# CS2500 Summer 2012 Solutions

## Question 1.a

```
public static void main(String[] args) {
        for (String argument : args) {
                System.out.println(argument);
         }
}
```

## Question 1.b

```
public static void sumInts() {
        Scanner scanner = new Scanner(System.in);

        int sum = 0;

        while (scanner.hasNextInt()) {
                sum += scanner.nextInt();
        }
        System.out.println(sum);
}
```

## Question 2.a

An object is an instance of a given class. The class is a blueprint for the object. It describes the objects instance variables and instance methods and how the object should be constructed.

## Question 2.b

```
public class Switch {
        private boolean currentState;
        private static int switchCount = 0;

        public Switch() {
                this.currentState = true;
                this.switchCount++;
         }

        public boolean getState() {
                return currentState;
          }

         public void toggleState() {
                currentState = !currentState;
          }
```

```
        public static int getSwitchCount() {
                return switchCount;
         }
}
```

## ***Question 3.a***

Inheritance is an object oriented feature supported by Java wherein the features of one Java class can be inherited/made available in another class. This creates a parent child relationship between these 2 classes. Class Inheritance in java mechanism is used to build new classes from existing classes. The inheritance relationship is transitive: if class x extends class y, then a class z, which extends class x, will also inherit from class 'y'. Object-oriented programming allows classes to inherit commonly used state and behaviour from other classes.

**Example:**

```
public class Parent  {
        private String name = "Rocky";
        public String getName() {
                return this.name;
        }
}

public class Child extends Parent {
        public static void main(String[] args) {
                System.out.println("Name in Parent is: " + getName());
        }
}
```

Here the getName() method is available only in the parent class but is directly used in the child class because the method is public and is directly accessible to the child class since it has extended the parent class.

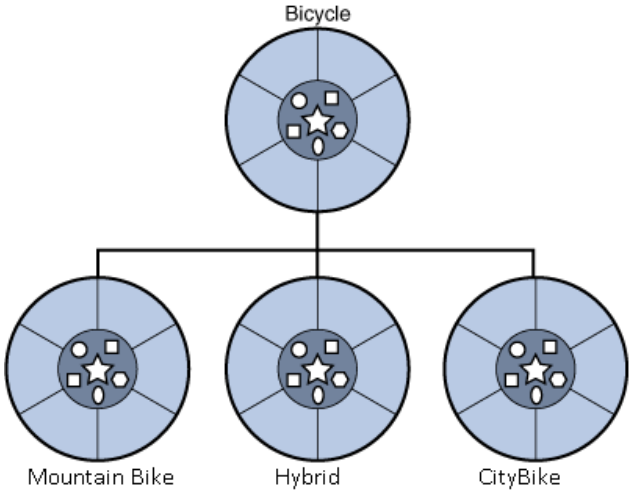
In Java subclasses may extend a unique superclass. When a subclass extends its superclass, the subclass inherits all public attributes and methods from the superclass. This means the subclass can access any inherited attribute and has the (default) method behaviour when calling an inherited method.

Advantages:

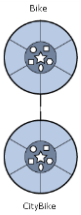
- Reusability of code. Don't have to rewrite the "Wheel"
- Extensibility -- extending the base class is easy. Example "super bike" extends bike.
- Inherit state and behaviour from a more general class with common state / behaviour / functionality
- Separates class specific from more general code
- Less error prone code base

***Question 3.b***

In Java terminology, a **more general class** in an inheritance hierarchy is called a ***superclass.*** A **more specific** class is a ***subclass***. `Bike` is a superclass of both `CityBike` and `MountainBike`. Going in the opposite direction, both `CityBike` and `MountainBike` are subclasses of `Bike`. When two classes are right next to each other in the inheritance hierarchy, their relationship is said to be ***direct***. The act of declaring a direct subclass is referred to in Java circles as ***class extension***.




**//IS-A**




The picture shows a parent class and a child class. The line between them shows the "is-a" relationship. The arrow points to the parent class from the child class. The picture can be read as "a CityBike is-a Bike." The phrase "is-a" is in common use in computer science. The arrow between a child and parent is sometimes called an "is-a link".

**//EXTENSION**

When you derive a class from a base class we say the derived subclass extends the base class i.e. CityBike extends the Bike. We inherit all public methods from the super-class “Bike”. Some of these methods may be abstract and so we need to define the behaviour in the sub-class.

**//SUB-CLASS**

A sub-class is a class that inherits some properties from its super-class. You can usually think of the subclass as being "a kind of" its super-class, as in a "a CityBike is a kind of Bike” or "a square is a kind of rectangle":

- A Bike has wheels, handle bar, and a saddle
- A rectangle has four sides with lengths w and h
- A square has all of the characteristics of a rectangle; in addition, w = h
- A sub-class is a more specific version of its super-class.

**//SUPER-CLASS**

A Java super-class is a class which gives a method or methods to a Java subclass. A Java class may be a sub-class, a super-class, both, or neither. Some super-classes can be abstract and so must have a

sub-class to implement its behaviour. I.e. you can't create a bike because its abstract, but you can create a CityBike to implement its methods.

## ***Question 3.c***

Overriding means you implement a method in the subclass that has the same signature as one in the parent class.

e.g.

```
// Parent class
public class PClass{
        public void func(){
        }
}

// Child class
public class CClass{
        @override
        public void func(){
                // overriden method that has the same signature
        }
}
```

Overloading means you only change the parameters

```
// Parent class
public class PClass{
        public void func(int a){
        }
}

// Child class
public class CClass{
        public void func(int a, int b){
                /* overriden method that has the same signature but different arguments */
        }
}
```

We can overload constructors but not override them. Overloading refers to methods with the same name but uses a different set of parameters. Whereas overriding refers to replacing the behaviour of an overridable method defined in its parent class.

**MORE INFO ON OVERLOADING / OVERRIDING**

**//OVERLOAD**

If two methods from the same class have the same name, the same visibility, but different types for their parameter list then the methods are said to overload each other.

The following two methods overload each other.

```
public void f( int a ) { }
public void f( int a, int b ) { }
```

OVERLOADING IS AN EXAMPLE OF COMPILE TIME POLYMORPHISM.

**//OVERRIDE**

If a class inherits a method from its super class, then there is a chance to override the method provided that it is not marked final.

The benefit of overriding is the ability to define a behaviour that's specific to the sub class type. Which means a subclass can implement a parent class method based on its requirement. In object oriented terms, overriding means to override the functionality of any existing method.

```
class Animal{
        public void move(){
                System.out.println("Animals can move");
        }
}

class Dog extends Animal{
        public void move(){
                System.out.println("Dogs can walk and run");
        }
}
```

OVERRIDING IS AN EXAMPLE OF RUN TIME POLYMORPHISM.

***Question 4***

```
// book.java
public abstract class Book {
        protected final String title;
        protected final int pageCount;
        protected double price;

        public Book (String title, int pageCount) {
                this.title = title;
                this.pageCount = pageCount;
        }

        public abstract void setPrice();
}

// ebook.java
public class EBook extends Book {
        @override
        Public void setPrice() {
                this.price = 0.10 * this.pageCount;
        }
}

// paperbook.java
public abstract PaperBook extends Book {
        public PaperBook (String title, int pageCount) {
                super (title, pageCount);
        }
}
```

```
// paperback.java
public class Paperback extends PaperBook {
        public Paperback (Sting title, int pageCount) {
                super (title, pageCount);
                setPrice();
        }

        @override
        public void setPrice() {
                this.price = (0.15 * this.pageCount) + 1;
        }
}

// hardback.java (not required for exam)
public class Hardback extends PaperBook {
        public Hardback (Sting title, int pageCount) {
                super (title, pageCount);
                setPrice();
        }

        @override
        public void setPrice() {
                this.price = (0.15 * this.pageCount) + 5;
        }
}
```

## Question 5

```
public class ExamButton extends JButton implements ActionListener {
        int clicks;

        private ExamButton( ) {
                super( "click me" );                    // Argument CLICK ME to original constructor
                addActionListener( this );              //Something happens
                clicks = 0;
        }

        // Override ActionListener
        @Override
        public void actionPerformed( ActionEvent event ) {
                setText( "Number of clicks = " + (++ clicks) );
        }

        // Main
        public static void main (String[] args ) {
                JFrame frame = new JFrame( );                                   // New JFrame
                ExamButton button = new ExamButton( );                          // New Button
                frame .getContentPane( ) . add ( button );
                frame . setDefaultCloseOperation( JFrame.EXIT ON_CLOSE );
                frame . setSize( 200, 300 );
                frame .setVisible( true );
        }
}
```

## ***Question 6.a***

Simply put, **recursion is when a function calls itself**. That is, in the course of the function definition there is a call to that very same function. At first this may seem like a never ending loop, or like a dog chasing its tail. It can never catch it. So too it seems our method will never finish. This might be true is some cases, but in practise **we can check to see if a certain condition is true and in that case exit (return from) our method**. The case in which we end our recursion is called a **base case.**
Additionally, just as in a loop, we must change some value and incrementally advance closer to our base case.

Every recursion should have the following characteristics.

- A simple base case which we have a solution for and a return value.
- A way of getting our problem closer to the base case. I.e. a way to chop out part of the problem to get a somewhat simpler problem.
- A recursive call which passes the simpler problem back into the method.

## ***Question 6.b***

We know: $(x+1)^2 = x^2 + 2x + 1$ *//From question*
Therefore: $((x-1) + 1)^2 = (x-1)^2 + 2(x-1) + 1$ *//Substitute x with x-1*
$X^2 = (x-1)^2 + 2x - 1$

```
public int square(int x) {
        if (x == 1) {                                   //Base Case = 1
                return;
        } else {
                return square(x-1) + x + x - 1;         //This is equivalent to (x-1)^2 +2x - 1
}
```

//Proof
Example – test a number to see if it returns the right value, we know that $5^2 = 25$, so our return at the end should be 25.

x = 5

```
sq(4) + 5 + 5 -1                                        // Return 25       (16 + 5 + 5 – 1 = 25)
        |> sq(3) + 4 + 4 -1                             // Return 16       (9 + 4 + 4 -1 = 16)
                |> sq(2) + 3 + 3 -1                     // Return 9        (4 + 3 + 3 – 1 = 9)
                        |> sq(1) + 2 + 2 -1             // Return 4        (1 + 2 + 2 -1 = 4)
                                |> 1                    //Base Case reached so just return 1
```

## ***Question 7.a***

```
public class PartialIterableClass implements Iterable<String> {
        private String[] things;

        public PartialIterableClass( String[] things ) {
```

```
                this.things = things;
        }

        public Iterable<String> iterator () {
                return new Iterator<String> {

                        int current = 0;

                        // If the array has another String return true otherwise return false
                        @Override
                        private boolean hasNext() {
                                if (current == things.lenght -1) {
                                        return false;
                                } else {
                                        return true;
                                }
                        }

                        // Get the next String in the array things because we know it has one from the hasNext()
                        @Override
                        private String next() {
                                return things[current ++];
                        }

                        // create a new array and store all the Stings in the array 'things' except for the current one
                        @Override
                        public void remove() {
                                // New temp array, -1 because it will have 1 less than 'things'
                                String[] newThings = new String [things.lenght – 1];
                                // Left hand side of current String
                                for ( int I = 0, I < current; i++ ) {
                                        newThings [i] = things[i];
                                        // Right hand side of current String
                                        for ( int j = current + 1; j < things.lenght; j++ ) {
                                                newThings[j] = things[j];
                                        }
                                }
                                newThings = things;             // replace 'things' with the items in 'newThings'
                        }
                }
        }
}
```

## ***Question 7.b***

```
public class GenericStack<T> {

        private ArrayList<T>;

        public GenericStack() {
                stack = new ArrayList<T> ();
        }

         public void push(<T> arg) {
```

```
                stack.add(arg);
        }

        public T pop() {
                return stack.remove(stack.size() -1);
        }

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

## Question 8

```
public static void handleException (Exception e) {
        //message + print
        String error = e.getMessage();
        If (!(error.equals(null)) {
                System.err.println(error);
                //Stack Trace
                E,printStacjTrace();
        }
}

public static void main(String[] args) {
        JoeThermometer therm;

        try {
                therm = new JoeThermometer;
                System.out.println(therm.getTemperature());
                therm.close;
        } catch (Exception e) {
                handleException(e);
        } finally {
                therm.close;
        }
}
```

## Question 9.a

Java enums and int enums both provide a way to define a collection of "constant" symbols. Java enums overcome all the disadvantages of int enums. Java enums are easy to use. For example, translating to String is automatic and iterating over all enum constants is easy. With int enums this is not the case: converting to String typically requires a switch statement and iterating is difficult if the constants are not contiguous. Java enums help you maintain your code. For example, rearranging enum constants, removing, or adding constants do not break code. With int enums rearranging may break code if typos result in duplicate constants. Likewise, removing and adding constants breaks the code for iterating.

## Question 9.b

```
/*
Obviously I've made up the numbers in this solution. You can't use ints to represent the mass or diameter but the important part is how to use enums.
*/


public enum Planet {
	EARTH(5.975e24, 6.378e6),
	VENUS(4.869e24, 6.052e6),
	MARS (6.419e23, 3.393e6);

	private double mass;
	private double diameter;

	Planet(double mass, double diameter) {
		this.mass = mass;
		this.diameter = diameter;
	}

	public double getMass() {
		return mass;
	}
	public double getDiameter() {
		return diameter;
	}
}
```