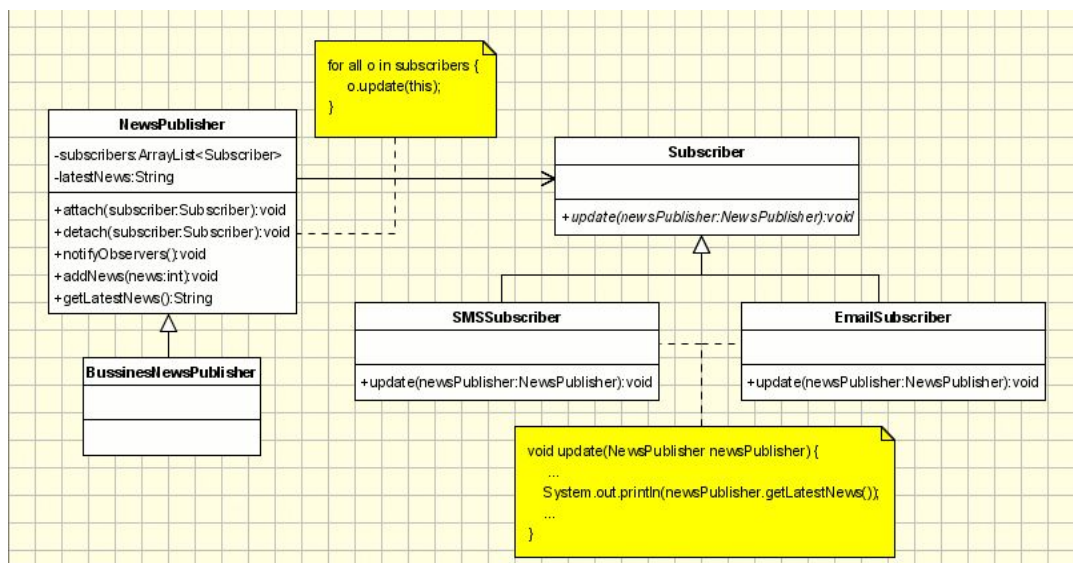
## Observer Pattern

- Motivation - Need a way for a number of objects to be informed of changes in another object.
- Intention - Have the objects maintain a list of objects that need to be informed and notify them whenever any change occurs.
- Implementation:
  - Observable - Interface defining the operations to attach and detach observers.
  - Concrete Observable - Maintains the state of the object and when changes are made, it notifies the attached observers.
  - Observer - Interface defining the operation to notify this object.
  - Concrete Observers - Implementation of Observer



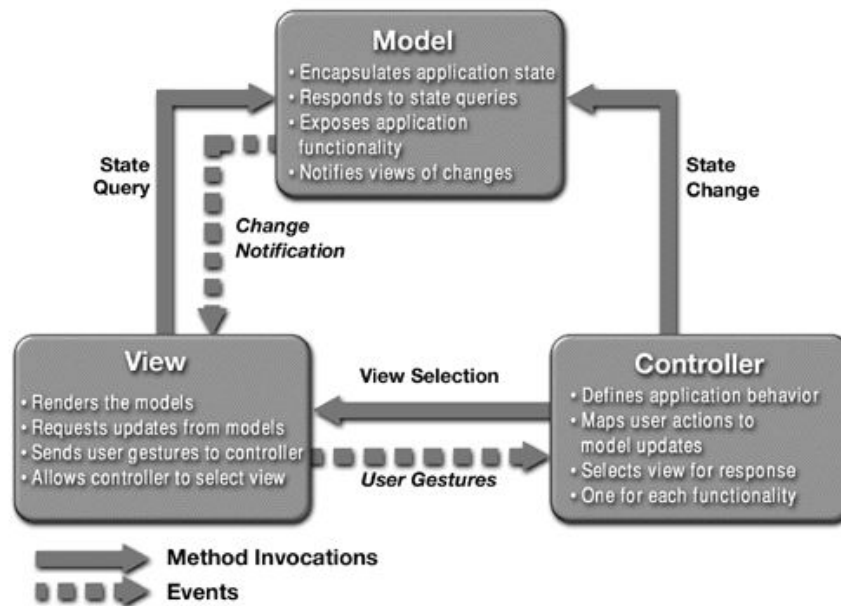


- The main framework instantiates the concrete observable object. Then it instantiates and attaches the concrete observers to it, using the methods defined in the observable interface. Each time the state of the subject changes, it notifies all the attached observers using the methods defined in the observer interface.
- News agency example:
  - Observable - NewsPublisher.
  - Concrete Observable - BusinessNewsPublisher
  - Observer - Subscriber
  - Concrete Observers - SMS/Email Subscriber

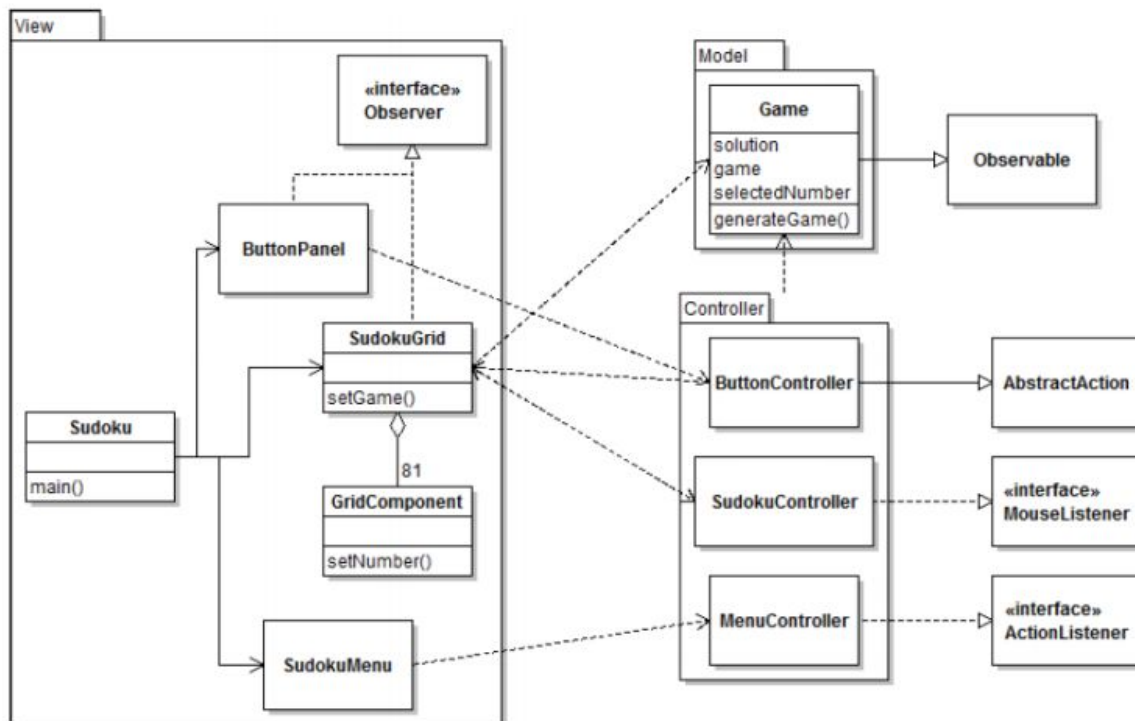


## Model View Controller Pattern

- Three parts:
  - Model - Represents the data and nothing else (nothing else is important).
  - View - How the model data is displayed.
  - Controller - Provides the model data to the view and interprets user actions.
    - Sometimes the controller and the view are the same object.
- This is good for when there are multiple views (e.g. websites).

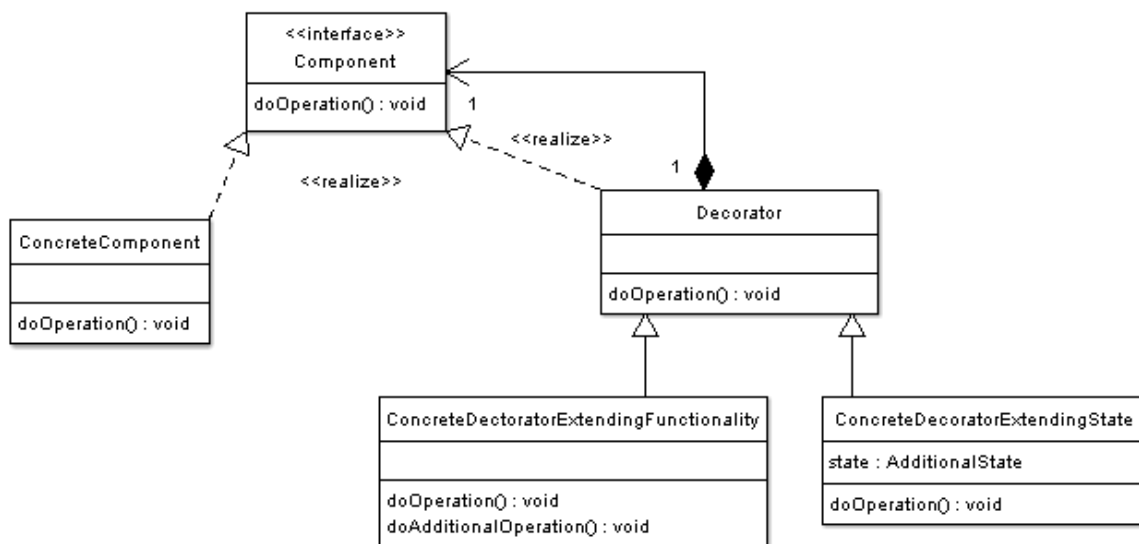


- Sudoku example:



## Decorator Pattern

- Motivation - Need a way to enhance an object's functionality dynamically at runtime, yet use the enhanced object the same way.
- Intent - Have a new class whose object manage other objects, whose methods provide modified functionality over the object.
- Implementation:
  - Component - Interface for objects that can have responsibilities added to them dynamically.
  - Concrete Component - Implementation of the component.
  - Decorator - Maintains a reference to a component object and defines an interface that conforms to the Component's interface.
  - Concrete Decorators - Extend the functionality of the component, by adding state or adding behaviour.



- Car example:
  - Component - Car
  - Concrete Component - BasicCar
  - Decorator - DecoratedCar
  - Concrete Decorators - SportsCar and FamilyCar

```
public interface Car {
 public void assemble();
}
```

```
public class BasicCar implements Car {
 public void assemble() {
 System.out.print("Basic Car.");
 }
}
```

```

public class CarDecorator implements Car {
 Car car;
 public CarDecorator(Car c){
 this.car=c;
 }

 public void assemble() {
 this.car.assemble();
 }
}

```

```

public class FamilyCar extends CarDecorator {
 public FamilyCar(Car c) {
 super(c);
 }

 public void assemble(){
 super.assemble();
 System.out.print(" Adding features of Family Car.");
 }
}

```

```

public class SportsCar extends CarDecorator {

 public SportsCar(Car c) {
 super(c);
 }

 public void assemble(){
 super.assemble();
 System.out.print(" Adding features of Sports Car.");
 }
}

```

```

public class PatternTest {

 public static void main(String[] args) {
 Car sportsFamilyCar = new SportsCar(new FamilyCar(new BasicCar()));
 sportsFamilyCar.assemble();
 }
}

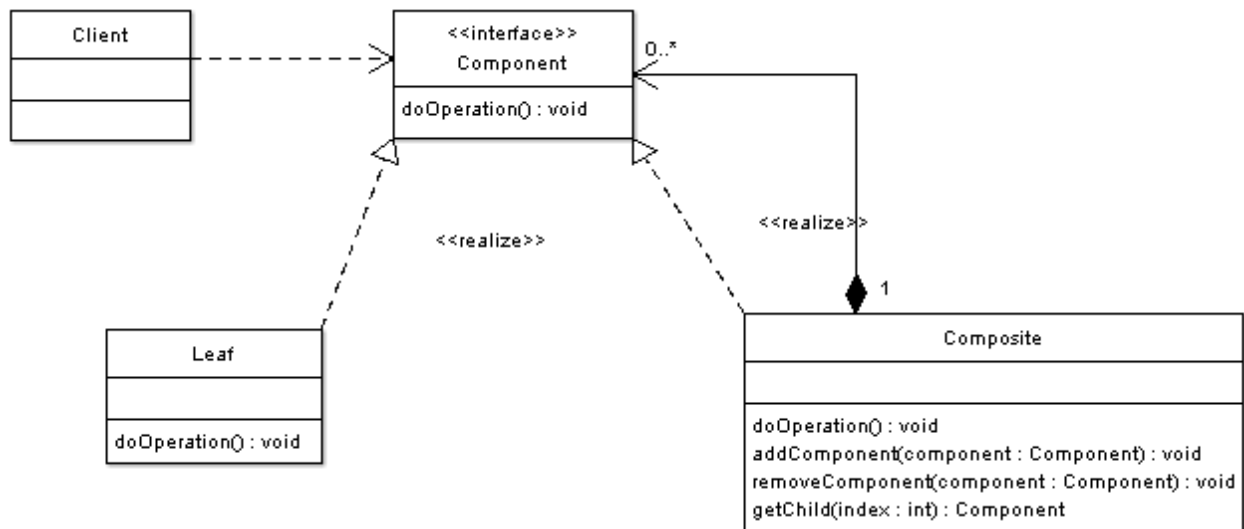
```

Output: Basic Car. Adding features of Family Car. Adding features of Sports Car.

### Composite Pattern

- Motivation - Need a way to treat collections of objects as if there a single (combined) whole.
- Intent - Have a new class whose objects manage a collection of other objects, where to apply a method to the whole requires applying it to each of the objects.

- Implementation:
  - Component - The abstraction for leafs and composites. It defines the interface that must be implemented by the objects in the composition.
  - Leaf - Objects with no children. They implement the services described by the component interface.
  - Composite - Stores child components in addition to implementing methods defined by the component interface. They do this by delegating to child components. Composites also provide methods for adding, removing and getting components.
  - Client - Manipulates objects in the hierarchy and uses the component interface.



- Bundling items example:
  - Component - Item
  - Leaf - Product
  - Composite - Bundle

```

public interface Item {
 public double getPrice();
}

```

```

public class Product implements Item{

 private double price;

 public Product(double price) {
 this.price = price;
 }

 public double getPrice() {
 return price;
 }
}

```

```

public class Bundle implements Item {
 private ArrayList<Item> items;

 public double getPrice() {
 double price = 0;
 for (Item i : items) {
 price += i.getPrice();
 }
 return price;
 }

 public void addItem(Item i) {
 items.add(i);
 }

 public void removeItem(Item i) {
 items.remove(i);
 }

 public Item getChild(int index) {
 return items.get(index);
 }
}

```

# Concurrency

## Producers and Consumers

- Producers - create items
- Consumers - use items
- E.g. Producers add things to a queue and a consumer takes things off the queue.

## Threads

- A process that shares memory with other threads.
- You can create producer and consumer threads that run “simultaneously” to the user.
- Need to be careful of race conditions, where threads are overlapping in accessing/manipulating the same data at the “same time”.
  - “Same time” as it isn’t necessarily simultaneous, but the interleaving of machine instructions are of different threads.
- The “synchronized” keyword means that methods have an implicit lock, i.e. one thread method can’t interrupt another.
  - Still not perfect for eliminating race conditions (all methods must be synchronized)
- Using a combination of synchronised and locks works, but all threads get woken up, which isn’t necessary.
- It’s best to use “reentrant” locks, which allows specific threads to be woken up.



```

public class BoundedQueue<E> {

 private Object[] elements;
 private int head;
 private int tail;
 private int size;

 private Lock queueLock = new ReentrantLock();
 private Condition space = queueLock.newCondition();
 private Condition value = queueLock.newCondition();

 public BoundedQueue(int capacity) {
 elements = new Object[capacity];
 head = 0;
 tail = 0;
 size = 0;
 }

 public void add (E newValue) throws InterruptedException {
 queueLock.lock();
 try {
 while (isFull()) space.await();
 elements[tail] = newValue;
 tail++;
 size++;
 if (tail == elements.length) {
 tail = 0;
 }
 value.signalAll();
 }
 finally {queueLock.unlock();}
 }

 public E remove() throws InterruptedException {
 queueLock.lock();
 try {
 while (isEmpty()) value.await();
 E toReturn = (E) elements[head];
 head++;
 size--;
 if (head == elements.length) {
 head = 0;
 }
 space.signalAll();
 return toReturn;
 }
 finally {queueLock.unlock();}
 }

 public boolean isFull() {
 return size == elements.length;
 }

 public boolean isEmpty() {
 return size == 0;
 }
}

```

```

public class Producer implements Runnable {

 private BoundedQueue<String> queue;
 private int limit;

 public Producer(BoundedQueue<String> queue, int limit) {
 this.queue = queue;
 this.limit = limit;
 }

 public void run() {
 try {
 int count = 0;
 while (count < limit) {
 queue.add("Value");
 count++;
 }
 } catch (InterruptedException exception) {};
 }
}

```

```

public class Consumer implements Runnable {

 private BoundedQueue<String> queue;
 private int limit;

 public Consumer(BoundedQueue<String> queue, int limit) {
 this.queue = queue;
 this.limit = limit;
 }

 public void run() {
 try {
 int count = 0;
 while (count < limit) {
 queue.remove();
 count++;
 }
 } catch (InterruptedException exception) {};
 }
}

```

```

public class Tester {

 public static void main(String[] args)
 {
 BoundedQueue<String> queue = new BoundedQueue<String>(10);
 Runnable run1 = new Producer(queue, 50);
 Runnable run2 = new Producer(queue, 50);
 Runnable run3 = new Consumer(queue, 100);

 Thread thread1 = new Thread(run1);
 Thread thread2 = new Thread(run2);
 Thread thread3 = new Thread(run3);

 thread1.start();
 thread2.start();
 thread3.start();
 }
}

```