Option D: Object-Oriented Programming Revision Guide

Contents

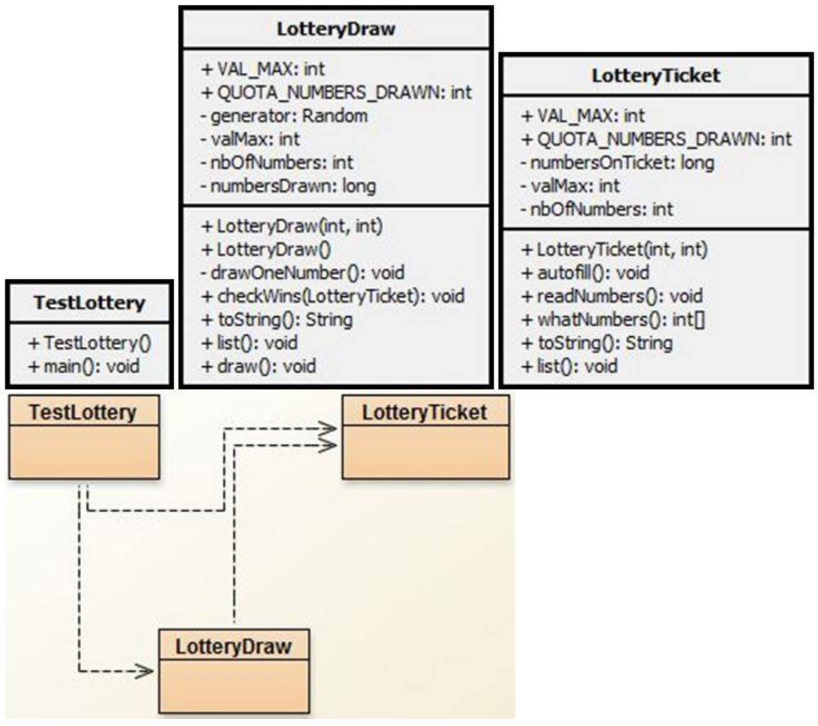
D1- Objects as a Programming Concept

D.1.1 & D1.2 Classes and Objects

An object is a representation of a real world entity e.g. book, car, student etc. When designing a new system you would identify all the objects that your system needs to deal with (Molly for example, student, parent, donation, alumni; Alice for example, student, event)

A **class** is a template for creating these objects. The class identifies what data needs to be stored (fields) for objects of this type and what methods are available to allow access to this data.

For example: The following boxes show example classes. The top half of the box shows the fields and the bottom half the methods



The constructor method of the class has the same name as the class.

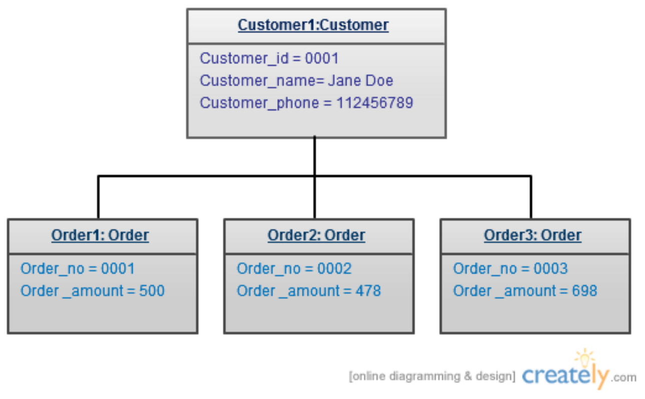
In the execution of a program the word object has a slightly different meaning. Here, an object is created by a constructor of the class, the resulting object is called an *instance* of the class. So in this example the class is a template for a lottery draw but you would have a new lottery object each Saturday for each new draw, each draw object having its own set of winning numbers for example. Similarly, you can generate lots of lottery ticket objects. They all follow the same structure but would have different numbers assigned to each object.

Molly and Alice in your IA, you had a class student. You then created a number of instances of this class (objects) called Angus, Theo, Ruby etc. They all had addresses, parents, houses, numEvents etc but they were all different. An instance of a class is when the program has allocated memory to hold the object i.e. the object exists at runtime (Instantiation).

D1.3 & D1.4 UML Diagrams and their interpretation

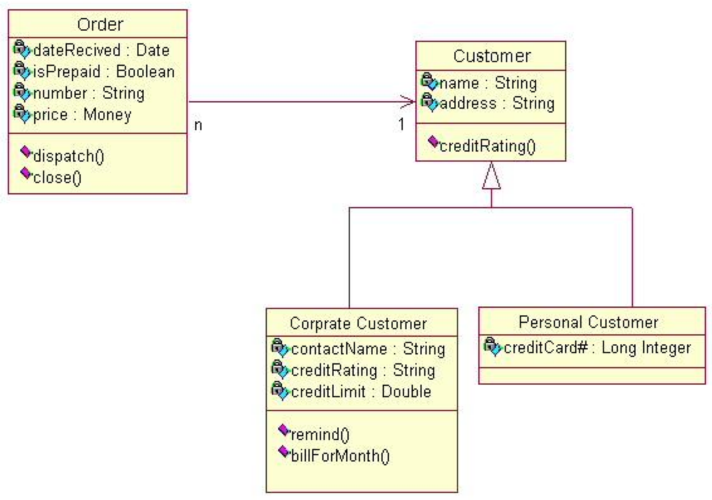
This is simply asking you to be able to draw object and class diagrams like in the previous example and in your IA. It is the way in which we visualise our computational thinking and program design.

Object Diagram



In this example the Customer Object ‘Customer1’ is associated with 3 order objects Order1, Order2 and Order 3.

Class Diagram

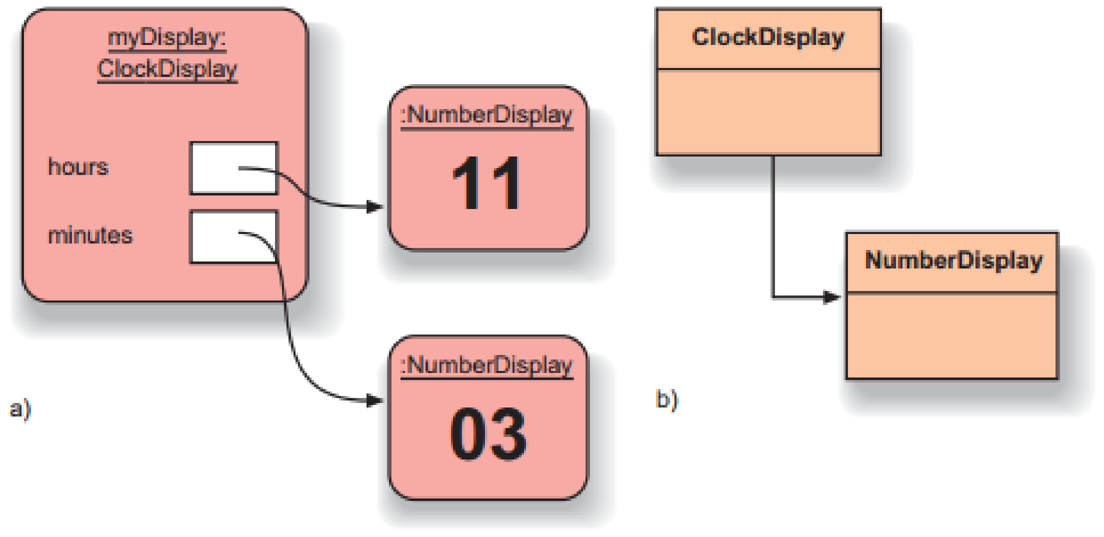


In this class diagram you have an Order class with fields: dateReceived of type Date; isPrepaid of type Boolean; number of type String and price of type Money. It has methods called: dispatch() and close(). You don’t know how those methods work you just know that you have access to them. The name of the methods should indicate what the method does.

You also have a class called Customer. All customers have a name field and an address field and a method called creditRating(). The classes Corporate Customer and Personal Customer are types of customer so the diagram shows (by the arrows) that they *INHERIT* all the fields and methods of the customer class. On top of this they have their own specific fields and methods which they do not share.

D1.5 Describe the process of decomposition into several related objects

This we did in year 12 when we looked at the digital clock in java. The clock is made of 2 two digit number displays so we created a class numberDisplay and a class clockDisplay which consisted of 2 numberDisplay objects. (ABSTRACTION)



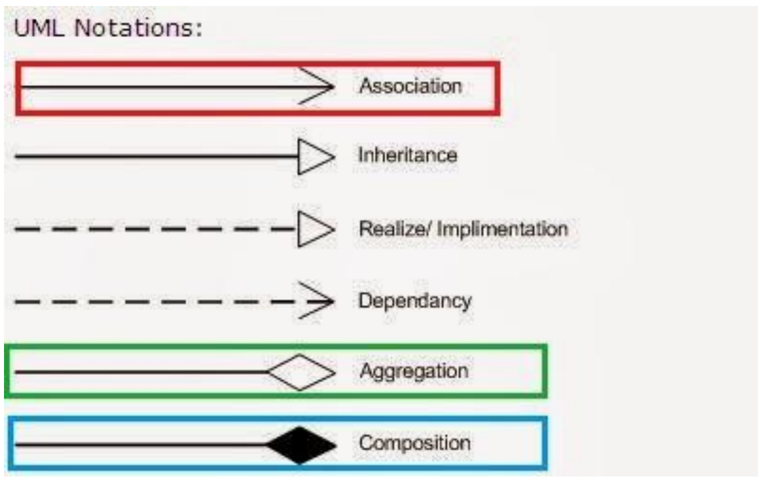
As another example, you may have a real-world object of a calendar. The calendar is made up of days which can be grouped into weeks which can be grouped into months etc. You have decomposed the calendar into days, weeks and months; its component parts.

D1.6 Describe the relationship between objects for a given problem

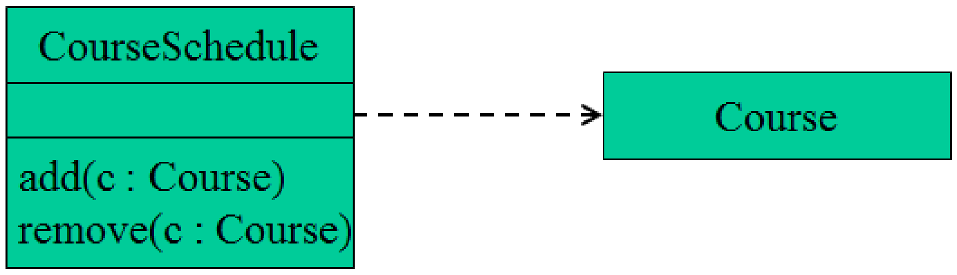
Objects can have 3 types of relationships:

1. Dependency ("uses")
2. Aggregation ("has a")
3. Inheritance ("is a")

In UML the notation looks like this:

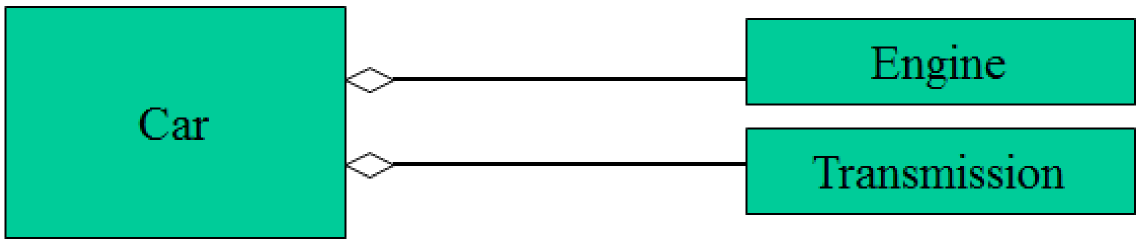


Dependency



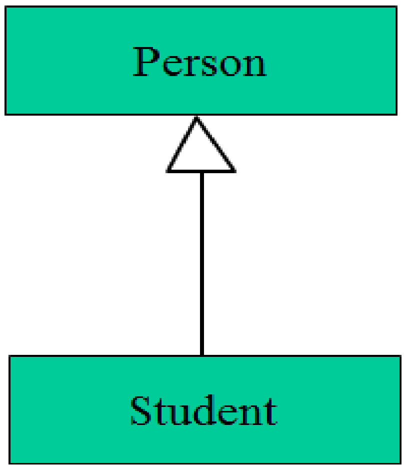
Here, the courseSchedule class depends on the course class because the add() and remove() **methods** both use the Course class.

Aggregation



Here, the Car class includes **fields** of type Engine and Transmission classes. The Engine class and Transmission class objects can exist in isolation of the car class.

Inheritance



A Class can inherit fields and methods from a superclass. Here the Student inherits fields and methods from the class Person. For example, a Person might have a name and an age with methods getAge() and setAge(). A student would have access to these but might have additional fields and methods like StudentNumber or Course and getStudentNumber() and enrolledOnCourse() that are not applicable to all 'Persons'.

D1.7 Outline the need to reduce dependencies between objects in a given problem

What is a Dependency?

Whenever a class A uses another class B, then A depends on B. A cannot carry out its work without B, and A cannot be reused without also reusing B. In such a situation the class A is called the "dependant" and the class or interface B is called the "dependency". A dependant depends on its dependencies.

Two classes that use each other are called "coupled". The coupling between classes can be loose or tight, or somewhere in between. The tightness of a coupling is not binary. It is not either "loose" or "tight". The degrees of tightness are continuous, not discrete. You can also characterize dependencies as "strong" or "weak". A tight coupling leads to strong dependencies, and a loose coupling leads to weak dependencies, or even no dependencies in some situations.

Dependencies, or couplings, are directional. That A depends on B doesn't mean that B also depends on A.

Why are Dependencies Bad?

Dependencies are bad because they decrease reuse. Decreased reuse is bad for many reasons. Often reuse has positive impact on development speed, code quality, code readability etc.

How dependencies can hurt reuse is best illustrated by an example:

Imagine you have a class CalendarReader that is able to read a calendar event list from an XML file. The implementation of CalendarReader is sketched below:

```
public class CalendarReader {

    public List readCalendarEvents(File calendarEventFile){

        //open InputStream from File and read calendar events.

    }

}
```

The method readCalendarEvents takes a File object as parameter. Thus this method depends on the File class. This dependency on the File class means that the CalendarReader is capable only of reading calendar events from local files in the file system. It cannot read calendar event files from a network connection, a database or from a resource on the classpath. You can say that the CalendarReader is tightly coupled to the File class and thereby the local file system.

A less tightly coupled implementation would be to exchange the File parameter with an InputStream parameter, as sketched below:

```
public class CalendarReader {

    public List readCalendarEvents(InputStream calendarEventFile){

        //read calendar events from InputStream

    }

}
```

As you may know, an InputStream can be obtained from either a File object, a network Socket, a URLConnection class, a Class object (Class.getResourceAsStream(String name)), a column in a database via JDBC etc. Now the CalendarReader is not coupled to the local file system anymore. It can read calendar event files from many different sources.

With the InputStream version of the readCalendarEvents() method the CalendarReader has become more reusable. The tight coupling to the local file system has been removed. Instead it has been replaced with a dependency on the InputStream class. The InputStream dependency is more flexible than the File class dependency, but that doesn't mean that the CalendarReader is 100% reusable. It still cannot easily read data from a NIO Channel, for instance.

(ref: http://tutorials.jenkov.com/ood/understanding-dependencies.html#whatis)

There are other overheads with dependencies in terms of maintaining the code, you need to be very careful that you have considered the impact of code changes on **all** classes involved in the dependency relationship to retain the integrity of your program. This can increase the length of time the work takes therefore incurring more costs in the production of the code. There is greater room for error and human oversight etc.

D1.8 Construct related objects for a given problem

In an exam you will be presented with a scenario (problem) and be asked to identify the objects involved and the relationships between them. You may be asked to represent this in a UML diagram as described above.

D1.9 Explain the need for different data types to represent data items

Integer

Used to represent whole numbers. 32-bits (4 bytes) used to store the information. It is stored as 2s complement signed integer so can take values from -2^{31} and a maximum value of $2^{31}-1$.

Real
Used to represent a decimal number. Java uses ‘double’ or ‘float’ for this. The difference between ‘double’ and ‘float’ is the amount of memory they use to store the number. ‘Float’ uses 32-bits and ‘double’ uses 64-bits. The double is the default type in java but ‘float’ should be used for arrays of real numbers in order to save storage space.

String
Used to represent a sequence of characters like a name. (In Java, string is a class not a data type as it has a set of associated methods.) Each character is stored in a **byte** of memory

Boolean
Used to represent data that can only take two values e.g. true/false; on/off; male/female. It occupies 1 **bit** of data in memory

We need different data types because we have different sorts of data. All data must be converted into binary in order for it to be stored and processed by the computer. Each data types ‘tells’ the computer how to deal with the data it is given and how much memory space the data will use. By choosing the correct data type we can control how much memory our program will use.

D1.10 Describe how data can be passed to and from actions as parameters

By actions they mean methods. Some methods, in order to complete its processing, will need data that it does not have direct access to. This data can be ‘sent into’ the method as an argument and accepted as a parameter:

```
public class PassPrimitiveByValue {

    public static void main(String[] args) {

        int x = 3;

        // invoke passMethod() with
        // x as argument
        passMethod(x);

        // print x to see if its
        // value has changed
        System.out.println("After invoking passMethod, x = " + x);

    }

    // change parameter in passMethod()
    public static void passMethod(int p) {
        p = p+ 10;
    }
}
```
When you run this program, the output is:

```
After invoking passMethod, x = 3
```

In this program x was sent in from the main program as an ARGUMENT to the method passMethod(). The passMethod() method then accepts it as PARAMETER ‘p’. It completes its task using p but p remains in the method and doesn’t get sent back to the main program (i.e. x does not change value).

If the main program wanted the new value of p returned the code would look like this:

```
public class PassPrimitiveByValue {

    public static void main(String[] args) {

        int x = 3;
        int y = 0;

        // invoke passMethod() with
        // x as argument
        y = passMethod(x);

        // print y to see if its
        // value has changed
        System.out.println("After invoking passMethod, y = " + y);

    }

    // change parameter in passMethod()
    public static int passMethod(int p) {
        p = p + 10;
        return p;
    }
}
```

When you run this program, the output is:

```
After invoking passMethod, y = 13
```

D2 – Features of OOP

D2.1 & D2.4 Define the term encapsulation, explain the advantages of encapsulation

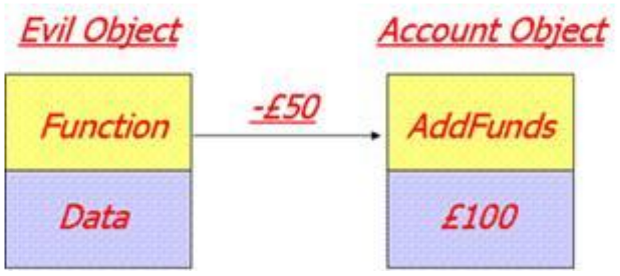
Encapsulation

Data and Methods are limited to the object in which they are defined. This is simply the Class. Inside a class you have a number of private fields. These fields are only available to other classes via the methods the class provides, they can’t be accessed directly. This concept is Encapsulation.

If you think of an object that is an instance of a class called File, then the object is likely to provide methods to open, close and read a file. Other objects using these methods do not know how the method is implemented(how it works internally). So in the case of the File example the calling object does not know how the file object will open the file, it just knows that it will. This is akin to a managing director of a business asking a software developer to add some new services to the company website. The director isn't interested in how the developer does it they just want the services to be added.

Advantages of Encapsulation

Let's imagine we have two objects, the first we will call "evil object" and the second object we will call "account object". The account object has a data attribute called "balance". If this attribute is public, allowing other objects to manipulate its value then the evil object can change it to whatever it wants. This means you could go from having £100 in your account to having £0 in your account. To address this situation we can make the balance attribute private and add a method called "AddFunds". This means that now the account balance can only go up, or does it? Have a look at the following diagram.



The evil object is invoking the AddFunds method but is passing a value of -£50. This means that instead of adding funds the balance would actually go down. We can address this issue by building checks into the AddFunds method, for example we could use the following logic:

Method: ADDFUNDS, Parameter: AMOUNT

```
IF THE AMOUNT IS GREATER THAN 0 THEN
      ADD THE FUNDS
ELSE
      LOG ERROR
END IF
```

Now we have some control on how the data can be accessed and modified. This type of check would not be possible with public data and direct modification access.

In summary, encapsulation is both information-hiding and providing clearly defined public methods. The public methods may be invoked by other objects and the data is kept hidden by marking it as private.

The shielding of the data means that should the format of the private data need to change for some reason the objects calling the methods on the object whose data has changed will not need to change the way they call the method. For example, if we imagine that we needed to change a data attribute from a type float to a type double. (Both represent real numbers but the double type can typically store larger values.) If a calling object is updating the value directly then by changing the data type you will also need to change the code in the calling object. If you are dealing with large systems with hundreds of classes this type of situation can quickly get out of hand with several layers of "knock-on" changes required. Therefore we would say that encapsulation promotes maintenance because code changes can be made independently without affecting other classes.

From a software development perspective by having well defined public methods it allows other developers to quickly understand your code and reuse it in other applications.

In summary the benefits of encapsulation are:

- Encapsulation promotes maintenance
- Code changes can be made independently
- Increases usability

(ref:
http://www.cems.uwe.ac.uk/~jsa/UMLJavaShortCourse09/CGOutput/Unit3/unit3(0809)/page_12.htm)

D2.2 & D2.5 Define the term inheritance, explain the advantages of inheritance

Inheritance

A parent object holds common fields and methods for a group of related child objects. For example, a Person Class will hold common fields and methods for a Student class and a Teacher class.

For example:

```
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

A class declaration for a `MountainBike` class that is a subclass of `Bicycle` might look like this:

```
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight,
                        int startCadence,
                        int startSpeed,
                        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```
`MountainBike` inherits all the fields and methods of `Bicycle` and adds the field `seatHeight` and a method to set it. Except for the constructor, it is as if you had written a new `MountainBike` class entirely from scratch, with four fields and five methods. However, you didn't have to do all the work. This would be especially valuable if the methods in the `Bicycle` class were complex and had taken substantial time to debug.

Advantages of Inheritance

One of the key benefits of inheritance is to minimize the amount of duplicate code in an application by sharing common code amongst several subclasses. Where equivalent code exists in two related classes, the hierarchy can usually be refactored to move the common code up to a mutual superclass. This also tends to result in a better organization of code and smaller, simpler compilation units.

Reusability - facility to use public methods of superclass class without rewriting the same code
Overriding – Methods can be implemented in the subclass that override methods of the same name in the superclass to make the methods more suitable for the subclass.
Data hiding - Superclass can decide to keep some data private so that it cannot be altered by the subclass.

Disadvantages (Not asked for in the spec but out of interest!)

1.One of the main disadvantages of inheritance in Java (the same in other object-oriented languages) is the increased time/effort it takes the program to jump through all the levels of overloaded classes. If a given class has ten levels of abstraction above it, then it will essentially take ten jumps to run through a function defined in each of those classes

2.Main disadvantage of using inheritance is that the two classes (super and subclass) get tightly coupled. This means one cannot be used independently of the other.

3. Also with time, during maintenance adding new features both base as well as derived classes are required to be changed. If a method signature is changed then we will be affected in both cases.

4. If a method is deleted in the "super class" or aggregate, then we will have to re-factor in case of using that method. Here things can get a bit complicated in case of inheritance because our programs will still compile, but the methods of the subclass will no longer be overriding superclass methods. These methods will become independent methods in their own right.

(ref: http://erpbasic.blogspot.co.uk/2012/01/inheritance-advantages-and.html)

D2.3 & D2.6 Define the term polymorphism, explain the advantages of polymorphism

Polymorphism

Methods have the same name but different parameter lists and processes. Polymorphism can be demonstrated with a minor modification to the `Bicycle` class. For example, a `printDescription` method could be added to the class that displays all the data currently stored in an instance(object of that class).

```
public void printDescription(){
    System.out.println("\nBike is " + "in gear " + this.gear
        + " with a cadence of " + this.cadence +
        " and travelling at a speed of " + this.speed + ". ");
}
```

To demonstrate polymorphic features in the Java language, extend the `Bicycle` class with a `MountainBike` and a `RoadBike` class. For`MountainBike`, add a field for `suspension`, which is a `String` value that indicates if the bike has a front shock absorber, `Front`. Or, the bike has a front and back shock absorber, `Dual`.

Here is the updated class:

```
public class MountainBike extends Bicycle {
    private String suspension;

    public MountainBike(
               int startCadence,
               int startSpeed,
               int startGear,
               String suspensionType){
        super(startCadence,
              startSpeed,
              startGear);
        this.setSuspension(suspensionType);
    }

    public String getSuspension(){
      return this.suspension;
    }

    public void setSuspension(String suspensionType) {
        this.suspension = suspensionType;
    }

    public void printDescription() {
        super.printDescription();
        System.out.println("The " + "MountainBike has a" +
            getSuspension() + " suspension.");
    }
}
```

Note the overridden `printDescription` method. In addition to the information provided before, additional data about the suspension is included to the output.

Next, create the `RoadBike` class. Because road or racing bikes have skinny tires, add an attribute to track the tire width. Here is the `RoadBike`class:

```
public class RoadBike extends Bicycle{
    // In millimeters (mm)
    private int tireWidth;

    public RoadBike(int startCadence,
                    int startSpeed,
                    int startGear,
                    int newTireWidth){
        super(startCadence,
              startSpeed,
              startGear);
        this.setTireWidth(newTireWidth);
    }

    public int getTireWidth(){
      return this.tireWidth;
    }

    public void setTireWidth(int newTireWidth){
        this.tireWidth = newTireWidth;
    }

    public void printDescription(){
        super.printDescription();
        System.out.println("The RoadBike" + " has " + getTireWidth() +
            " MM tires.");
    }
}
```

Note that once again, the `printDescription` method has been overridden. This time, information about the tire width is displayed.

To summarize, there are three classes: `Bicycle`, `MountainBike`, and `RoadBike`. The two subclasses override the `printDescription` method and print unique information.

Here is a test program that creates three `Bicycle` variables. Each variable is assigned to one of the three bicycle classes. Each variable is then printed.

```
public class TestBikes {
  public static void main(String[] args){
    Bicycle bike01, bike02, bike03;

    bike01 = new Bicycle(20, 10, 1);
    bike02 = new MountainBike(20, 10, 5, "Dual");
    bike03 = new RoadBike(40, 20, 8, 23);

    bike01.printDescription();
    bike02.printDescription();
    bike03.printDescription();
  }
}
```

The following is the output from the test program:

```
Bike is in gear 1 with a cadence of 20 and travelling at a speed of 10.

Bike is in gear 5 with a cadence of 20 and travelling at a speed of 10.
The MountainBike has a Dual suspension.

Bike is in gear 8 with a cadence of 40 and travelling at a speed of 20.
The RoadBike has 23 MM tires.
```

(ref: http://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html)

Advantages of Polymorphism

Maybe you are a car collector, like Jay Leno. You want to have dozens and dozens of cars. Some will be porsches, some corvettes, some yugos. It would be nice if you could put them all into a list.

Now, Java only lets you put one type of thing into a *collection (you need to understand this data structure)*. You don't want to have a porsche list, a corvette list, and a yugo list - you want a list of cars - so that's what you do, create a superclass of car and subclasses for each car type (inheritance). Then, since all of these ARE cars, you can put them all into the collection.

Then you want to start each and every car once a week (to keep it in good condition). You want to iterate through your collection and call each car's start() method. You can do this:

```
for (Car c : carCollection)
{
    c.start();
}
```

There are several beautiful things about this. First, you don't need to code it like this:

```
//not actually java, but pseudo-code

for (Object o : collection)
{
   if o is a porsche
       Porsche p = (Porsche)o;
       p.start();
   else if o is a Yugo
       Yugo y = (Yugo)o;
       y.start();
   else...
}
```

Look at these two code examples. Now imagine that after a few years, we decide to buy a BMW. In the first example, you don't need to touch the code, and it will handle a BMW just fine.

This means that the first code example can hand object types THAT DIDN'T EXIST when you originally wrote it. The less you have to touch code, the less likely you are to introduce a bug. The second code example would need to be updated each and every time I want to buy a new type of car. Not good!

The Polymorphism here is in the implementation of the start() method which would be different for each type of car. The calling program however, sees it as the same (implementation is hidden)

(ref: http://www.coderanch.com/t/548649/java-programmer-SCJP/certification/point-advantage-polymorphism)

D2.7 The Advantages of Libraries of Objects

Libraries are made available for code that is likely to be re-used. Examples of this are standard search and sorting algorithms or file handling methods. By providing the code in a library the programmer does not have to keep re-inventing the wheel. The code that is contained in the library has been tested and is reliable. In java libraries are used in the code by ‘importing’ them at the top of a class e.g.

```
import java.lang.Math.*;
```

D2.8 Describe the Disadvantages of OOP

Object Oriented Programming has several disadvantages which made it unpopular in the early years.

Size: Object Oriented programs are much larger than other programs. In the early days of computing, space on hard drives, floppy drives and in memory was at a premium. Today we do not have these restrictions.

Effort: Object Oriented programs require a lot of work to create. Specifically, a great deal of planning goes into an object oriented program well before a single piece of code is ever written. Initially, this early effort was felt by many to be a waste of time. In addition, because the programs were larger (see above) coders spent more time actually writing the program.

Speed: Object Oriented programs are slower than other programs, partially because of their size. Other aspects of Object Oriented Programs also demand more system resources, thus slowing the program down.

Not suitable for all types of problems: There are problems that lend themselves well to functional-programming style, logic-programming style, or procedure-based programming style, and applying object-oriented programming in those situations will not result in efficient programs.

In recent years, however, improvements in computer performance have made restrictions about size and speed inconsequential. The question of human effort still exists, however; many novice programmers do not like Object Oriented Programming because of the great deal of work required to produce minimal results.

Advantages of OOP (not in the spec but of interest)

Object Oriented Programming has great advantages over other programming styles:

- ***Code Reuse and Recycling***: Objects created for Object Oriented Programs can easily be reused in other programs.
- ***Encapsulation (part 1)***: Once an Object is created, knowledge of its implementation is not necessary for its use. In older programs, coders needed understand the details of a piece of code before using it (in this or another program).

- ***Encapsulation (part 2)***: Objects have the ability to hide certain parts of themselves from programmers. This prevents programmers from tampering with values they shouldn't. Additionally, the object controls how one interacts with it, preventing other kinds of errors. For example, a programmer (or another program) cannot set the width of a window to -400.

- ***Design Benefits***: Large programs are very difficult to write. Object Oriented Programs force designers to go through an extensive planning phase, which makes for better designs with less flaws. In addition, once a program reaches a certain size, Object Oriented Programs are actually *easier* to program than non-Object Oriented ones.

- ***Software Maintenance:*** Programs are not disposable. Legacy code must be dealt with on a daily basis, either to be improved upon (for a new version of an exist piece of software) or made to work with newer computers and software. An Object Oriented Program is much easier to modify and maintain than a non-Object Oriented Program. So although a lot of work is spent before the program is written, less work is needed to maintain it over time.

D2.9 Discuss the use of programming teams

This comparison is to programmers working alone. Think about how OOP allows you to split a system into objects that are defined by their own class. Each object could be given to a different person to program. As long as other programmers know what methods are available from this class if they need to make use of them, they can each program their class at the same time.

This way of splitting the work up obviously needs careful planning and good communication between team members. Its success also relies on everyone pulling their weight as at some point the system will have to be pieced together for testing and you might start to get bottlenecks if team members have to wait for others to complete before they can proceed.

Benefits

- Attack bigger problems in a short period of time
- Utilize the collective experience of everyone (people can work to their strengths)

Risks

- Communication and coordination issues
- Groupthink: diffusion of responsibility; going along
- Working by inertia; not planning ahead
- Conflict or mistrust between team members

D2.10 Explain the advantages of modularity in program development

This means a program is split into smaller sections (modules) that can be tackled individually. Once each module is working successfully the modules are put together. This keeps debugging very focussed and quick as you are only working on a small section of code at a time. Testing is quicker because you test each module individually and only present it for integration into the system when you are sure it works properly so system testing should be pretty straight forward. Modules can be allocated to different people (see above) so completion can be quicker too.

D3 Program Development

D3.1, 3.2 & 3.3 Define Terms

Class: Defines the fields and methods for a group of similar objects. An object is an instance of a class.

Identifier: A broadbrush name for the name of a variable or method.

Primitive: (data types) byte, int, long, double, char, boolean

Instance variable: The fields in a class. So when an object is created of that class type, it has its own version of the fields (instance variables)

Parameter variable: Information sent into a method from the calling program e.g

If you had a method declared as:

```
public int add(int x, int y)
{
        Return x+y
}
```

The calling problem might look something like this:

Add(firstNum,secondNum)

Here firstNum & secondNum are all parameter variables

(x and y are called *arguments* of the method)

Local variable: A local variable exists only as long as the code to which it belongs exist. So if it is declared in a method, when the method is no longer in use during the run of a program, the variable no longer exists(memory is not used to store them anymore, its overwritten). The arguments are used as local variables.

Method: A program routine contained within an object designed to perform a particular task on the fields within the object.

Accessor: A method that allows access to the fields within an object e.g. getName()

Mutator: A method that allows the values of the fields in an object to be changed e.g. setName()

Constructor: The method that correctly initialises and sets up an object when it is created. This method has the same name as the class it belongs to and no return datatype e.g. public person()

Signature: The method name and its parameter types.

For example, the following two methods have distinct signatures:

```
doSomething(int y);
doSomething(String y);
```

The following three methods do have the same signatures and are considered the same, as only the return value differs. The name of the parameter is not part of the method signature and is ignored by the compiler for checking method uniqueness.

```
int doSomething(int y)
String doSomething(int x)
int doSomething(int z) throws java.lang.Exception
```

Return Value: The value returned by a method to the calling program

Private: Methods, Fields and Constructors that are declared private can only be accessed within the declared class itself. Private access modifier is the most restrictive access level. Classes cannot be private. Fields that are declared private can be accessed outside the class if public accessor methods are present in the class. Using the private modifier is the main way that an object encapsulates itself and hides its data from the outside world.

Protected: Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class. The protected access modifier cannot be applied to classes. Methods and fields can be declared protected.

Public: A class, method, constructor etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe. Because of class inheritance, all public methods and fields of a class are inherited by its subclasses.

Extends: This keyword is used by a subclass to say that it inherits the fields and methods from a superclass (see inheritance earlier)

Static: In Java, a static member is a member of a class that isn't associated with an instance of a class. Instead, the member belongs to the class itself. As a result, you can access the static member without first creating a class instance.

The two types of static members are static fields and static methods:

- Static field: A field that's declared with the `static` keyword, like this:

  ```
  private static int ballCount;
  ```

  The value of a static field is the same across all instances of the class. In other words, if a class has a static field named `CompanyName`, all objects created from the class will have the same value for `CompanyName`.

  Static fields are created and initialized when the class is first loaded. That happens when a static member of the class is referred to or when an instance of the class is created, whichever comes first.

- Static method: A method declared with the static keyword. Like static fields, static methods are associated with the class itself, not with any particular object created from the class. As a result, you don't have to create an object from a class before you can use static methods defined by the class.

The best-known static method is `main`, which is called by the Java runtime to start an application. The `main` method must be `static`, which means that applications run in a static context by default.

One of the basic rules of working with static methods is that you can’t access a nonstatic method or field from a `static` method because the `static` method doesn’t have an instance of the class to use to reference instance methods or fields.

D3.4 Describe the uses of the primitive data types and the reference class string

Byte:

- Byte data type is an 8-bit signed two's complement integer.
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
- Example: byte a = 100 , byte b = -50

Int:

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648.(-2^31)
- Maximum value is 2,147,483,647(inclusive).(2^31 -1)
- Int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.
- Example: int a = 100000, int b = -200000

Long:

- Long data type is a 64-bit signed two's complement integer.
- Minimum value is -9,223,372,036,854,775,808.(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
- This type is used when a wider range than int is needed.
- Default value is 0L.
- Example: long a = 100000L, int b = -200000L

Double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values, generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example: double d1 = 123.4

Boolean:

- boolean data type represents one bit of information.
- There are only two possible values: true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example: boolean one = true

Char:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example: char letterA ='A'

The String reference class

Rather than having a simple data type string, Java has a class String. This means that Java provides a number of methods that can be used to manipulate Strings which is really helpful.

The commonly-used method in the String class are summarized below. Refer to the JDK API for java.lang.String a complete listing.

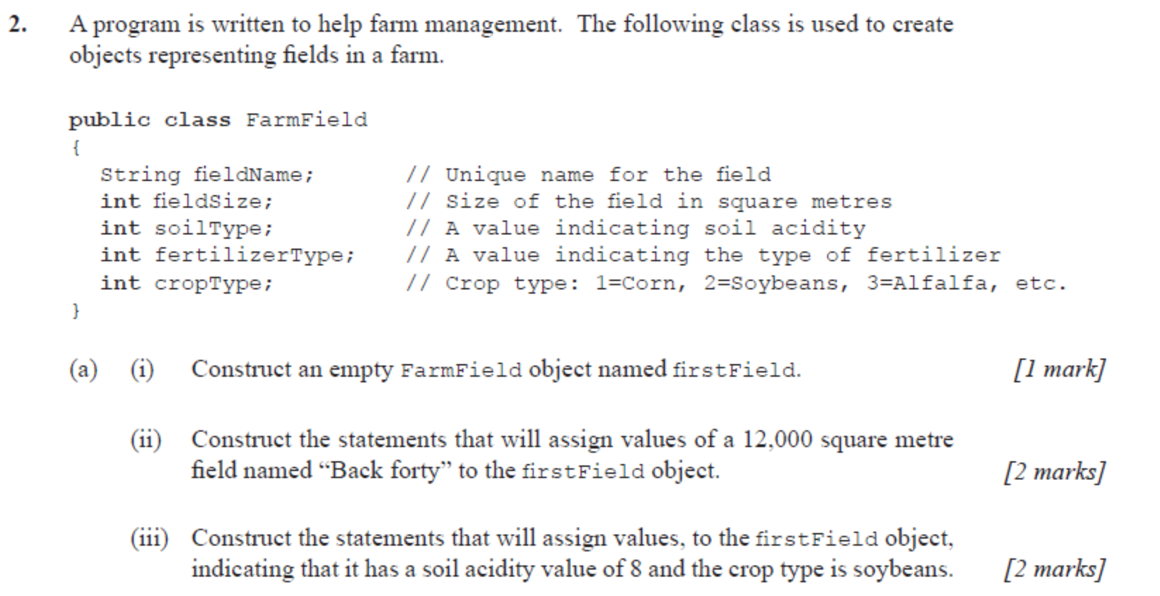
```
// Length
int length()       // returns the length of the String
boolean isEmpty()  // same as thisString.length == 0

// Comparison
boolean equals(String another) // CANNOT use '==' or '!=' to compare two Strings
in Java
boolean equalsIgnoreCase(String another)
int compareTo(String another)  // return 0 if this string is the same as another;
                               // <0 if lexicographically less than another; or >0
```

```
int compareToIgnoreCase(String another)
boolean startsWith(String another)
boolean startsWith(String another, int fromIndex)  // search begins at fromIndex
boolean endsWith(String another)

// Searching & Indexing
int indexOf(String search)
int indexOf(String search, int fromIndex)
int indexOf(int character)
int  indexOf(int  character,  int  fromIndex)        //  search  forward  starting  at
fromIndex
int lastIndexOf(String search)
int  lastIndexOf(String  search,  int  fromIndex)   //  search  backward  starting  at
fromIndex
int lastIndexOf(int character)
int lastIndexOf(int character, int fromIndex)

// Extracting a char or part of the String (substring)
char charAt(int index)              // index from 0 to String's length - 1
String substring(int fromIndex)
String substring(int fromIndex, int endIndex)  // exclude endIndex

// Creating a new String or char[] from the original (Strings are immutable!)
String toLowerCase()
String toUpperCase()
String trim()          // create a new String removing white spaces from front and
back
String replace(char oldChar, char newChar)  // create a new String with oldChar
replaced by newChar
String concat(String another)               // same as thisString + another
char[] toCharArray()                        // create a char[] from this string
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)  // copy into
dst char[]

// Static methods for converting primitives to String
static String ValueOf(type arg)  // type can be primitives or char[]

// Static method resulted in a formatted String using format specifiers
static  String  format(String  formattingString,  Object...  args)     //  same  as
printf()

// Regular Expression (JDK 1.4)
boolean matches(String regexe)
String replaceAll(String regexe, String replacement)
String replaceAll(String regexe, String replacement)
String[] split(String regexe)                // Split the String using regexe as
delimiter,
                                          // return a String array
String[] split(String regexe, int count)  // for count times only
```

D3.5 Construct Code to implement sections D3.1-3.4

In the exam you will be expected to write Java code using all of the above to show that you understand them. You may be given some example code and asked to identify parts of it like the constructor. You may also be asked to trace through a piece of code. This means to draw a table with a column for each variable and then write in the values that the variables take as the code is executed line by line.

For example:



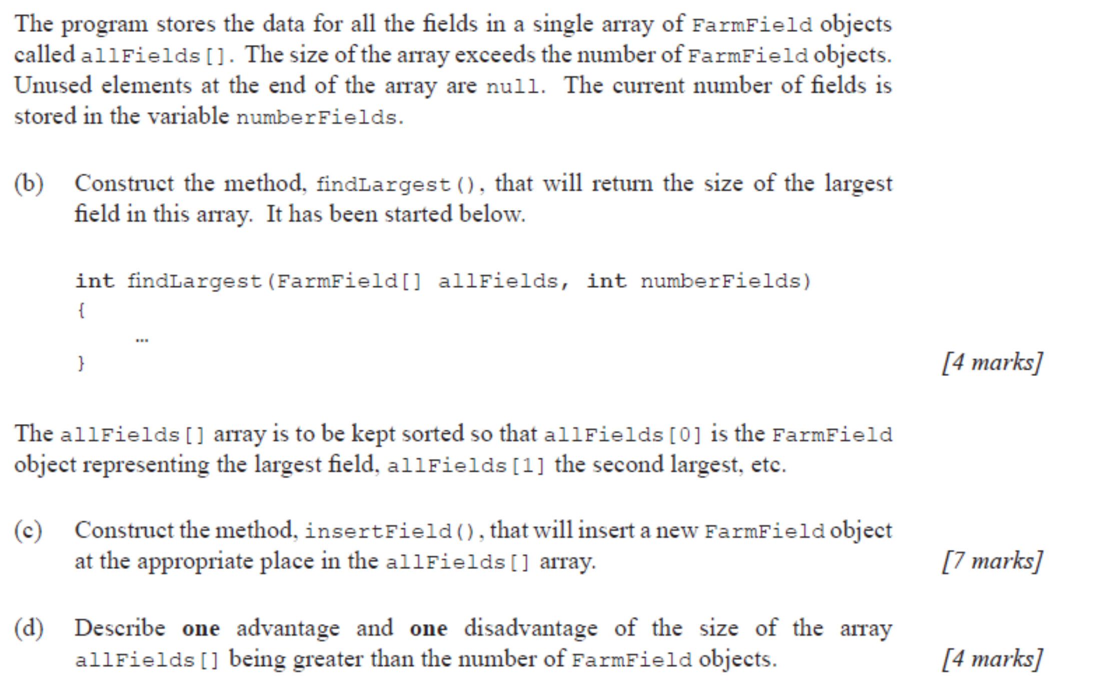
2. A program is written to help farm management. The following class is used to create objects representing fields in a farm.

```
public class FarmField
{
  String fieldName;       // Unique name for the field
  int fieldSize;          // Size of the field in square metres
  int soilType;           // A value indicating soil acidity
  int fertilizerType;     // A value indicating the type of fertilizer
  int cropType;           // Crop type: 1=Corn, 2=Soybeans, 3=Alfalfa, etc.
}
```

(a) (i) Construct an empty `FarmField` object named `firstField`. *[1 mark]*

(ii) Construct the statements that will assign values of a 12,000 square metre field named "Back forty" to the `firstField` object. *[2 marks]*

(iii) Construct the statements that will assign values, to the `firstField` object, indicating that it has a soil acidity value of 8 and the crop type is soybeans. *[2 marks]*



The program stores the data for all the fields in a single array of `FarmField` objects called `allFields[]`. The size of the array exceeds the number of `FarmField` objects. Unused elements at the end of the array are `null`. The current number of fields is stored in the variable `numberFields`.

(b) Construct the method, `findLargest()`, that will return the size of the largest field in this array. It has been started below.

```
int findLargest(FarmField[] allFields, int numberFields)
{
    …
}
```

[4 marks]

The `allFields[]` array is to be kept sorted so that `allFields[0]` is the `FarmField` object representing the largest field, `allFields[1]` the second largest, etc.

(c) Construct the method, `insertField()`, that will insert a new `FarmField` object at the appropriate place in the `allFields[]` array. *[7 marks]*

(d) Describe **one** advantage and **one** disadvantage of the size of the array `allFields[]` being greater than the number of `FarmField` objects. *[4 marks]*

D3.7 Construct code examples related to selection statements

In the exam you will be expected to trace, explain and write Java code that makes use of simple and nested IF statements.

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

     int user = 17;

     if (user <= 18) {
         System.out.println("User is 18 or younger");
     }
     else {
         System.out.println("User is older than 18");
     }
}
```

Or:

```
if ( num > 90 )
{
  System.out.println( "You earned an A" ) ;
}
else
   if ( num > 80 )
   {
      System.out.println( "You earned a B" ) ;
   }
   else
      if ( num > 70 )
      {
         System.out.println( "You earned a C" ) ;
      }
```

D3.7 Construct code examples related to repetition statements

In the exam you will be expected to trace, explain and write Java code that makes use of while loops and for loops. You need to understand the difference between the two.

The For Loop

This loop is executed a finite number of time. Its form is:

```
for(initialization; Boolean_expression; update)
{
   //Statements
}
```

The loop is controlled by a value. The initialization says what value this controller starts at. The Boolean expression says under what condition the loop will stop and the update says how to update the controller (usually increment by 1)

Here is the flow of control in a for loop:

- The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

- Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.

- After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.

- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

<u>Example:</u>

```
public class Test {

   public static void main(String args[]) {

      for(int x = 10; x < 20; x = x+1) {
         System.out.print("value of x : " + x );
         System.out.print("\n");
      }
   }
}
```

This would produce the following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

The While Loop

This loop continues while a Boolean expression is true. There is no knowing how many times the loop will be executed prior to run time. This loop might never be executed if the expression is false at the start of the loop.

The form of a while loop is:

```
while(Boolean_expression)
{
   //Statements
}
```

Example:

```
public class Test {

   public static void main(String args[]) {
      int x = 10;

      while( x < 20 ) {
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");
      }
   }
}
```

This would produce the following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

If instead of ‘x++’ the code said ‘x=x+y’ where y was another variable it may get to be bigger than 20 very quickly or slowly depending on the value of y. You would then have more or less executions of the loop.

The Do..While loop

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

The form of a do...while loop is:

```
do
{
   //Statements
}while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Example:

```
public class Test {

   public static void main(String args[]){
      int x = 10;

      do{
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");
      }while( x < 20 );
   }
}
```

This would produce the following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

D3.8 Construct Code examples related to Static Arrays

In the exam you will be expected to trace, explain and write Java code that makes use of static arrays. Not 100% sure what they mean by static array as it could be interpreted two ways; they could mean

1. The size of an array is fixed (static) when it is declared (as opposed to an ArrayList whose size is dynamic i.e. changes at runtime). In this case you have problems with wasted space if the whole of the array is not being used during runtime for most of the time. You have to set the size of an array to the maximum that would possibly be needed to avoid 'array out of bounds' errors. However they are easier to understand, implement and maintain from a programmers viewpoint.

   Example:

```
        String[] suit = new String[] {
                "spades", "hearts", "diamonds", "clubs"};
```

2. The array is declared as a static variable which means that the array belongs to the class not the object just as static variables explained previously.

Just make sure you understand both interpretations.

D3.9 Discuss the features of modern programming languages that enable internationalisation

Unicode is a computing industry standard for the consistent encoding, representation and handling of text expressed in most of the world's writing systems. Developed in conjunction with the Universal Character Set standard and published in book form as *The Unicode Standard*, the latest version of Unicode contains a repertoire of more than 110,000 characters covering 100 scripts and various symbols. The standard consists of a set of code charts for visual reference, an encoding method and set of standard character encodings, a set of reference data computer files, and a number of related items, such as character properties, rules for normalization, decomposition, collation, rendering, and bidirectional display order (for the correct display of text containing both right-to-left scripts, such as Arabic and Hebrew, and left-to-right scripts).[1] As of September 2013, the most recent version is *Unicode 6.3*. The standard is maintained by the Unicode Consortium.

Unicode's success at unifying character sets has led to its widespread and predominant use in the internationalization and localization of computer software. The standard has been implemented in many recent technologies, including modern operating systems, XML, the Java programming language, and the Microsoft .NET Framework.

See also: http://en.wikipedia.org/wiki/Internationalization_and_localization

D3.10 Discuss the ethical and moral obligations of programmers

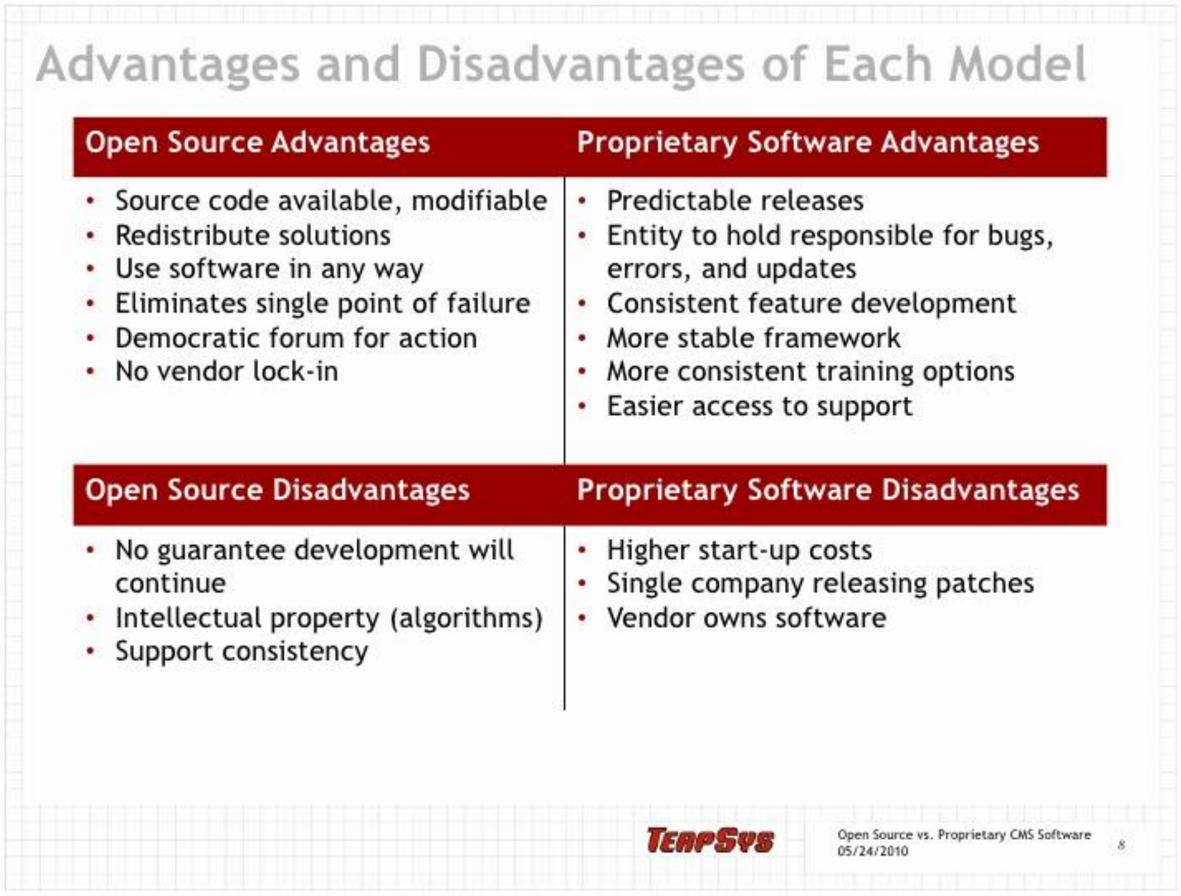
Be able to consider things like

- To what extent should programs be tested to avoid damage to people, government or things. If it goes wrong who is responsible, the coder, the tester, the user?
- Acknowledging the work of other programmers
- The main aims of the opensource movement

OpenSource Movement

Open source software is made available for anybody to use or modify, as its source code is made available. Some open-source software is based on a share-alike principle, whereby users are free to pass on the software subject to the stipulation that any enhancements or changes are just as freely available to the public, while other open-source projects may be freely incorporated into any derivative work, open-source or proprietary. Open source software promotes learning and understanding through the dissemination of understanding

Programmers who support the open source movement philosophy contribute to the open source community by voluntarily writing and exchanging programming code for software development. The term “open source” requires that no one can discriminate against a group in not sharing the edited code or hinder others from editing their already-edited work. This approach to software development allows anyone to obtain and modify open source code. These modifications are distributed back to the developers within the open source community of people who are working with the software. In this way, the identities of all individuals participating in code modification are disclosed and the transformation of the code is documented over time. This method makes it difficult to establish ownership of a particular bit of code but is in keeping with the open source movement philosophy. These goals promote the production of "high quality programs" as well as "working cooperatively with other similarly minded people" to improve open source technologies



Advantages and Disadvantages of Each Model

Open Source Advantages	Proprietary Software Advantages
• Source code available, modifiable • Redistribute solutions • Use software in any way • Eliminates single point of failure • Democratic forum for action • No vendor lock-in	• Predictable releases • Entity to hold responsible for bugs, errors, and updates • Consistent feature development • More stable framework • More consistent training options • Easier access to support

Open Source Disadvantages	Proprietary Software Disadvantages
• No guarantee development will continue • Intellectual property (algorithms) • Support consistency	• Higher start-up costs • Single company releasing patches • Vendor owns software

TEAPSYS Open Source vs. Proprietary CMS Software 05/24/2010 8

D4 (Higher Level Only) Advanced Program Development

D4.1 Define the term recursion

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem. Most computer programming languages support recursion by allowing a function to call itself within the program text.

Example:

```
void myMethod(int counter)
{
if(counter == 0)
     return;
else
       {
       System.out.println(""+counter);
       myMethod(counter-1);
       return;
       }
}
```

Note how this function calls itself.

D4.2 Describe the application of recursive algorithms

Recursive algorithms are rarely used in practice. They are difficult for humans to follow in order to understand and maintain code. They should only be used when they provide an elegant solution to a problem.

They might be used to:

1. Print Fibonacci series in Java for a given number
2. Calculate factorial of a give number in Java
3. Calculate power of a give number in java
4. Reverse a String using recursion in Java
5. Print out the contents of a binary tree in order

D4.3 Construct algorithms that use recursion

In the exam you will only have to deal with methods that return a single value and contains only 1 or 2 recursive calls. If you look at the earlier example, think about how it would work if counter starts with a value of 4.

D4.4 Trace Recursive Algorithms

You must show all steps and calls clearly. For example:

```
void myMethod(int counter)
{
if(counter == 0)
     return;
else
       {
       System.out.println(""+counter);
       myMethod(counter-1);
       return;
       }
}
```

Call this 'return a'

Call this 'return b'

Method Call	Output	
myMethod(4)	4	
myMethod(3)	3	
myMethod(2)	2	
myMethod(1)	1	
myMethod(0)		
return a		Returns from myMethod(0) as counter is 0
return b		Returns from myMethod(1)
return b		Returns from myMethod(2)
return b		Returns from myMethod(3)
return b		Returns from myMethod(4)

D4.5 Define the term object reference

A reference to the object in memory, in Java it is shown by the use of the dot notation to access fields and methods e.g.

Person p = new Person();

p.setName("Fred");

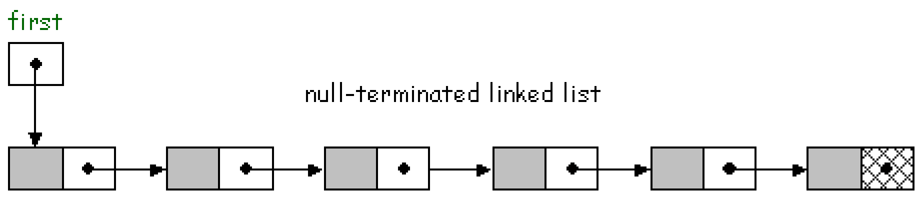
p is the object reference. You cannot call the method without it.

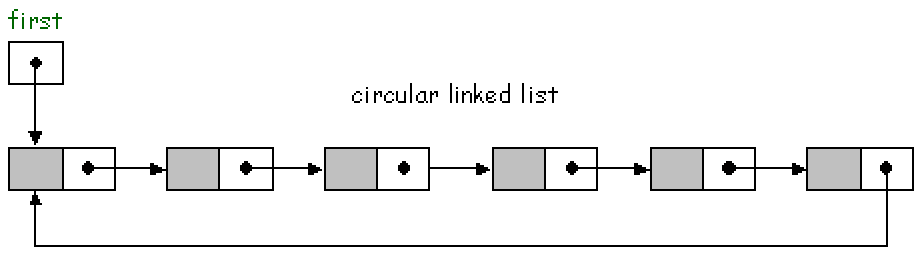
D4.6 Construct algorithms that use reference Mechanisms

In the exam you will be expected to construct java code using this dot notation properly. Just like in your IA.

D4.7 Identify the features of the ADT list

A list is a data structure that can grow in size at runtime as memory is allocated dynamically.





A list is a generic ADT and usually comes with the following basic methods:

add, size, get, isEmpty and remove

Add adds an element to the list, size returns the number of elements in the list, get returns an element from the list but the element remains in the list, remove removes the element from the list for ever. IsEmpty returns true if there are no elements in the list and false if the list contains elements.

Lists can be indexed to allow the program to add and remove elements from specified positions in the list.

D4.8 Describe applications of Lists

Lists can be implemented as **Stacks or Queues**.

Stacks can be visualised liked a stack of trays of plates. The first element added to the stack sits at the bottom. The last one added is the first one to be removed. LIFO(Last In First Out). The add method for a stack is usually called ‘push’ and the remove method is usually called ‘pop’.

Queues act like a queue of people. The elements are added to the end of the queue and removed from the front. FIFO(First In First Out). The add method for a Queue is usually called Enqueue and the remove method is usually called DeQueue.

D4.9 & D4.10 Construct algorithms using a static implementation of a list and object references

In the exam you will be expected to be able to write, explain and trace java code for a linked list written within a class and from the calling program. You need to be familiar with the methods: add(head and tail i.e. to the front and back of the list), insert(in order), delete, list(print), isEmpty, isFull.

D4.11 & D4.12 Construct algorithms using the standard library collections ArrayList and LinkedList

You need to be familiar with the methods available, not how the methods are coded internally. In exam questions, definitions of ArrayList and LinkedList methods will be given where necessary. You need to be able to call them correctly.

D4.13 Explain the advantages of using library collections

See D2.7

D4.14 Outline the features of ADTS stack, queue and binary tree

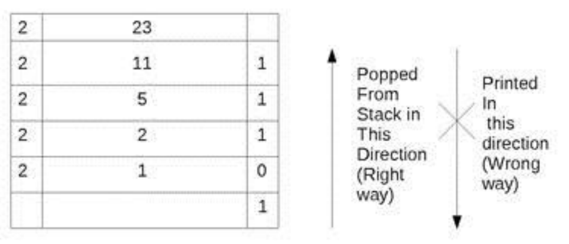
In the exam you will be expected to provide diagrams, applications and descriptions of these ADTs. Look at the code for your IA. ADTs are dynamic data structures i.e. they grow during runtime to accommodate as much data as is necessary. They are very efficient on space (memory usage).

Stacks

They need a pointer that keeps track of the ***head*** of the stack. This is because this is where we want to add and remove elements from.

Uses

Used to Convert a decimal number into a binary number



2	23	
2	11	1
2	5	1
2	2	1
2	1	0
		1

Popped From Stack in This Direction (Right way)
Printed In this direction (Wrong way)

Decimal to binary conversion of 23

The logic for transforming a decimal number into a binary number is as follows:

1. Read a number
2. Iteration (while number is greater than zero)
3. Find out the remainder after dividing the number by 2
4. Print the remainder
5. End the iteration

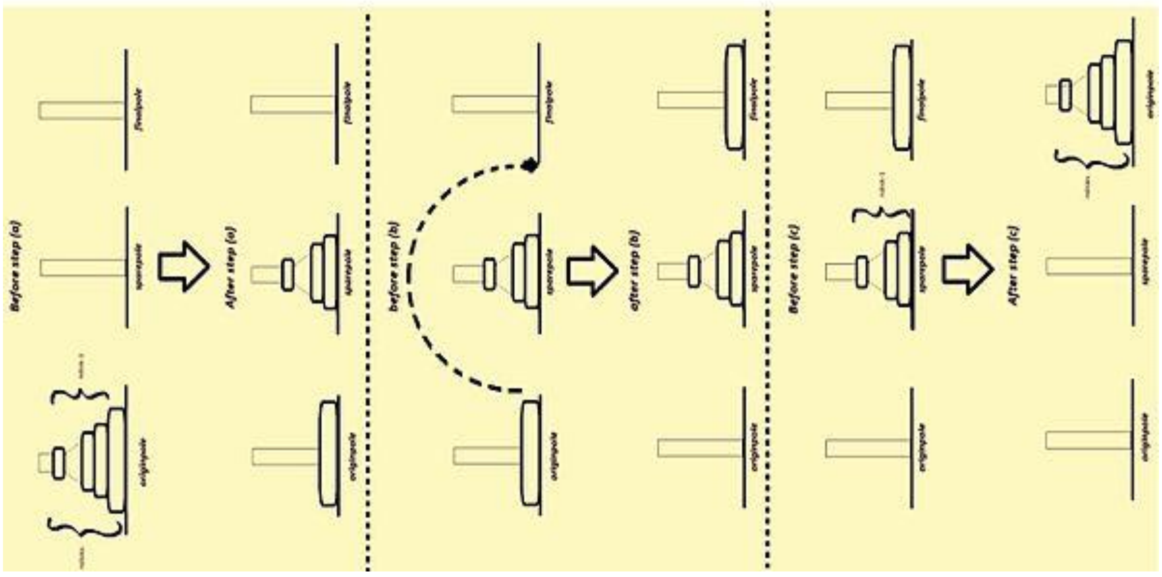
However, there is a problem with this logic. Suppose the number, whose binary form we want to find, is 23. Using this logic, we get the result as 11101, instead of getting 10111.

To solve this problem, we use a stack. We make use of the LIFO property of the stack. Initially we push the binary digit formed into the stack, instead of printing it directly. After the entire number has been converted into the binary form, we pop one digit at a time from the stack and print it. Therefore we get the decimal number converted into its proper binary form.

Algorithm:

```
function outputInBinary(Integer n)
    Stack s = new Stack
    while n > 0 do
        Integer bit = n modulo 2
        s.push(bit)
        if s is full then
            return error
        end if
        n = floor(n / 2)
    end while
    while s is not empty do
        output(s.pop())
    end while
end function
```

Towers of Hanoi



First implementation (using stacks implicitly by recursion) *Not JAVA*

```
#include <stdio.h>

void TowersofHanoi(int n, int a, int b, int c)
{
    if(n > 0)
    {
        TowersofHanoi(n-1, a, c, b);   //recursion
        printf("> Move top disk from tower %d to tower %d.\n", a, c);
        TowersofHanoi(n-1, b, a, c);   //recursion
    }
}
```

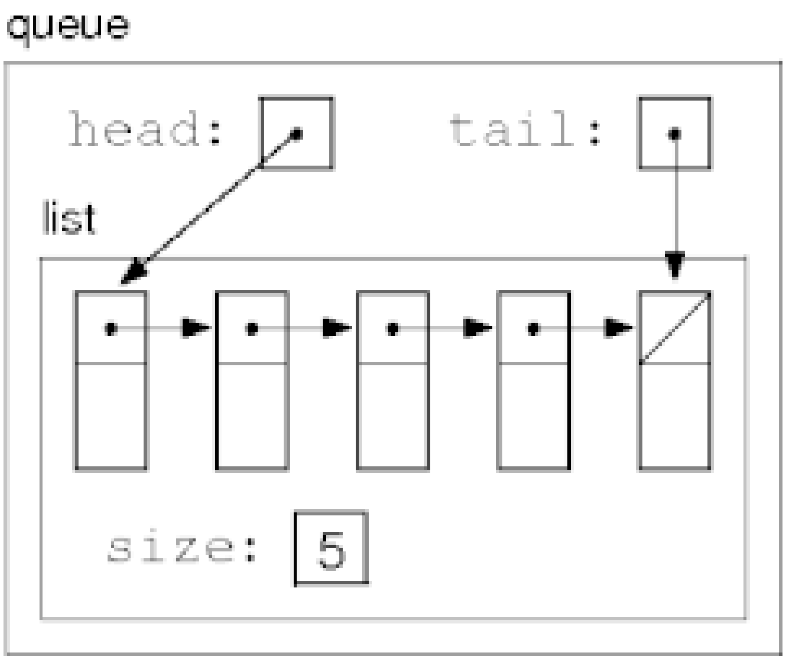
Second implementation (using stacks explicitly) NOT JAVA

```
// Global variable , tower [1:3] are three towers
arrayStack<int> tower[4];

void TowerofHanoi(int n)
{
    // Preprocessor for moveAndShow.
    for (int d = n; d > 0; d--)          //initialize
        tower[1].push(d);                //add disk d to tower 1
    moveAndShow(n, 1, 2, 3);             /*move n disks from tower 1 to
tower 3 using
                                         tower 2 as intermediate tower*/
}

void moveAndShow(int n, int a, int b, int c)
{
    // Move the top n disks from tower a to tower b showing states.
    // Use tower c for intermediate storage.
    if(n > 0)
    {
        moveAndShow(n-1, a, c, b);       //recursion
        int d = tower[a].top();          //move a disc from top of tower
x to top of
        tower[a].pop();                  //tower y
        tower[c].push(d);
        showState();                     //show state of 3 towers
        moveAndShow(n-1, b, a, c);       //recursion
    }
}
```

Stacks are also used by the operating system to keep track of instructions and which to execute next. Additionally, the are used to keep track of events in software use so that the undo button can be used.

Queues

Queues need pointers to keep track of the head and tail of the queue (front and back) as we add to the rear and take from the front.



Usage:

Imagine you have a web-site which serves files to thousands of users. You cannot service all requests, you can only handle say 100 at once. A fair policy would be first-come-first serve: serve 100 at a time in order of arrival. A Queue would definitely be the most appropriate data structure.

Similarly in a multitasking operating system, the CPU cannot run all jobs at once, so jobs must be batched up and then scheduled according to some policy. Again, a queue might be a suitable option in this case.

Say you have a number of documents to be printed at once. Your OS puts all of these docs in a queue and sends them to the printer. The printer takes and prints each document in the order the docs are put in the queue, ie, First In, First Out.

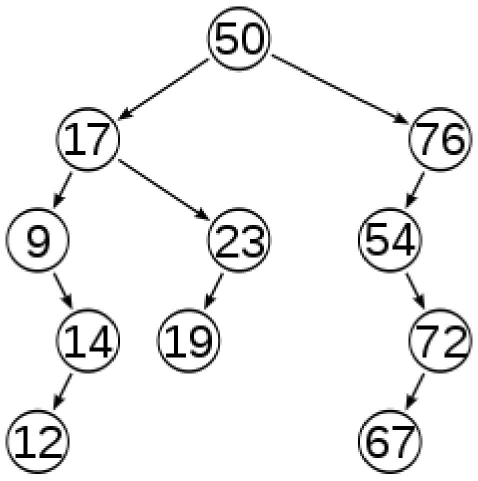
In the situation where there are multiple users or a networked computer system, you probably share a printer with other users. When you request to print a file, your request is added to the print queue. When your request reaches the front of the print queue, your file is printed. This ensures that only one person at a time has access to the printer and that this access is given on a first-come, first-served basis.

Binary Trees

An ordered list of elements where elements of a lower number or letter are stored to the left of the node and elements of a higher number or letter are stored to the right of the node. All binary trees have a **root** node at the top of the tree. All searching starts at this point. Recursion is used to print out the content of the nodes in the tree by considering each set of three nodes as a tiny binary tree and then moving on.

You can traverse a tree pre-order, in-order, post-order see here
http://datastructuresnotes.blogspot.co.uk/2009/02/binary-tree-traversal-preorder-inorder.html

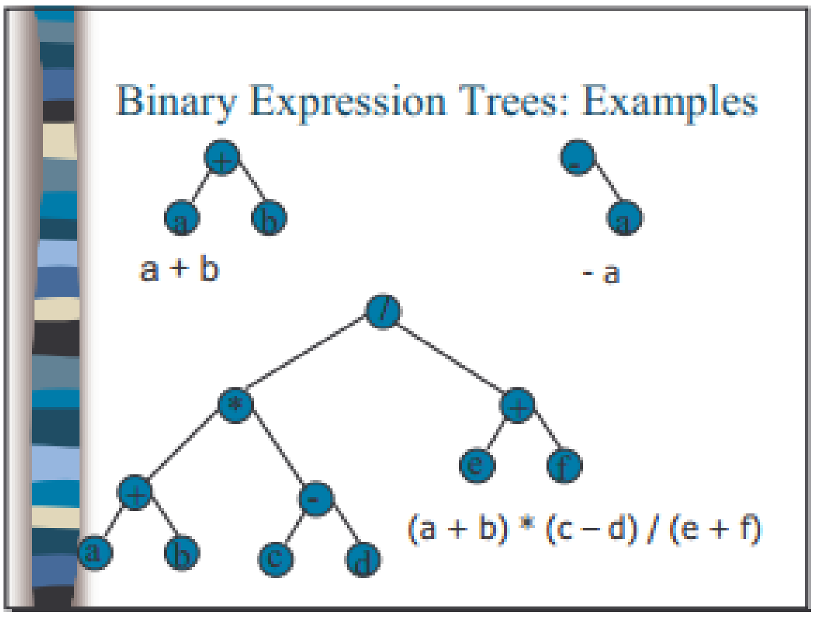
You must understand this for the exam.



Usage:

Binary trees are very useful if you want to find something quickly because they are stored in order. For this reason a binary tree can be used to efficiently store and retrieve unique keys which might index a file for example.

Binary trees can be used to store a mathematical expression before it is evaluated:


Binary Expression Trees: Examples
a + b
- a
(a + b) * (c – d) / (e + f)

See also:
http://www.starteractivity.com/ictlesson/computing/cpt1/stacks%20queues%20and%20trees.pdf

D4.15 Explain the importance of style and naming conventions in code

When working on code, especially in a team it is very important to code in a certain way, to follow coding conventions. These are things like starting class names with a capital letter and objects with a little letter, using meaningful names for the fields and variables you are using, indenting the code, inline annotation etc.

The main point is that code is read more often than it is written. If everyone codes in the same way it becomes more easily read. This cuts down time on maintenance and reduces the chance of error through misinterpretation. This in turn saves time, money and effort.

This point is even more important when working remotely i.e. away from your colleagues. You might be working across the country or across the world. If everyone follows the conventions, code is easy to read and update.