**FIT2099: Exam Excerpts**

**Index**

Contents

[Keeping implementation details of classes private 1](#_Toc11404402)

[Risks of repeated code 1](#_Toc11404403)

[Fixing repeated code 2](#_Toc11404404)

[JavaDocs for private attributes 2](#_Toc11404405)

[Declared Exceptions vs Undeclared Exceptions 2](#_Toc11404406)

[Writing Executable Preconditions with Assertions and Exceptions 3](#_Toc11404407)

[Why not to cope with violated preconditions 3](#_Toc11404408)

[Access control for methods in preconditions 3](#_Toc11404409)

[Benefits of Writing Preconditions before Implementation 3](#_Toc11404410)

[Benefits of Writing Postconditions Before Implementation 3](#_Toc11404411)

[Command-Query Separation, Relation to Design by Contract 3](#_Toc11404412)

[Liskov Substitution Principle and Pre/Post Conditions 3](#_Toc11404413)

[Interface in Java 4](#_Toc11404414)

[Abstract Class in Java 4](#_Toc11404415)

[Interfaces vs Abstract Classes 4](#_Toc11404416)

[Why Reduce Dependencies 4](#_Toc11404417)

[Encapsulation and Access Modifiers to control dependencies 4](#_Toc11404418)

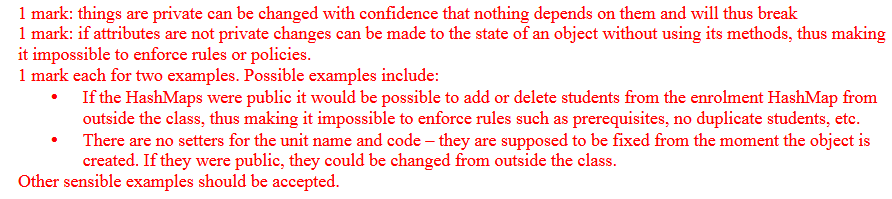
[What is good abstraction? Abstraction vs Abstract Classes 4](#_Toc11404419)

[More Specific Stuff 4](#_Toc11404420)

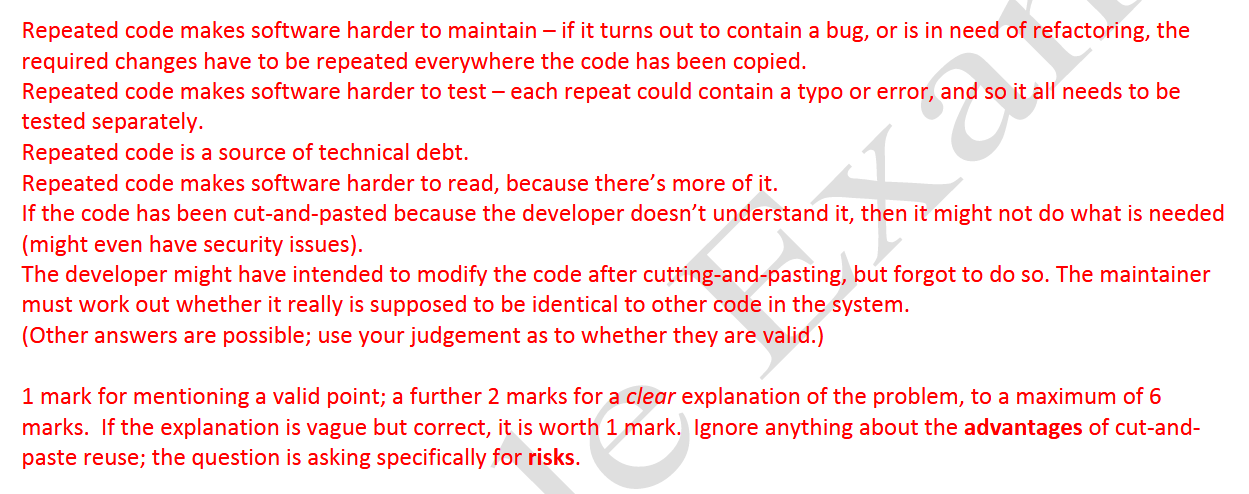
[Class Diagram and Sequence Diagram Mark Schemes 4](#_Toc11404421)

[Watch Code 8](#_Toc11404422)

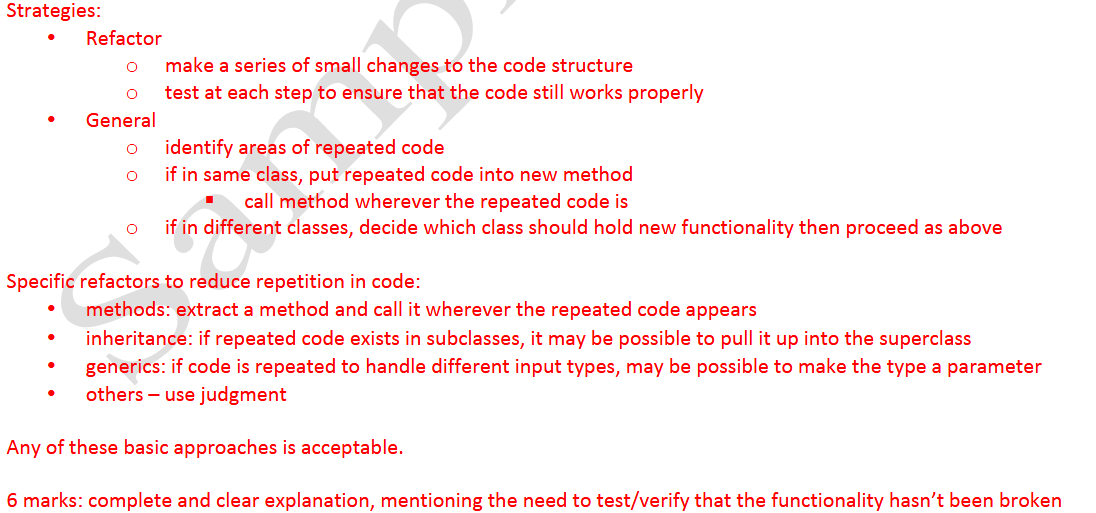
# Keeping implementation details of classes private

****

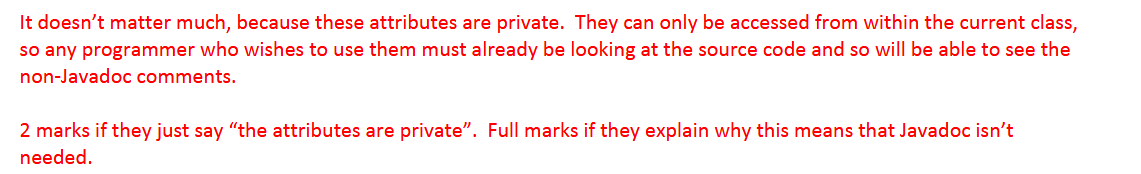
# Risks of repeated code



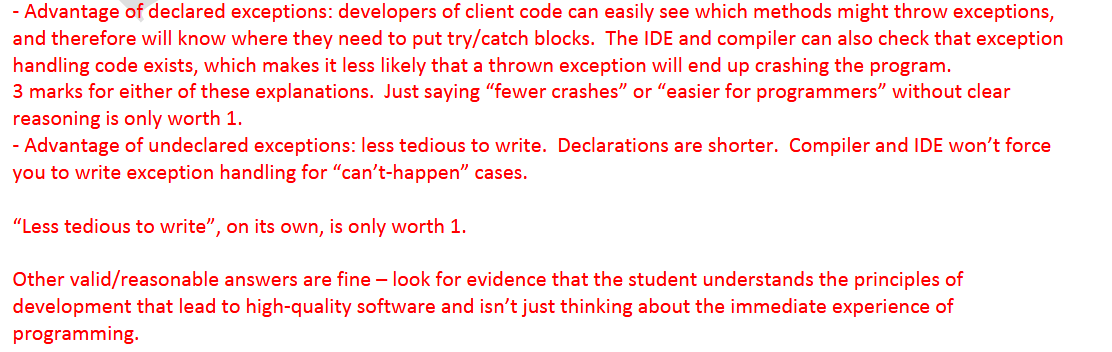
# Fixing repeated code



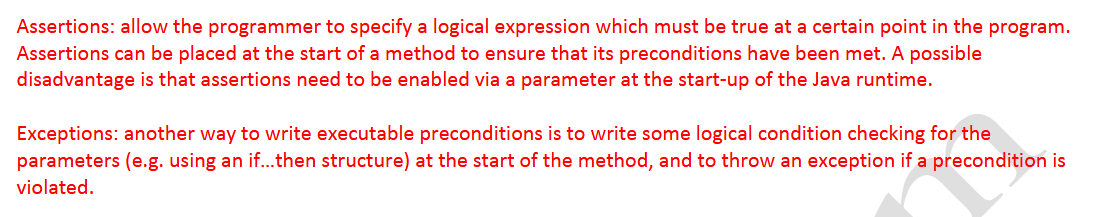
# JavaDocs for private attributes

****

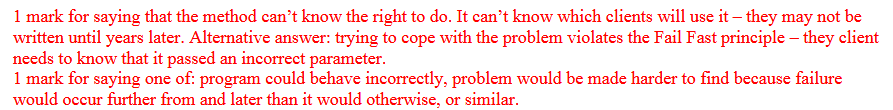
# Declared Exceptions vs Undeclared Exceptions

****

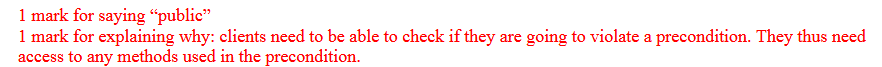
# Writing Executable Preconditions with Assertions and Exceptions

****

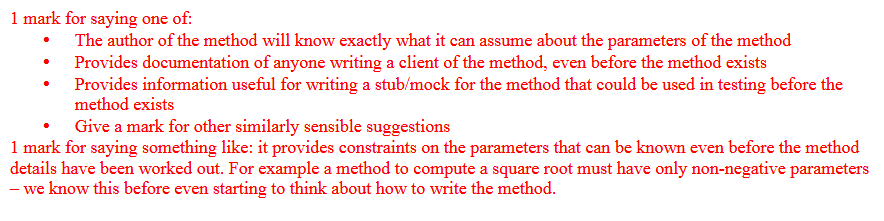
# Why not to cope with violated preconditions



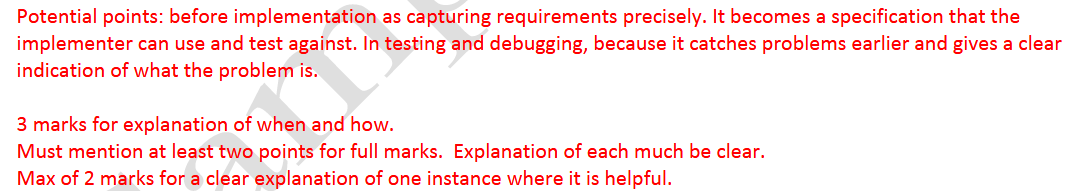
# Access control for methods in preconditions

****

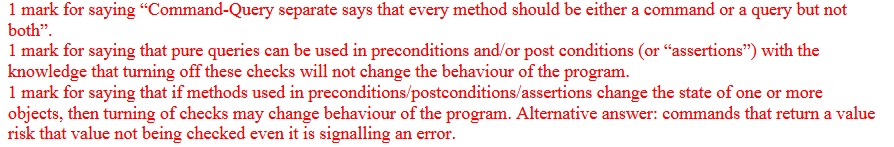
# Benefits of Writing Preconditions before Implementation

****

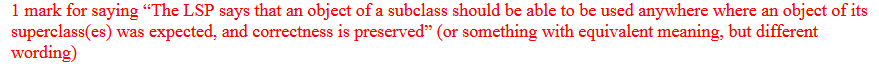
# Benefits of Writing Postconditions Before Implementation

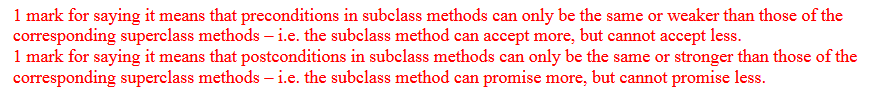
****

# Command-Query Separation, Relation to Design by Contract

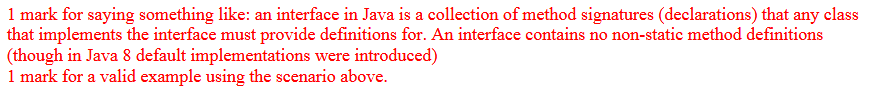


# Liskov Substitution Principle and Pre/Post Conditions

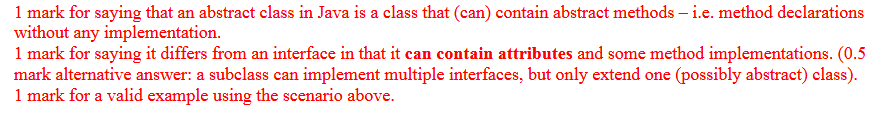
****

****

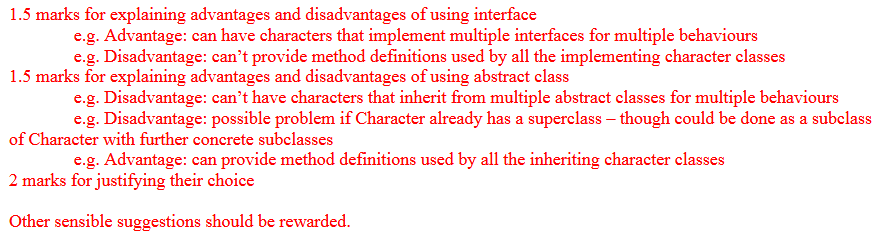
# Interface in Java

****

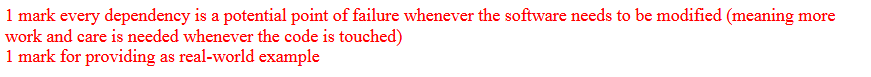
# Abstract Class in Java

****

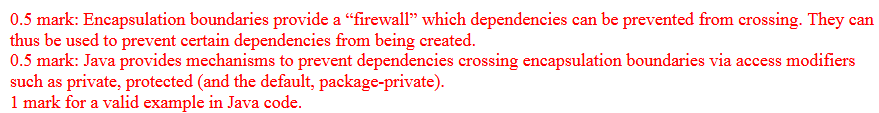
# Interfaces vs Abstract Classes

****

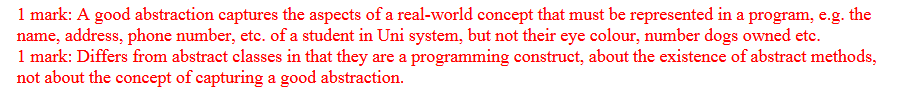
# Why Reduce Dependencies

****

# Encapsulation and Access Modifiers to control dependencies

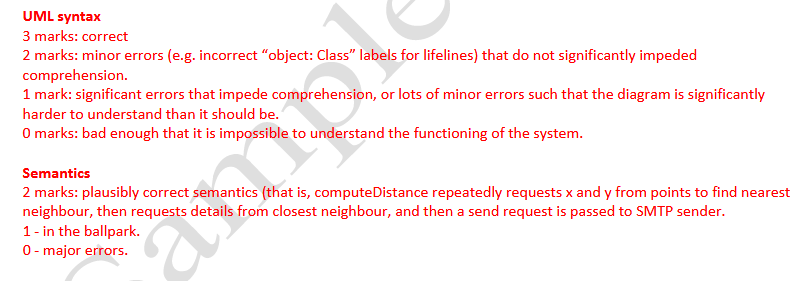
****

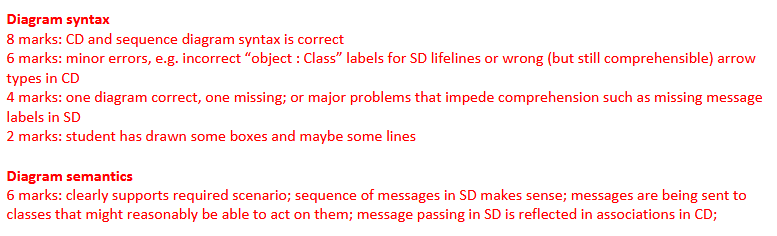
# What is good abstraction? Abstraction vs Abstract Classes

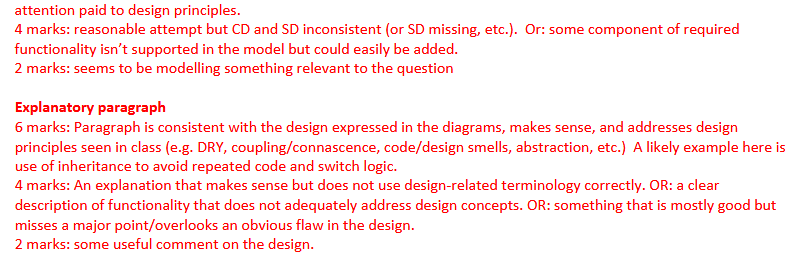
****

# More Specific Stuff

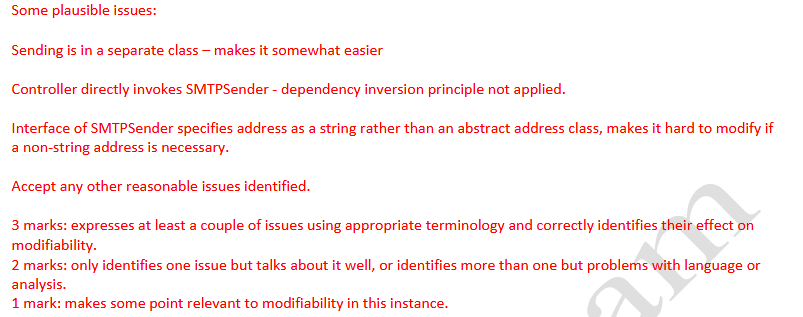
# Class Diagram and Sequence Diagram Mark Schemes



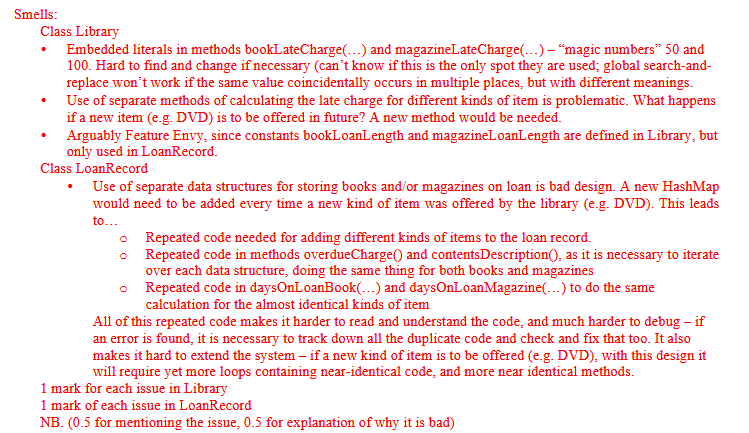


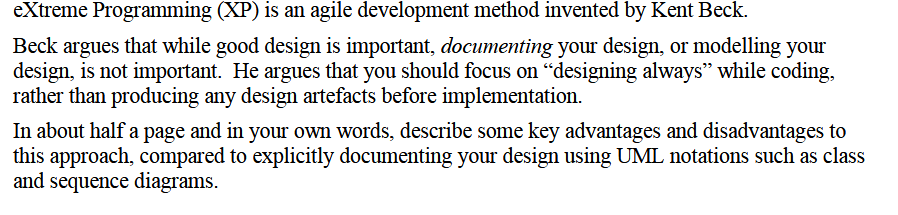


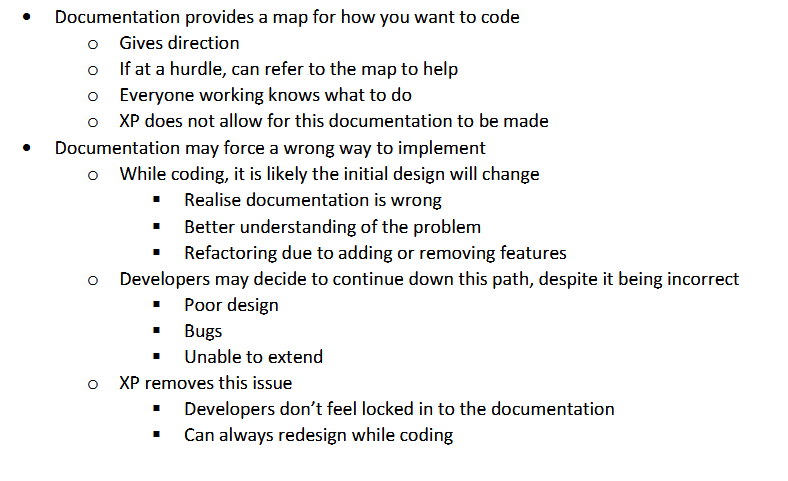
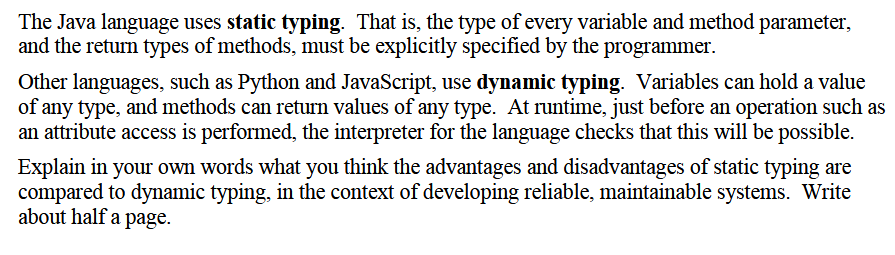
**Ease of Extending a System**

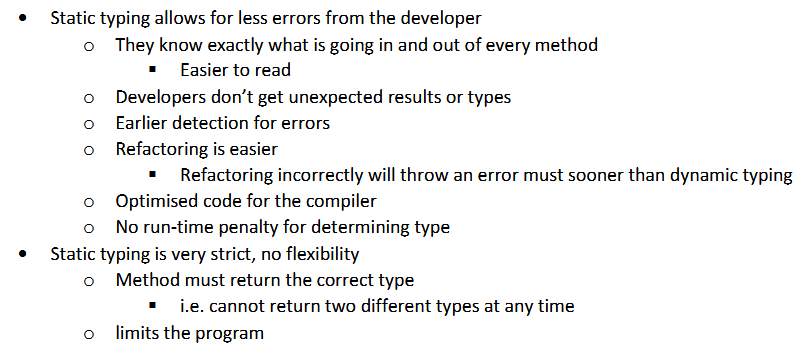
****

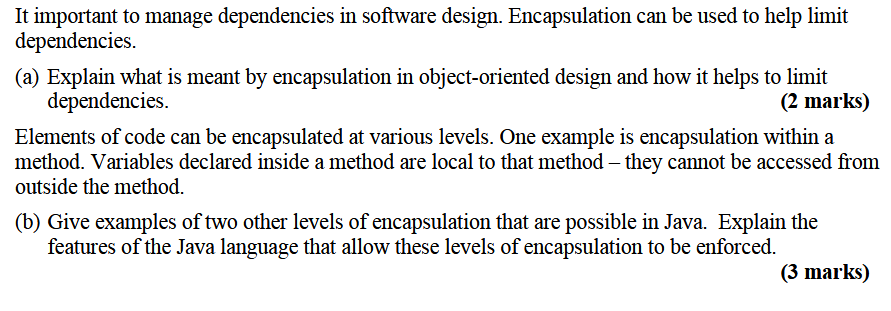
**Code Smells**

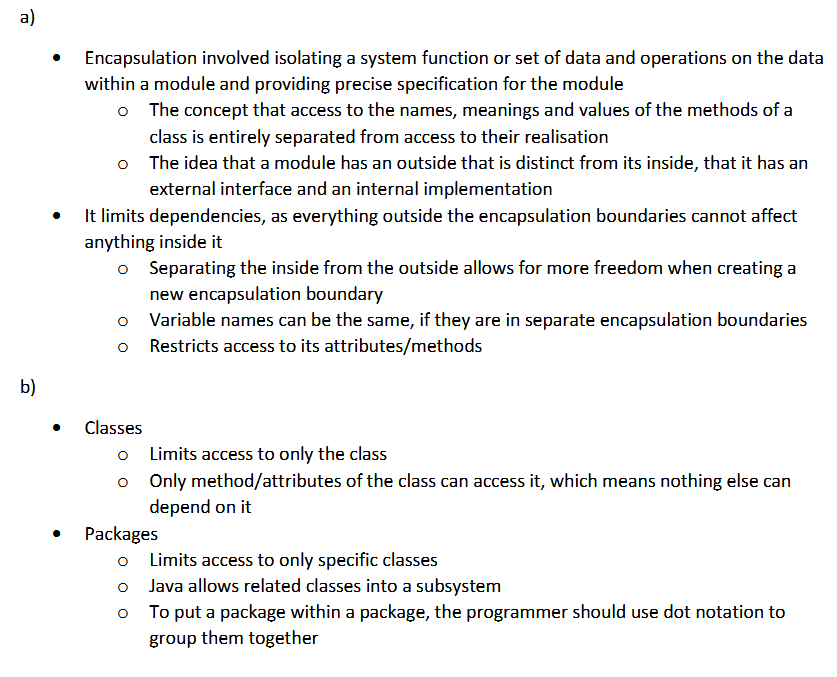
****

****

****

****

****

****

# Watch Code

**Driver.java**

import java.util.ArrayList;  
  
import edu.monash.fit2099.watches.\*;  
  
public class Driver {  
   
 public static void main(String[] args) {   
   
 ArrayList<Watch> watches = new ArrayList<Watch>();  
   
 watches.add(new Watch1());  
 watches.add(new Watch2());  
 try {  
 watches.add(new Watch(new int[] {24, 60, -60, 1000} ));  
 }  
 catch (Exception e) {  
 System.out.println("Watch construction failed with message: \n\t"  
 + e.getMessage()  
 + "\nLet's not bother with this watch for now."  
 );  
 }  
   
 System.out.println("##############################");  
 for (Watch watch : watches) {  
 System.out.println("Testing Watch: " + watch.getClass().getSimpleName());  
 watch.testWatch(200);   
 System.out.println("##############################");  
 }  
 }  
  
}

**Counter.java**

package edu.monash.fit2099.counters;  
  
public class Counter {  
  
 private int value = 0;  
   
 public void reset() {  
 value = 0;  
 }  
   
 public void decrement() {  
 value--;  
 }  
   
 public void increment() {  
 value++;  
 }  
   
 public int getValue() {  
 return value;  
 }  
  
 @Override  
 public String toString() {  
 return Integer.valueOf(this.getValue()).toString();  
 }  
  
}

**LinkedCounter.java**

package edu.monash.fit2099.counters;  
  
public class LinkedCounter extends MaxCounter {  
   
 Counter neighbour;  
   
 public LinkedCounter(int max, Counter neighbour) {  
 super(max);  
 this.neighbour = neighbour;  
 }  
   
 @Override  
 public void increment() {  
 super.increment();  
 if (this.getValue() == 0) {  
 neighbour.increment();  
 }  
 }  
}

**MaxCounter.java**

package edu.monash.fit2099.counters;  
  
public class MaxCounter extends Counter {  
  
 private final int max;  
 private final String fieldFormat;  
  
 public MaxCounter(int max) {  
  
 if (max <= 0) {  
 throw new IllegalArgumentException("Maximum value of a MaxCounter must be greater than zero.");  
 }  
   
 this.max = max;  
 // create a format string with the correct field width for this counter  
 double fieldWidth = Math.ceil(Math.log10(max));  
 fieldFormat = "%0" + String.format("%.0f", fieldWidth) + "d";  
 }  
   
 public int getMax() {  
 return max;  
 }  
   
 @Override   
 public void increment() {  
 super.increment();  
 if (this.getValue() == max) {  
 this.reset();  
 }  
 }  
   
 @Override  
 public String toString() {  
 return String.format(fieldFormat, this.getValue());  
 }  
}

**Watch.java**

package edu.monash.fit2099.watches;  
  
import java.util.ArrayList;  
  
import edu.monash.fit2099.counters.\*;  
  
/\*\*  
 \* Implements a watch made up of an arbitrary number of counters linked together.  
 \* Each counter has a maximum value. When it reaches its maximum, it resets its  
 \* value to zero, and increments its neighbour.  
 \*   
 \* @author David Squire  
 \*  
 \*/  
public class Watch {  
   
 static public final int MAX\_HOURS = 24;  
 static public final int MAX\_MINUTES = 60;  
 static public final int MAX\_SECONDS = 60;  
 static public final int MAX\_MILLISECONDS = 1000;  
   
 private ArrayList<MaxCounter> counters = new ArrayList<MaxCounter>();  
   
 public Watch() {  
 /\* needed so no-argument constructors in subclasses can call it.  
 \* A Watch created using this will do nothing, as it has no  
 \* counters.  
 \*/  
 }  
   
 /\*\*  
 \* Create a Watch using an array integers that specify the maximum values of the counters  
 \* that make up the desired Watch. Elements of the array must be in order from the most significant  
 \* counter (e.g. hours) at position {@code 0}, to the least significant (e.g. seconds) at position  
 \* {@code maxValues.length - 1}  
 \*   
 \* @param maxValues an array of integers that specify the maximum values of the counters.  
 \* There must be at least one element in the array, and all the maximum values must be  
 \* greater than 0.  
 \*   
 \*/  
 public Watch(int[] maxValues) {  
   
 assert maxValues != null : "Null reference passed to Watch constructor.";  
 assert maxValues.length >= 1 : "Must pass at least one counter maximum value in array parameter";  
   
 MaxCounter lastCounter = new MaxCounter(maxValues[0]);  
 this.addCounter(lastCounter);  
 for (int i = 1; i < maxValues.length; i++) { // notice we start from 1, not 0  
 // Commented out to demonstrate the Exception thrown by the MaxCounter constructor  
 // assert maxValues[i] > 0 : "Counter maximum values must be greater than zero. maxValues[" + i + "] = " + maxValues[i];  
 MaxCounter thisCounter = new LinkedCounter(maxValues[i], lastCounter);  
 this.addCounter(thisCounter);  
 lastCounter = thisCounter; // notice we can assign a LinkedCounter to a MaxCounter  
 }  
 }  
   
 protected void addCounter(MaxCounter newCounter) {  
 counters.add(newCounter);  
 }  
   
 protected MaxCounter getLeastSignificantCounter() {  
 return counters.get(counters.size() - 1);  
 }  
   
 public void display() {   
 String prefix = "";   
 for (MaxCounter thisCounter : counters) {  
 System.out.print(prefix + thisCounter);  
 prefix = ":";  
 }  
 System.out.println();  
 }  
   
 /\*\*  
 \* Increment the least significant counter of the Watch.  
 \*/  
 public void tick() {  
 getLeastSignificantCounter().increment();  
 }  
   
 public void testWatch(int numTicks) {  
 for (int i = 0; i < numTicks; i++) {  
 display();  
 tick();  
 }  
 }  
  
}

**Watch1.java**

package edu.monash.fit2099.watches;  
  
public class Watch1 extends Watch {  
   
 private Watch2 myWatch2;  
   
 public Watch1() {   
 myWatch2 = new Watch2();  
 }  
   
 public void tick() {  
 myWatch2.tick(); // delegation  
 }  
   
 @Override  
 public void display() {  
 myWatch2.display(); // delegation  
 }  
   
}

**Watch2.java**

package edu.monash.fit2099.watches;  
  
public class Watch2 extends Watch {  
   
 public Watch2() {  
 super(new int[] {MAX\_HOURS, MAX\_MINUTES});  
 }  
}

**Watch3.java**

package edu.monash.fit2099.watches;  
  
public class Watch3 extends Watch {  
   
 public Watch3() {  
 super(new int[] {MAX\_HOURS, MAX\_MINUTES, MAX\_SECONDS, MAX\_MILLISECONDS});  
 }  
   
}