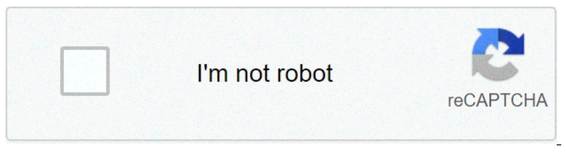

Continue

Introduction to oops in java

EncapsulationThe object-oriented paradigm encourages encapsulation. Encapsulation is used to hide the mechanics of the object, allowing the actual implementation of the object to be hidden, so that we don't need to understand how the object works. All we need to understand is the interface that is provided for us.You can think of this in the case of the Television class, where the functionality of the television is hidden from us, but we are provided with a remote control, or set of controls for interacting with the television, providing a high level of abstraction. So, as in Figure 1.4 there is no requirement to understand how the signal is decoded from the aerial and converted into a picture to be displayed on the screen before you can use the television.There is a sub-set of functionality that the user is allowed to call, termed the interface. In the case of the television, this would be the functionality that we could use through the remote control or buttons on the front of the television.The full implemenation of a class is the sum of the public interface plus the private implementation.Figure 1.4. The Television interface example.Encapsulation is the term used to describe the way that the interface is separated from the implementation. You can think of encapsulation as "data-hiding", allowing certain parts of an object to be visible, while other parts remain hidden. This has advantages for both the user and the programmer.For the user (who could be another programmer):The user need only understand the interface.The user need not understand how the implementation works or was created.For the programmer:The programmer can change the implementation, but need not notify the user.So, providing the programmer does not change the interface in any way, the user will be unaware of any changes, except maybe a minor change in the actual functionality of the application.We can identify a level of 'hiding' of particular methods or states within a class using the public, private and protected keywords:public methods - describe the interface.private methods - describe the implementation.Figure 1.5 shows encapsulation as it relates to the Television class. According to UML notation private methods are denoted with a minus sign and public methods are denoted with a plus sign. The private methods would be methods written that are part of the inner workings of the television, but need not be understood by the user. For example, the user would need to call the powerOn() method but the private displayPicture() method would also be called, but internally as required, not directly by the user. This method is therefore not added to the interface, but hidden internally in the implementation by using the private keyword.Figure 1.5. The Television class example showing encapsulation.InheritanceIf we have several descriptions with some commonality between these descriptions, we can group the descriptions and their commonality using inheritance to provide a compact representation of these descriptions. The object-oriented programming approach allows us to group the commonalities and create classes that can describe their differences from other classes.Humans use this concept in categorising objects and descriptions. For example you may have answered the question - "What is a duck?", with "a bird that swims", or even more accurately, "a bird that swims, with webbed feet, and a bill instead of a beak". So we could say that a Duck is a Bird that swims, so we could describe this as in Figure 1.6. This figure illustrates the inheritance relationship between a Duck and a Bird. In effect we can say that a Duck is a special type of Bird.Figure 1.6. The Duck class showing inheritance.For example: if were to be given an unstructured group of descriptions such as Car, Saloon, Estate, Van, Vehicle, Motorbike and Scooter, and asked to organise these descriptions by their differences. You might say that a Saloon car is a Car but has a long boot, whereas an Estate car is a car with a very large boot. Figure 1.7 shows an example of how we may organise these descriptions using inheritance.Figure 1.7. The grouped set of classes.So we can describe this relationship as a child/parent relationship, where Figure 1.8 illustrates the relationship between a base class and a derived class. A derived class inherits from a base class, so in Figure 1.7 the Car class is a child of the Vehicle class, so Car inherits from Vehicle.Figure 1.8. The Base class and Derived class.One way to determine that you have organised your classes correctly is to check them using the "IS-A" and "IS-A-PART-OF" relationship checks. It is easy to confuse objects within a class and children of classes when you first begin programming with an OOP methodology. So, to check the previous relationship between Car and Vehicle, we can see this in Figure 1.9.Figure 1.9. The IS-A/IS-A-PART-OF relationships and the Vehicle class.The IS-A relationship describes the inheritance in the figure, where we can say, "A Car IS-A Vehicle" and "A SaloonCar IS-A Car", so all relationships are correct. The IS-A-PART-OF relationship describes the composition (or aggregation) of a class. So in the same figure (Figure 1.9) we can say "An Engine IS-A-PART-OF a Vehicle", or "An Engine, Colour and Wheels IS-A-PART-OF a Vehicle". This is the case even though an Engine is also a class! where there could be many different descriptions of an Engine - petrol, diesel, 1.4, 2.0, 16 valve etc.So, using inheritance the programmer can:Inherit a behaviour and add further specialised behaviour - for example a Car IS A Vehicle with the addition of four Wheel objects, Seats etc.Inherit a behaviour and replace it - for example the SaloonCar class will inherit from Car and provide a new "boot" implementation.Cut down on the amount of code that needs to be written and debugged - for example in this case only the differences are detailed, a SaloonCar is essentially identical to the Car, with only the differences requiring description.PolymorphismWhen a class inherits from another class it inherits both the states and methods of that class, so in the case of the Car class inheriting from the Vehicle class the Car class inherits the methods of the Vehicle class, such as engineStart(), gearChange(), lightsOn() etc. The Car class will also inherit the states of the Vehicle class, such as isEngineOn, isLightsOn, numberWheels etc.Polymorphism means "multiple forms". In OOP these multiple forms refer to multiple forms of the same method, where the exact same method name can be used in different classes, or the same method name can be used in the same class with slightly different paramaters. There are two forms of polymorphism, over-riding and over-loading.Over-RidingAs discussed, a derived class inherits its methods from the base class. It may be necessary to redefine an inherited method to provide specific behaviour for a derived class - and so alter the implementation. So, over-riding is the term used to describe the situation where the same method name is called on two different objects and each object responds differently.Over-riding allows different kinds of objects that share a common behaviour to be used in code that only requires that common behaviour.Figure 1.10. The over-ridden draw() method.Consider the previous example of the Vehicle class diagram in Figure 1.7. In this case Car inherits from Vehicle and from this class Car there are further derived classes SaloonCar and EstateCar. If a draw()method is added to the Car class, that is required to draw a picture of a generic vehicle. This method will not adequately draw an estate car, or other child classes. Over-Riding allows us to write a specialised draw() method for the EstateCar class - There is no need to write a new draw() method for the SaloonCar class as the Car class provides a suitable enough draw() method. All we have to do is write a new draw() method in the EstateCar class with the exact same method name. So, Over-Riding allows:A more straightforward API where we can call methods the same name, even thought these methods have slightly different functionality.A better level of abstraction, in that the implementation mechanics remain hidden.Over-LoadingOver-Loading is the second form of polymorphism. The same method name can be used, but the number of parameters or the types of parameters can differ, allowing the correct method to be chosen by the compiler. For example: add (int x, int y) add (String x, String y) are two different methods that have the same name and the same number of parameters. However, when we pass two String objects instead of two int variables then we expect different functionality. When we add two int values we expect an intresult - for example 6 + 7 = 13. However, if we passed two String objects we would expect a result of "6" + "7" = "67". In other words the strings should be concatenated.The number of arguments can also determine which method should be run. For example: channel() channel(int x) will provide different functionality where the first method may simply display the current channel number, but the second method will set the channel number to the number passed.Abstract ClassesAn abstract class is a class that is incomplete, in that it describes a set of operations, but is missing the actual implementation of these operations. Abstract classes:Cannot be instantiated.So, can only be used through inheritance.For example: In the Vehicle class example previously the draw() method may be defined as abstract as it is not really possible to draw a generic vehicle. By doing this we are forcing all derived classes to write a draw() method if they are to be instantiated.As discussed previously, a class is like a set of plans from which you can create objects. In relation to this analogy, an abstract class is like a set of plans with some part of the plans missing. E.g. it could be a car with no engine - you would not be able to make complete car objects without the missing parts of the plan.Figure 1.11. The abstract draw() method in the Vehicle class.Figure 1.11 illustrates this example. The draw() has been written in all of the classes and has some functionality. The draw() in the Vehicle has been tagged as abstract and so this class cannot be instantiated - i.e. we cannot create an object of the Vehicle class, as it is incomplete. In Figure 1.11 the SaloonCar has no draw() method, but it does inherit a draw() method from the parent Car class. Therefore, it is possible to create objects of SaloonCar.If we required we could also tag the draw() method as abstract in a derived class, for example we could also have tagged the draw() as abstract in the Car class. This would mean that you could not create an object of the Car class and would pass on responsibility for implementing the draw() method to its children - see Figure 1.12.Figure 1.12. The abstract draw() method in the Vehicle and Car classes.Object-Oriented Analysis and DesignAs discussed previously, object-oriented programming has been around since the 1990s. Formal design processes when using objects involves many complex stages and are the debate of much research and development.Why use the object-oriented approach?Consider the general cycle that a programmer goes through to solve a programming problem:Formulate the problem - The programmer must completely understand the problem.Analyse the problem - The programmer must find the important concepts of the problem.Design - The programmer must design a solution based on the analysis.Code - Finally the programmer writes the code to implement the design.The Waterfall ModelThe Waterfall Model[1], as illustrated in Figure 1.13, is a linear sequential model that begins with definition and ends with system operation and maintenance. It is the most common software development life cycle model and is particularly useful when specifying overview project plans, as it fits neatly into a Gantt chart format[2].Figure 1.13. The Waterfall ModelThe seven phases in the process as shown in Figure 1.13 are:Requirements Definition: The customer must define the requirements to allow the developer to understand what is required of the software system. If this development is part of a larger system then other development teams must communicate to develop system interfaces.Analysis:The requirements must be analysed to form the initial software system model.Design: The design stage involves the detailed definition of inputs, outputs and processing required of the components of the software system model.Coding: The design is now coded, requiring quality assurance of inspection, unit testing and integration testing.System Tests: Once the coding phase is complete, system tests are performed to locate as many software errors as possible. This is carried out by developer before the software is passed to the client. The client may carry out further tests, or carry out joint tests with the developer.Installation and Conversion: The software system is installed. As part of a larger system, it may be an upgrade; in which case, further testing may be required to ensure that the conversion to the upgrade does not effect the regular corporate activity.Operation and Maintenance: Software operation begins once it is installed on the client site. Maintenance will be required over the life of the software system once it is installed. This maintenance could be repair, to fix a fault identified by the client, adaptive to use the current system features to fulfill new requirements, or perfective to add new features to improve performance and/or functionality.The Waterfall Model is a general model, where in small projects some of the phases can be dropped. In large scale software development projects some of these phases may be split into further phases. At the end of each phase the outcome is evaluated and if it is approved then development can progress to the next phase. If the evaluation is rejected then the last phase must be revisited and in some cases earlier phases may need to be examined. In Figure 1.13 the thicker line shows the likely path if all phases are performing as planned. The thinner lines show a retrace of steps to the same phase or previous phases.The Spiral ModelThe Spiral Model[3] was suggested by Boehm (1988) as a methodology for overseeing large scale software development projects that show high prospects for failure. It is an iterative model that builds in risk analysis and formal client participation into prototype development. This model can be illustrated as in Figure 1.14.Figure 1.14. The Spiral ModelThe spiral, as shown in Figure 1.14 of development is iterative, with each iteration involving planning, risk analysis, engineering (from design, to coding, testing, installation and then release) and customer evaluation (including comments, changes and further requirements). More advanced forms of this model are available for dealing with further communication with the client.The spiral model is particularly suited to large scale software development projects and needs constant review. For smaller projects an agile development model is more suitable.The Object-Oriented Design ModelOne object-oriented methodology is based around the re-use of development modules and components. As such, a new development model is required that takes this re-use into account. The object-oriented model as shown in Figure 1.15 builds integration of existing software modules into the system development. A database of reusable components supplies the components for re-use. The object-oriented model starts with the formulation and analysis of the problem. The design phase is followed by a survey of the component library to see if any of the components can be re-used in the system development. If the component is not available in the library then a new component must be developed, involving formulation, analysis, coding and testing of the module. The new component is added to the library and used to construct the new application.This model aims to reduce costs by integrating existing modules into development. These modules are usually of a higher quality as they have been tested in the field by other clients and should have been debugged. The development time using this model should be lower as there is less code to write.Figure 1.15. The Object-Oriented Design ModelThe object-oriented model should provide advantages over the other models, especially as the library of components that is developed grows over time.An Example Design ProblemTask: If we were given the problem; “Write a program to implement a simple savings account”… The account should allow deposits, withdrawals, interest and fees.Solution: The problem produces many concepts, such as bank account, deposit, withdrawal, balance etc.. that are important to understand. An OO language allows the programmer to bring these concepts right through to the coding step. The savings account may be built with the properties of an account number and balance and with the methods of deposit and withdrawal, in keeping with the concept of the bank account. This allows an almost direct mapping between the design and the coding stages, allowing code that is easy to read and understand (reducing maintenance and development costs).OOP also allows software re-use! … The concept of this savings account should be understood, independent of the rest of the problem. This general savings account will certainly find re-use in some other financial problem.So after discussion with the client, the following formulation could be achieved - Design a banking system that contains both teller and ATM interaction with the rules:The cashiers and ATMs dispense cash.The network is shared by several banks.Each transaction involves an account and documentation.There are different types of bank accounts.There are different kinds of transactions.All banks use the same currency.Foreign currency transactions are permitted.ATMs and tellers require a cash card.Step 1. Identify Possible ClassesATM, cashier, cashier station, software, customer, cash.banking network, bank.transaction, transaction record.account, deposit account, long term savings account, current account.withdrawal, lodgement, cheque.currency.foreign currency, euro cheque.cash card, computer system.Step 2. Remove Vague Classessoftware, computer system, cash.Step 3. Add New classes that arise!Step 4. Create AssociationsBanking Network (includes cashiers and ATMs)Banks (holds accounts)Account (has a balance, a currency, a log of transactions)Transaction (requires a cash card)Lodgement (has an account number, an amount)Withdrawal (has an account number, an amount)Cheque (is a withdrawal, has a payee, an amount)Eurocheque (is a cheque, has a currency)ATMs (accept cashcards, dispense cash)Step 5. Refine the ClassesBank:has a namehas accountshas a base currencyhas a sort codeAccount:has an ownerhas a balancehas an account numberhas a log of transactionsDeposit Account:is an accounthas a shared interest rateCurrent Account:is an accounthas an overdraft limitTransaction:has an accounthas a datehas a valuehas a bankhas an account Numberhas a numberWithdrawal:Lodgement:Cheque:is a withdrawalhas a payeeEuroCheque:is a chequehas a currencyCurrencyConverter:Step 6. Visual Representation of the ClassesFigure 1.16. The Bank class.Figure 1.17. The Account class.Figure 1.18. The Transaction class.OOP AssessmentsSelf-assessments allow you to check your understanding of a topic using multiple choice questions. These self-assessments are corrected on-line and provide explanations for questions that you may have answered incorrectly. These assessments are completely anonymous.Please go to the DCU Loop page for this module at loop.dcu.ie© Dr. Derek Molloy (DCU).[1] Boehm, B. W. (1981) Software Engineering Economics, Ch. 4 Prentice Hall, Upper Saddle River, NJ.Royce, W. W. (1970) "Managing the development of large software systems: concepts and techniques", Proceedings of IEEE WESCON, August 1970.[2] 3] Boehm, B. W. (1988) "A spiral model of software development and enhancement", Computer, 21(5), 61-72.