A Simple and Generic Introduction to Object Oriented Algorithm Design

By
Mike Robey

Table of Contents

ISBN: 978-0646-56956-7

Cover Picture: The picture on the cover has nothing to do with object oriented algorithm design but I couldn't think of a suitable picture and, hey, its a cool picture!

DISCLAIMER: This is a first draft and undoubtedly contains many typographical errors. None of these errors will be significant so apologies for the errors but they should not be an obstacle to your studies.

Preface

The picture of the model plane on the cover of this book may seem unrelated. Here is the connection. Many students enter a computing degree thinking that knowing everything about installing and playing computer games and being able to write small programs in a non commercial programming language will give them a head start. Not having dealt with designing and implementing large scale software written in a commercial programming language, they understandably, don't appreciate the craft, design and sheer scale involved. This leads many students to feel disheartened and to think they have made the wrong career choice. I have spent the last 20 years explaining to students that while their career isn't what they expected it IS the right choice. Most things don't turn out the way we expected but if we soldier on we often find they can actually be more rewarding than we ever imagined. This is absolutely true of software engineering. It isn't what you think it is but take heart! Soldier on because you are about to embark on an exciting and world changing career path.

Having spouted words to that effect for twenty years, I recently had an experience where I found myself in the exact same situation and was faced with following my own advice! I recently decided to take up flying radio controlled model planes. In my life I have held a private pilot's license and also built and flown an ultra light aircraft. This past experience led me to believe that learning to fly model planes was going to be pretty straight forward for me. I bought my little foam model plane, lined up an instructor and off I went. Having arrived at the flying field I eyed off some of the sexy aerobatic models, thinking I would be flying one of them in a week or so. My little foam trainer had enough battery power to stay up for about 5 minutes. Pretty quick you might think. My instructor launched my plane, took it up to a safe height and then handed the transmitter to me. Five minutes felt like an eternity! Five minutes later I was a sweating, shaking wreck! This was NOTHING like I expected it to be. This was REALLY hard! My full size piloting experience was no help at all! My instructor quietly pulled me to one side and told me that it was probably going to take a bit longer for me to learn than most people but if I didn't give up I would master the skills required. His words hit me like physical blows. Their impact was caused not just because things were not turning out as expected, he was using MY WORDS! Having said I would return next Sunday I climbed into my car thinking that hell would freeze over before I came back! However as I drove out of the car park, the word *hypocrite* floated to the surface of my mind. I spent 20 years telling students not to give up just because things are not what they expect and now I am ready to give up under the same circumstances. I made a promise to myself that, come hell or high water, I was going to master the art and craft of flying radio controlled model planes. Once I had, I would then decide if I wanted to continue. It takes your average teenager about 4 to 6 weeks to learn the basics of flying a model plane. It took me 6 months. In that six months, I reduced my poor little foam trainer to scrap foam four times. In the end its airframe was composed almost entirely of the foam packing that

Macintosh computers come in! Finally I had to replace it with a larger, balsa and ply trainer because the extra weight of the glue I was using to put it back together was making it too heavy to fly! Many club members knew to be around when Mike flew. My erratic swoops and dives were considered good entertainment! Not to mention the spectacular crashes as I once again reduced my model plane to scrap foam. I knew this but I never gave up. Six months later, there I was taking off by myself, carefully flying downwind, knowing my instructor had cut the apron strings. I had to do this all by myself for the first time. Turn base, cut the throttle, turn finals, fear and tension building. Flashes of my bright red trainer being turned into scrap balsa if I screwed up the landing zoomed through my mind. I ignored them. Keep the wings level, keep the speed correct, remember throttle controls height, elevator controls speed. Don't get confused. Ground getting close now but don't flare too soon. Damn everyone is watching. Ignore them. Okay flare now! Gently apply elevator. Not too much though! Keep the freaking wings level! Then the most magical thing happened. My little plane flared perfectly, wheels inches above the grass, slowed and settled gently to the ground. Finally it rolled to a stop. I had done it. My plane was now on the ground and I HAD DONE IT! The sense of achievement which was coursing through my brain had me floating six inches off the ground!

It wasn't what I expected. It was way cooler! However I couldn't appreciate that until after I had succeeded. I look back now and I am so glad I didn't give up (my wife possibly doesn't feel the same way because now our house is littered with model planes). The plane on the cover of the this book is the trainer that I achieved that first landing with and that picture reminds me that a *never give and never surrender approach* to life will get you places that intellect and talent alone cannot.

The point is I didn't give up on my dreams and aspirations just because things didn't start out as expected. I soldiered on and was finally rewarded in ways I could not have possibly imagined when I started out.

I suggest you do the same.

One final point is that this text is offered free. I had publishers who were willing to publish and sell this book. Trouble is, I became an academic out of passion to teach and to solve complex problems. I am not in it for the money and so I offer this text to the world free of charge. Enjoy.

Introduction

Why are you reading this book?

Presumably you are attending some form of computing course, most likely a Computer Science or Software Engineering degree. You are reading this book because the prescribed text spends a lot of time talking about where the brackets and semicolons go in various programming languages which hasn't helped you a great deal in terms of understanding the underlying concepts.

The problem is that what you are studying is part technical and part craft. The technical part is much more clear cut and much easier to explain. This is why there are literally millions of software design books which spend all of their time discussing the specifics of programming languages (such as Java, C++, C# or Python). I have written this book in an attempt to focus totally on the concepts and craft involved in software design. You will find nothing in this book which is specific to a particular programming language. You will find that everything in this book applies to all object oriented programming languages. You will find lots of information in this book which is both useful and not found in any first year computing text books. I know this because I spent years searching for a book like this before I gave up and wrote this one.

There are two main areas covered within these pages. The first is the concepts and tools that are used in designing object oriented software. The second, and more important, is the right way of thinking. That might sound unrelated but most of the concepts involved are much easier to understand if you have the right mindset. Most of the people reading this book will not have the right mindset. This is because most of the people reading this book have grown up with technology and hence take it all for granted. For example, if someone handed you a digital watch and asked you to set the time, you would probably start pushing buttons and messing about until you managed to get the time set correctly. Because you have grown up with technology like digital watches, you have developed an instinctive approach to using them. However, if you were then asked to describe the exact steps required to set the time, you probably couldn't. You achieved your goal running on instinct which means all of the logic is buried in your sub-conscious. Hence you cannot explain exactly what you did and how someone might repeat your steps. If we searched the world and found a human being who had never seen a digital watch before then that person would have picked up the small slip of paper that you discarded when you took the watch out of its packaging; the instructions! These instructions would have been very badly worded and quite difficult to understand but this person would have spent the time trying to figure them out before they attempted to play with the watch. Their instincts are of no help to them. They must methodically develop an understanding of the functionality of the watch in order to get it programmed correctly. However that person could explain the steps they followed in programming the watched because they worked it out based on an understanding of the watch and a methodical, problem solving approach. This kind of approach is essential in

software design. Hence familiarity with technology can actually act as a disadvantage!

The one absolute fact in learning Object Oriented Algorithm Design is that, initially anyway, your instincts are your enemy. Not only won't they help you, they will often encourage you in the wrong direction.

The one absolute rule in object oriented algorithm design is NEVER, EVER guess. If you don't know something then you must find a way to understand first (because your instincts are your enemy!).

You will also have to think in a far more logical and detailed manner. In its simplest form, the design for a piece of software consists of a series of instructions which the computer will follow one after the other. These steps are grouped together into what is known as an algorithm. You have been specifying and following algorithms all of your life. You didn't call them algorithms you called them *assembly instructions* or *recipes* and so on. The major difference between the algorithms you have either designed or followed and the ones intended for computers is that a human being is much smarter than a computer and doesn't have to have every little detail specified. A computer processes 1's and 0's and that is pretty much all it can do. This means that computer algorithms must not leave any ambiguity. A good analogy is the difference between making a live action movie and a computer generated animated movie. Live action movies can find and use existing locations, film directors rely on the actors skills and expertise when interpreting their instructions. In a computer animated movie you start with nothing. Every blade of grass, every twig, literally everything about a location has to be designed and input into the animation software. Actors are computer models which mean that every part of their physical appearance and manner of movement has to be input into the animation software. Typically each actor/model will have a team of animators assigned to it and they have to ensure that it sounds and appears as realistic as possible. Don't get me wrong I love watching CG movies (even the ones for kiddies) but every time I do, I marvel at the tremendous amount of extra effort and creativity that went into the movie because it was computer generated. Software is the same. If a software application is easy and reliable to use then congratulations go, not to the computers that execute it, but to the software developers who created it. Their efforts in designing algorithms which, to the user, do incredibly complex tasks, but to the computer get reduced to manipulating 1's and 0's, is to be respected and admired.

Humans have two characteristics which make the design and execution of human centred algorithms much easier. The first is a life time of knowledge. If the algorithm says set microwave to 3 minutes on high, you know what a microwave is and you probably know how to use it so that doesn't need to be explained. The second characteristic is common sense. If the algorithm says put 20 ml of butter in microwave and then cook on high for 50 minutes, you probably know better (or you soon will after spending a few hours wiping up the mess). If the instructions tell you to plug your power drill into the electricity

and then drop it into a bucket of water then you know better than to follow those instructions. In other words the steps in the algorithms assume you won't do anything that seems silly or dangerous and that you already know how to do lots of things. Computers do not have either of these characteristics. Software developers have tried very hard to provide the illusion that the computer knows what it is doing. As you progress through your course you will realise that to achieve this illusion requires a tremendous effort.

This means that, as you attempt to design algorithms which are intended for machines, you will have to make no assumptions about what the computer knows (because it knows nothing) and how much common sense it has (because it has none). Remember that humans can also lack these characteristics from time to time. My father ran a landscaping business for over 30 years. Many of his employees were not exactly the sharpest tools in the shed so to speak. One day when things were not going well, one of his workers asked him what he should do with sand from the hole he had just dug. My father, who was hot, frustrated and angry yelled "Dig a bloody hole and bury it!". The worker, quite sensibly, retreated. A few hours later the worker returned to my father and asked what he should do with the sand from the second hole (the one he dug so he could have a hole to put the sand from the first hole in). While from a business perspective, my father had just lost a few hours of valuable employee time. From a human perspective, he could not help but laugh which in turn dispelled his foul mood. The point is that his worker simply followed his instructions without question. In other words his employee acted like a computer! Most of the time humans do not do this. In fact, the most valuable employees are usually the ones who can interpret their instructions and make allowances for things that their employer may not have thought of. How much ability does a computer have in this area? None at all. If I asked you to add two numbers together, you would calculate the result and tell me. I would not need to explicitly ask you to tell me the answer. Common sense told you that if I asked you to add the numbers together then I want you to tell me the result because otherwise my request is pointless. If you designed and implemented an algorithm to do the same thing on a computer and you did not explicitly state the the sum of the two numbers needs to be displayed to the user then the computer would have added the numbers together, placed the result in memory and ended the program without having displayed the result. How much common sense and initiative does a computer have? None at all.

Algorithm design is a creative process and hence imagination and artistic ability is required. You will also learn to be a brilliant detective. Designing and testing algorithms and the resulting software is not a perfect science. If a company informs you that their software has been *completely* tested and works *perfectly* then they are either incompetent, lying or, most likely, both!

You will need to develop good diagnostic skills in order to track down where the errors in your algorithm are. Notice I said errors I did not use the term bugs. The term bug comes from an incident where a moth flew between two

circuit boards and in doing so, formed a physical connection between the two boards and shorted them out. In other words the problem was literally caused by an insect (i.e. bug). The cause of the malfunction was a random event that the circuit board designers had no control over. If your algorithm does not work it will be because you have either left something out or something is incorrect. In other words you made some *mistakes*. Notice the use of plurals here. That is because if you have a problem then the cause is unlikely to be a single error (although that does sometimes happen) but a collection of errors. I have always assumed that the term bug was carried over to software simply as a term which implied a problem with the computer. It is not an accurate term because it implies the errors were random and caused externally. That won't be the case. If there is a mistake it will be because you made it. The good news is, that also means that you can find it and correct it.

The final difficulty with software is that it is constructed by typing words into a series of files and then converting those files into binary instructions that the computer can directly execute. In that sense your design is abstract. It only exists a a collection of words. You cannot pick it up, turn it around, shine a light on it or try to physically fit pieces of it together. In that sense it is like a mathematical proof. It is an abstraction. However when it runs it physically manifests itself in the real world. It can display graphics, make sound or control physical devices. It can turn your fridge on or off, start your washing machine and hence possibly be responsible for burning your house to the ground. In other words, a software design task involves dealing with the problem in an abstract form but having to allow for many of the complications of the real world.

Having depressed you completely about how difficult all this is going to be let me cheer you up. Firstly, if it wasn't difficult then a university degree would not be required and your whole choice of career would not exist! Secondly as you work your way through this book I will attempt to provide you with the help and understanding you need in order to develop the skills required to be a top notch algorithm designer. Let us also be clear on the fact that, given a well defined algorithm you can train a monkey to turn it into Java, C#, C++ or whatever programming language you like. Algorithm designers are called Software Engineers and are paid high salaries because they possess high level design skills. The same companies pay much smaller salaries to the people who turn the design into code. Code cutters are sometimes referred to as *code monkeys*. i.e. a code monkey doesn't have to be very bright and are paid peanuts. If you wish to be a code monkey then stop reading now. If you want a successful career in software design then read on!

As far as I am concerned anyway, software design is an art and like all art there is a style, elegance and beauty to a well crafted algorithm that can only be appreciated by another software developer. In the same way that, if a World War Two Spitfire fighter roared over your head, you would probably be frightened and annoyed by the sound of its engine. A flyer however would regard the throaty growl of a Rolls Royce Merlin engine as pure song of the highest quality. In other words, beauty is the eye of the beholder and there is

beauty in software design and it can be a thrill to experience and appreciate it.

One last point is that software developers change the world and it can be one heck of a high to know you might have been responsible for that. I met one developer whose company had developed the software system that a leading Australian supermarket chain uses on its checkouts. Not exactly an exciting arena but he told me that every time he walks past a store belonging to the supermarket concerned he is totally wowed with the knowledge that all these people are using HIS software. All those people and he did that!

You will need to appreciate that algorithm design isn't achieved by bashing away in a particular programming language. It requires a deep understanding of the tools of the trade and the ability to apply these tools in an elegant and artistic manner. You will need to gain a lot of skill and knowledge to do this.

On the other hand, with the right attitude and a *"Never give up never surrender approach"*, almost anyone can learn to do this well. Just remember never guess and never give up the search for understanding and you will become a first class software developer. Having become a first class software developer, you will go onto to develop software which will, in one form or another, change the world. How many people can say that!

Finally, though not directly related to Computer Science or Software Engineering, if you have not read the works of Douglas Adams please do so. That might seem a bit flippant and frivolous until you understand that he wrote an entire series of books centred around the concept that the answer to a question is only useful when you understand the question itself. This will be the case for many of you in your attempts to understand this book.

Good luck and as the delightful Douglas Adams says "Don't Panic".

Chapter One

"They said I was daft to build a castle in a swamp but I did it anyway just to show them! It sank into the swamp. So, I built a second one. That sank into the swamp and so I built a third castle. That one burnt down, fell over and then sank into the swamp. But the fourth one stayed up!"
Monty Python and The Holy Grail, 1974

Problem Solving

Introduction

An algorithm is a series of steps, which when followed, perform some task or achieve some goal. Before you delve into the mysteries of designing algorithms for computers, you need to first hone your skills in the area of problem solving and logical thinking. This chapter will attempt to describe the related concepts and techniques and illustrate both with examples. In the introduction I mentioned that artistic/creative talent is required. When I use the word art I don't mean paintings and statues. I mean the result of using your creative skills to methodically work towards a viable solution. The answers won't be obvious and imagination and adherence to a methodical approach will be vital to a successful outcome. Modern technology has embedded in all of us the need for instant gratification. We are no longer happy to wait for what we require, we want it now and often technology can provide what we ask for immediately. Coming up with the steps required to achieve some task does not come instantly. Instant gratification is a negative influence because it drives us to try to find an instant solution and often the solution is the result of days of painstaking thought. Having said that, almost anyone can become adept at problem solving by developing the stamina and discipline required to methodically work through a problem to its solution.

Okay, so I have a problem to solve. Where do I start?

Firstly let me clarify what I mean when I speak of problems and solutions. Problems, in this case are tasks to be performed, results to be achieved. Solutions will always be procedural instructions which result in the task being performed or the result being achieved. The complete solution to anything but the most trivial of problems is rarely obvious. However there is usually some facet or hint which can be used as a starting point. From this starting point you must slowly and carefully evolve a solution. This process is part science and part creative (i.e. art). It is science in the sense that you can employ logic and

reasoning to deduce facts or parts of the total solution. The intuitive leaps that you will have take to fill the gaps that science leaves behind is what I am calling art. Once a leap is made, science swings in to verify whether the artistic step forward is valid or not.

Here are my golden rules of thumb for problem solving:
1. Never guess.
2. Start by identifying the sub problems and then solving them.
3. Every step or decision should be justified.
4. Always be prepared to throw away some part (or all) of the solution provided the decision to do so can be justified with good sound reasons.
5. Resist the temptation to leap to the final conclusion.
6. Always work methodically breaking one big problem into a set of smaller ones.

Let us go over each of these rules in more detail.

Never Guess

This is the most important rule. Guesses either involve making wild guesses about what the next step should be or equally wild guesses about some facet of the problem to be solved. Both will lead you in the wrong direction. If there is some facet of the problem which you do not understand then you must put your problem solving on stand-by and spend time understanding whatever it is that is required. For example, if a problem required understanding what a prime number is and you were not sure then break out the mathematics books and learn about prime numbers before continuing, otherwise your solution will inevitably be flawed. A wild guess about the next step of the solution is a gamble. The odds of your guess being correct are slim. A few guesses down the track and you are guaranteed never to go anywhere near a valid solution. Note that this is not the same as making a small intuitive leap based upon some kind of logic and then verifying your leap. The first is coming up with a step randomly (or by being mislead by our instincts). An intuitive leap is much more likely correct if it is small and based upon logic. Imagine you are lost in the country side. A wild guess means you simply pick a direction at random and start walking. An intuitive leap might involve observing the position of the sun, and striking out in the general direction that you want to go in relation to roughly estimating north, south, east and west by the position of the sun and the rough time of day. The first method has you wandering forever around the country side while the second assumes that if you proceed in a known direction (roughly), you are going to hit civilisation sooner or later!

Start by Identifying the sub problems

The solution to a problem of any complexity can always be decomposed into solving a set of sub problems which, when combined together, solve the overall problem. Too much detail always confuses us. Identifying sub problems and then solving each independently reduces the amount of complexity which has to be dealt with at any given time. We do this all the

time when trying to do things in the real world. Consider the construction of a garden shed from a kit. The sub problems would be:

1. Read and attempt to understand the instructions.
2. Layout and identify the parts.
3. Identify the main components of the shed (i.e. walls, roof, door) and sort the parts into groups (i.e. wall parts, door parts, roof parts).
4. Identify and obtain the tools required.
5. Assemble each component.
6. Finish the shed by fitting the components together.

On the surface we might think that the above steps are the solution to our problem. However, anyone who has attempted such a task will tell you that it is not that simple and that each of the above steps require further consideration. The difficulty is that mostly we do this problem sub-division instinctively and in devising solutions for computers our instincts will not be of any use to us and so we need to learn to break our problems down by using logic and reasoning.

Every step or decision should be justified.

Every time you choose a course of action or decide on the next piece of the puzzle then you need to be able to justify why that is so. I don't mean in a formal scientifically provable way but there must have been a reason. This is important as you are proving to yourself that you are not making a guess but have a reason for proposing this step. Get used to putting your reasons down on paper. Quite often the attempt to justify will fail and lead you to the conclusion that your step is wrong. Don't be disheartened! Firstly, this is normal, our intuition works at a sub conscious level so we often feel that we are correct but if we cannot consciously justify it then we have to accept that our intuition has sent us in the wrong direction. Secondly, if you cannot justify that the proposed step is correct, can you justify that it is not correct? If so you have understood the problem at hand a little more than you did previously. This is an important point. The knowledge as to why a decision is wrong is still progress because you now realise something about the problem at hand that you didn't previously know or understand.

Always be prepared to throw away some part of the solution.

One of the most common traps that students fall into when solving problems is that once they have a piece of the puzzle, they will not consider the possibility of discarding it. In other words they are unwilling to back track and try again. This is understandable. No matter how thrilling (or not) a person may find problem solving, it is very hard work. That leaves one with the tendency to not want to give up the result of hours of hard work. Never the less, if the problem is to be solved and it is becoming obvious that you are barking up the wrong tree then it is better to cut your losses sooner than later. Of course, the previous rule means that your decision to take a step back must be justified. Accept that this is likely to happen to you many times in your attempts to solve a problem. Further accept that this is a NORMAL part

of problem solving and happens to everyone, every time they attempt to solve a problem. Being in denial will only waste hours of precious time. Also remember that discarding some part of your solution means that you have understood something which you were previously unaware of so the effort has not been wasted. This also happens when solving real world tasks, its just that in those cases, the situation will reach a point where backtracking is not an option but an absolute necessity. Going back to the garden shed example, if we had used a mounting bracket, intended for the roof, on the wall, we would reach the point where we had a wall bracket spare and no roof bracket. At that point, our intrepid garden shed constructor would have no choice but to disassemble the wall to the point where the brackets can be swapped.

By the way, can you guess what I was struggling with around the time of writing this chapter? There is no evil in this world which surpasses the evil master minds who intentionally make garden shed assembly instructions both ambiguous and mysterious.

Resist the temptation to leap to the final conclusion.

As the solution to a problem unfolds, the temptation to simply guess the rest of it becomes harder and harder to resist. This is driven by our need for instant gratification. You must resist it at all costs. Remember that as you solve the problem you are also building up a better and better understanding of the problem and anything related to it. If you cease building that knowledge by simply attempting to guess the rest of the solution you will go horribly wrong. Guessing will always lead to disaster, so don't. Imagine attempting to cross a mine field using some kind of mine detector. The experience will be terrifying and the progress slow. As you get closer to the other side, the temptation to simply run the last little bit will be strong. If you do, you will probably die. If you resist the temptation and keep up the methodical search for a mine free path then you will not be blown into tiny little pieces.

Always work by breaking one big problem into a set of smaller ones.

As previously stated we start by dividing the problem into a series of smaller, sub problems. The problem sub division process should not stop there but should be applied each time there is too much complexity to deal with. It is always easier to solve a series of smaller problems one at a time than one large, complex problem all at once. If you can see a way of dividing the problem up then do so. In very complex problems, identifying, isolating and solving one small sub problem is often the only way a total solution will be found. No matter how brilliant we are, each of us can only deal with a small amount of complexity at a time. In reducing the solution of a complex problem down to finding the solution of a series of smaller, less complex problems, we reduce the complexity that has to be dealt with at any given time, to a manageable level. This is where a methodical approach comes in. The first attempt at solving any problem should aim only to identify the sub problems. No attempt should be made at this point to come up with detailed steps for the solution of each sub problem. That will come later as each sub problem is

solved independently of the others. We do this all the time in the real world. For example, nobody writes an essay in one draft, starting from the first word of the title and ending with the last word in the conclusion. We start by identifying the main sections of the essay (i.e. identifying the sub problems). The next step involves producing a rough draft for each section, followed by a continual refinement until we decide we are happy with the result. This same process is used in problem solving.

At this point one might think that problem solving seems like a very simple and straight forward process. Like many things in life of course, its not that simple. There is a lot of subjectivity to this process. For example, when is a problem small enough that it does not need to be further sub divided? How do I identify the sub problems? How do I make sure that when the solutions to the sub problems are merged, the overall solution has not left something out? A good analogy to this would be flying an aircraft. Just knowing how the controls work does not mean that a person could leap into an aircraft and successfully fly it. In fact learning to fly requires a huge amount of practice. The problem solving skills that you need to develop will also take much practice. The good news is that, with practice they will come. This is not a skill that requires great intellect. Practising a methodical and disciplined approach to problem solving will always yield good results.

Why are my instincts always wrong?

The problem with problem solving in the context of algorithm design is that your initial instincts will, at first, always tend be wrong. You will break all of the rules described above a number of times over the next few months of your course. The main reasons for this are:

1. You are not used to dealing with the amount of detail that you will be required to.
2. Your instincts either do not apply to modern technology or will mislead you because of the way you have been using technology.
3. Machines don't work like humans, they can achieve similar goals but in a completely different manner.

Humans and animals are magnificent in what they can do and machines are crude by comparison. For example an aeroplane does not flap its wings to fly. Therefore an Ornithologist (bird expert) is not much help in either the design and construction of an aircraft nor in trying to fly it. Not only do machines function in a different way, they can also upset and confuse our own biological systems. Evolution is an extremely slow process and technology has not existed in the world for very long. Technology is also changing at a fast pace. All of these factors mean that we cannot rely on our instincts or feelings when developing solutions to our problems. This has been an issue for as long as there has been technology. For example, the torque of an aircraft engine and the motion of an aircraft through the air upsets our sense of balance so that when the aircraft is flying straight and level, the people inside feel like that are climbing and turning to the left. When a pilot has a clear view of the horizon this is not an issue because his vision system is providing extra information

which assures him that the aircraft is straight and level. However, if the aircraft enters cloud and the horizon was no longer visible, the balance system then takes over as the major source of sensory input. There are instruments which indicate whether or not the aircraft is straight and level but reading the instruments does not have the same impact as a clear view of the horizon. Without the proper training, a pilot flying in cloud, will adjust the attitude of the aircraft so that his balance system feels right. In other words he will lower the nose and start a turn to the right. In aviation circles this is known as the graveyard spiral because the ultimate conclusion is that the aircraft spirals down and crashes into the ground. Student pilots spend a lot of time learning to ignore their balance system and believe what the instruments tell them before they can successfully fly an aircraft in cloud.

We must also learn to ignore our instincts and to develop a methodical and disciplined approach to problem solving. It can be done but it requires much practice and the belief that such an approach will ultimately lead to the desired solution. I will keep telling you that it will but, at some point, you have to start believing me where your instincts will be telling you not to.

Breaking The Problem Down

One of the most difficult skills to learn is the ability to break a large problem down into a series of smaller ones. It is always possible to sub divide one large problem into a set of smaller ones. However, the ability to do that is not an instinctive one. Your attempts in this area are your first step towards developing the right mindset. A simple example would be a large jigsaw puzzle. Lets suppose that this jigsaw puzzle has thousands of pieces. Worse still, the top half of the picture consists of blue sky with a few small clouds and the bottom half mostly ocean. If we approach the solution to the problem at its simplest level we might say the large problem of assembling the jigsaw can be broken down into a set of smaller problems; fitting a few pieces together. Ah ha! we have managed to break the problem down. After a while we see that our solution has accomplished very little progress because there are so many choices of combinations of pieces to fit together. At this rate we will probably die of old age before the jigsaw is complete. We need to abandon our algorithm and think again. The first step is to analyse why our current solution does not appear to work. The problem is there are too many pieces to choose from. What is required is some way of reducing the number of pieces under consideration at any given point. One way of doing that is to divide the pieces up into categories. Examination of the pieces reveals a number of categories:

- edge
- corner
- sky
- cloud
- ocean
- other

After the pieces have been sorted into their categories we realise that each

corner piece can be placed in its correct position. The edge pieces can be further categorised into sky edge, cloud edge and ocean edge. Finally, for each category we go back to our original plan which is simply trying to find pieces that fit. Finishing the jigsaw puzzle is still difficult and time consuming but we have significantly reduced the number of pieces available to choose from at any given time. The point is we have transformed an unsolvable problem into a difficult but solvable one. Along the way we had to revise our thinking in order to achieve a solution. However notice that our final solution was based upon our knowledge of why the initial solution wasn't going to work. In other words our initial effort wasn't wasted because it still ultimately led us to a viable solution.

Designing an Algorithm for a Moron

As has been previously stated, algorithms intended for computers, cannot make any assumptions about prior knowledge or common sense. The basic problem is that anything not clearly specified in the algorithm will not happen.

Context is Everything!

A human is very good at interpreting instructions in the correct context. Computers are not. This means that the algorithm has to clarify any contextual issues as well as clearly define the steps.

For example, let us consider numerical precision. Without being told a human will automatically adjust the precision of any measurements or calculation to a suitable level. A carpet layer is going to measure room dimensions and cut the carpet to the nearest cm. A shop assistant is going to work to two decimal places when calculating change. Your lawnmower man is going to calculate lawn area to the nearest metre. If all these people used software to make their measurements then the algorithms in the software would have to explicitly allow for the required precision. Otherwise the computer would use the maximum precision available resulting in carpet and lawn dimensions being specified to a millionth of a millimetre and change calculations specified to a couple of thousand decimal places. Many problems easily solved by humans are impossible to solve with computers. Ever wondered why we all don't have our very own C3PO to do our housework? Because just getting a robot to imitate basic human functionality such as walking, recognising objects, picking up objects etc. is either only achievable in constrained situations or impossible. The reason these things are so difficult is because, everything must be precisely detailed whereas humans do not require that. Humans use context and their own experience to fill in the details. For example, the carpet layer knows how accurate he/she has to be from experience. The lawn mower man knows that he only needs a rough area calculation as a means to estimating how long he will take to cut the lawn. The shop assistant knows that change can only be given to two decimal places. None of these people need to be told this in order to do their jobs. However an algorithm on a computer knows nothing and the concept of context can not be represented in software. Hence all this must be specified in the algorithm.

How to go about solving problems

The types of problems that algorithm designers have to solve are usually very structured in that they do lend themselves to being broken down into sub problems. Unfortunately most students will ignore this and simply leap in and try to solve the whole problem all at once. The reason they do is that their drive for instant gratification is urging them to get on with it and emotionally, a student gets satisfaction from producing something, anything. The fact that hours later everything so far produced will most likely be scrapped is completely overridden by the urge to get started on a solution. In other words, the student's every instinct will be guiding them away from the correct approach to achieving their goal.

When I was an undergraduate student, I studied a course on a particularly verbose and difficult to use programming language known as COBOL. The lecturer for this unit told us all that we should spend 80% of our time designing and only 20% actually writing COBOL. I decided to put this idea to the test on the assignment for the course. We had four weeks to complete it. I allowed myself 3 weeks to work up a clear and precise problem statement, design an algorithm and test it by hand. That left me 1 week to convert my algorithm into COBOL and submit the result. Every other student ignored the lecturer's advice and headed for the labs. Every fibre of my being was screaming at me to do the same. After all, how could I alone be right and all of them be wrong? I spent one day on the problem statement. Being an assignment, the requirements were pretty clear anyway but I still needed to be sure I knew what had to be done. Note that this also involved a small amount of research related to understanding different facets of the problem that I needed to clarify. The next 3 weeks were spent developing and testing my algorithm. The entire time, a little voice in my head was screaming at me to stop wasting time and head for the labs. I resisted the voice but it took a lot of discipline to do so. Finally after many iterations I had an algorithm which I had tested by hand to the point where I knew it would perform as required. I gathered up my bits of paper, wandered into the nearest lab and started converting my algorithm into the COBOL programming language. It took me one morning to get the COBOL entered and to the point where my program would execute without any errors. It was at this point I realised two very significant things:

1. The lecturer was correct. Putting the thought into solving the problem properly and ignoring my instincts had yielded to required solution.
2. My life was in danger because every other student in the course was out to kill me because I was the only student who managed to submit a working assignment. They all wanted an extension but I was living proof that they had been given plenty of time.

A good place to start with algorithm design is to try to design algorithms designed to be executed by humans rather than computers. Even though you have some experience in this regard, you will, be surprised at how difficult this can be sometimes. Consider a recipe which is to be used by someone who does not have much experience or ability at cooking. Phrases like *add salt to*

taste, *cook until brown*, or my personal favourite, *cook until done* cannot be used because the person using the recipe might not understand those phrases (I know I don't!). When designing an algorithm, even one for a human, you cannot assume specialist knowledge. What is obvious to you may not be to them. As an example, I had a friend who was attempting to install software on his computer which would allow him to connect to an internet service provider. He phoned me and asked me for help. I spent quite a while instructing him to try various things and reporting back the result but nothing seemed to work. The problem was that he could not get his computer to read the CD supplied to him by the service provider (yes I am that old, in those days DVD's did not exist and Blu-ray was not even a figment of somebody's imagination). I had reached the conclusion that either the CD that he was attempting to install the software from was corrupt or his CD drive was not functioning correctly. I asked him to bring the CD to my house and as I placed the CD into the CD drive tray of my computer, my friend exclaimed *"Oh, they go that way up do they?"*. You might be tempted to laugh or reach some, not very nice, conclusions about my friend's intelligence but you would be wrong. He is not a stupid person, he was very successful in his business ventures but coming from an older generation, he had simply never used a CD before in his life and you have to admit they do fit nicely either way up. He reasoned that seeing that you feed fax pages into a fax machine face down then the same should be true of CD's. . He wasn't taking a wild guess he was making his decision based upon his experience. Unfortunately his experience never involved CD's. In the same way, if we design an algorithm which doesn't clearly state how each step is to be done then it is unlikely to be a viable solution.

The Algorithm for Designing Algorithms

If we consider the art of problem solving/ algorithm design as a problem in itself then it is possible to design an algorithm for solving algorithms! We already have the basic three steps already:

1. Analyse the problem and form a problem statement.
2. Break the problem down in to sub problems.
3. Solve each sub problem.

The above algorithm needs more refinement. The steps need to be repeated for each sub problem until a solution is found. Also there needs to be some evaluation of each sub problem solution.

```
1. Develop a good understanding of the requirements of the problem.
2. Break the problem down into sub problems.
3. For each sub problem:
      3.1. Develop a good understanding of the requirements of
           the sub problem.
      3.2. If the sub problem is small enough to be solved Then
               3.2.1. Develop the steps to solve it.
               3.2.2. Test and evaluate your solution.
           Otherwise
               3.2.3. Break the sub problem down into smaller sub
                      problems and repeat the process.
4. Test and evaluate your solution.
```

The algorithm more or less assumes that step 4 will yield positive results. This may not be the case:

```
1. While problem is not solved:
1.1    Develop (or further refine) the requirements of the problem.
1.2.   Break the problem down into sub problems.
1.3.   For each sub problem:
          1.3.1. Develop a good understanding of the requirements of
                 the sub problem.
          1.3.2. If the sub problem is small enough to be solved Then
                    1.3.2.1. Develop the steps to solve it.
                    1.3.2.2. Test and evaluate your solution.
                 Otherwise
                    1.3.2.3. Break the sub problem down into smaller
                             sub problems and repeat the process.
1.4.   Test and evaluate your solution.
1.5   IF tests and/or evaluation reveal errors or that the
         requirements are not being met properly Then
           1.5.1. Identify the issues related to the failure(s)
                  and use them to guide the next iteration.
```

With a lot of practice the above algorithm will work. It incorporates all of the significant issues:

1. Forming and refining the problem statement.
2. Repeated refinement of the solution.
3. Reasoning as to why something does not work and applying the results of that reasoning into a new solution. Human beings are very good at hindsight, looking back at a situation and making observations about it. We are pretty poor at foresight so our algorithm must make heavy use of hindsight if it is to be a practical solution.
4. Testing your solution to be as sure as possible that it works correctly.

An Example, Arming and Disarming a Burglar Alarm

Suppose you had a burglar alarm in your house and were off on holidays. You have arranged for someone to come over every few days and feed the fish, water the plants etc. This person has never used a burglar alarm before and so needs an algorithm for both arming and disarming the alarm which assumes nothing about the way in which burglar alarms operate. As the complexity for disarming and arming the alarm is the same and as disarming has to happen first, then its the best place to start. Note that I have justified where I am starting. First we need to outline our problem statement before attempting an algorithm. A naive problem statement might be simply:

Problem: Disarm the alarm.

That is not very informative so a little more detail is required:

Problem: User must enter house, find the alarm control panel and use it to disarm the alarm.

Note that this problem statement not only makes the main steps required

obvious but also implies other requirements such as explaining where the alarm control panel is. The first step is to identify the sub problems:

```
1. Get to the control panel.
2. Use the control panel to turn off the alarm system.
```

The next step is to develop an understanding of each sub problem:

- The control panel is located in the kitchen, next to a door which opens onto the garage.
- The alarm is turned off by entering a four digit code.

After a little thought the following algorithm is produced:

```
1. Enter the house and go to the control panel.
1.1.   Enter garage.
1.2.   Open kitchen door and enter kitchen.
1.3.   Control panel is on the wall, to the right of the kitchen
       door. Turn and face it.
2. Disarm alarm system:
2.1.   On the control panel is a set of 10 buttons labelled with the
       digits 0-9 and another key marked Enter.
2.2.   Press the following buttons in order: 4, 6, 2, 8, enter.
2.3.   The alarm should beep once and the LCD display on the alarm
       should state "Dis-Armed". If you pressed the wrong buttons
       then the alarm will beep three times and you need to re-enter
       the numbers correctly. If you do not do this quickly enough
       the alarm siren will sound but you should keep going until
       you have entered the numbers correctly.. This last step
       cannot be stressed enough as my neighbours are large, violent
       and do not like loud noises.
```

Notice two things:

- Some of the information is simply explaining details about where things are or what will happen. We will have to do this differently when it comes to algorithms for computers.
- Step 1.1 and 1.2 haven't explained how to open the garage and kitchen doors. With this in mind we refine the algorithm further.

```
1. Enter the house and go to the control panel.
1.1.   Enter garage:
1.1.1.      Face garage door and press the button on the white remote
            control.
1.1.2.      Wait for garage door to open and then walk into the
            garage and face the kitchen door.
1.2.   Open kitchen door and enter kitchen:
1.2.1.      Use the bronze key on the key ring to unlock kitchen
            door.
1.2.2.      Open door and enter kitchen.
1.3.   Control panel is on the wall, to the right of the kitchen
       door. Turn and face it.
2. Disarm alarm system:
2.1.   On the control panel is a set of 10 buttons labelled with the
       digits 0-9 and another key marked Enter.
2.2.   Press the following buttons in order: 4, 6, 2, 8, enter.
2.3.   The alarm should beep once and the LCD display on the alarm
       should state "Dis-Armed". If you pressed the wrong buttons
       then the alarm will beep three times and you need to re-enter
       the numbers correctly. If you do not do this quickly enough
       the alarm siren will sound but you should keep going until
       you have entered the numbers correctly.. This last step
       cannot be stressed enough as my neighbours are large, violent
       and do not like loud noises.
```

Further evaluation reveals that step 2.3 is a little wordy and as a misunderstanding of the instructions could result in loud noises and angry neighbours, it needs further refinement:

```
1. Enter the house and go to the control panel.
1.1.   Enter garage:
1.1.1.      Face garage door and press the button on the white remote
            control.
1.1.2.      Wait for garage door to open and then walk into the
            garage and face the kitchen door.
1.2.   Open kitchen door and enter kitchen:
1.2.1.      Use the bronze key on the key ring to unlock kitchen
            door.
1.2.2.      Open door and enter kitchen.
1.3.   Control panel is on the wall, to the right of the kitchen
       door. Turn and face it.
2. Disarm alarm system:
2.1.   On the control panel is a set of 10 buttons labelled with the
       digits 0-9 and another key marked Enter.
2.2.   Press the following buttons in order: 4, 6, 2, 8, enter.
2.3.   WHILE LCD Display does not say "Dis-Armed"
2.3.1       Press the following buttons in order: 4, 6, 2, 8, enter.
2.3.2       IF the alarm starts beeping THEN
2.3.2.1           The alarm siren may start sounding. Ignore it.
       ENDWHILE
2.4.   Alarm is disarmed. Algorithm complete.
```

Evaluation and testing of the algorithm yields positive results. Your friend runs through the algorithm and all goes well. However something has been omitted. It is important and it was omitted because the problem statement was incomplete. Do you know what has been omitted? Have a think before reading on to see if you are correct.

The major omission is there are no instructions as to how to arm the alarm and leave the house after the fish have been fed and the plants watered. This means there were three sub problems not two:

```
1. Get to the control panel.
2. Use the control panel to turn off the alarm system.
3. When ready to leave, arm the alarm and lock up the house.
```

Adding and solving the last sub problem yields:

```
1. Enter the house and go to the control panel.
1.1.   Enter garage:
1.1.1.      Face garage door and press the button on the white remote
            control.
1.1.2.      Wait for garage door to open and then walk into the
            garage and face the kitchen door.
1.2.   Open kitchen door and enter kitchen:
1.2.1.      Use the bronze key on the key ring to unlock kitchen
            door.
1.2.2.      Open door and enter kitchen.
1.3.   Control panel is on the wall, to the right of the kitchen
       door. Turn and face it.
2. Disarm alarm system:
2.1.   On the control panel is a set of 10 buttons labelled with the
       digits 0-9 and another key marked Enter.
2.2.   Press the following buttons in order: 4, 6, 2, 8, enter

2.3.   WHILE LCD Display does not say "Dis-Armed"
2.3.1       Press the following buttons in order: 4, 6, 2, 8, enter.
2.3.2       IF the alarm starts beeping THEN
```

```
2.3.2.1          The alarm siren may start sounding. Ignore it.
      ENDWHILE
2.4.  Alarm is disarmed. Fish feeding and plant watering may
      commence.
3. When ready to leave the house, arm the alarm and lock up the
   house:
3.1.  press the button marked "Arm" on the control panel.
3.2.  Proceed from the kitchen into the garage.
3.3   Close the kitchen door and lock it.
3.4   Exit the garage.
3.5   Face garage door and press the button on the white remote
      control.
3.5.  Alarm is armed, house is locked, algorithm is complete.
```

As it turns out, there was one other deficiency in the algorithm but because the algorithm was being executed by a human it did not cause any problems. It turns out that on one visit the power was out and so the garage door would not open. The human thing to do was to go away and return again when the power was restored. A robot following this algorithm would have simply pressed the garage remote button once and waited for the garage door to open. You would have returned from your holiday to find a rusty robot with a flat battery standing in your driveway (possibly covered in graffiti depending upon the socio-economic status of your neighbourhood) along with a houseful of dead plants and a fish tank full of deceased fish.

Each failed algorithm, yields a better understanding of the solution required. In other words the failure itself tells you more information about the problem which in turn leads to a refined solution. This exactly the same process you employed when proof reading your essays at school. This is critical because the number one reason many students give up on algorithm design is simply that their first few attempts fail and they give up, rather than learn from the failure. The last algorithm is good but nobody (myself included) could have come up with it on the first attempt. A solution was only possible by identifying the sub problems, developing solutions for them and continually analysing and refining the algorithm.

Conclusion

Hopefully this chapter has helped you understand that algorithm design requires patience and a methodical approach. Hopefully you have also started to realise that, no matter how complex the problem, this approach will always lead to a successful solution (provided the problem is solvable in the first place!). When designing algorithms for computers, there is another issue which needs to be considered. Most algorithms work by manipulating information. Humans deal with this implicitly. For example, our burglar alarm algorithm did not need to specify where information about the status of the garage and kitchen doors (i.e. locked or unlocked) should be kept. Algorithms for computers need to specify where information is to be stored as well as what form it should take. The next chapter will discuss this issue.

Chapter Two

"You know", said Arthur, "It's moments like this, when I'm trapped in a Vogon airlock with a man from Betelgeuse, and about to die of asphyxiation in deep space that I really wished I'd listened to what my mother told me when I was young".
"Why, what did she tell you?", replied Ford.
"I don't know, I didn't listen!"
Douglas Adams, The Hitch-Hiker's Guide to the Galaxy", 1979.

Data

Introduction
Data is the information processed or produced by a computer in the execution of an algorithm. As discussed in Chapter One, when humans perform tasks they either do not need to store information or they make notes or possibly even tell a helper to remember or watch for something. Algorithms for computers must explicitly state exactly what information is to be stored and where, in the computer's memory, it is to be placed. The exact form of the data can be vague with humans. For example, determining whether or not a number should be integer or real is never a consideration that humans have to deal with explicitly. A computer does. Nobody really knows exactly how complex information is represented in the human mind. In a computer we have the ability to store 0's and 1's and that is all. By grouping zeros and ones we can represent integers. By grouping integers we can represent real numbers (i.e. the integer part, the fractional part and the power of 10 are represented as three integers). Character information is represented using a character code, where we match each character to an integer code number. This means that anything complex must be represented by a combination of numbers and possibly characters. The limitations of these representations cause a lot of complexity in computer algorithms. For example, if we represent colour as a proportion of red, green and blue then we could use three integers, one for red, green and blue. The complexity arises because of the precision involved. For example what if we wanted to determine if a colour was red. Naively we could say that the integer values for green and blue should be zero and the value for red should be at its maximum. But what about light red and dark red? How red does the colour have to be? This situation gets worse if we are talking about a non primary colour, for example purple. Note that identifying a colour is something a human can do easily. This means that we have to choose the most appropriate representation for our data and then our algorithms have to work around the limitations that result from our decision.

Variables

Memory is cut up into a series of memory locations. Data can be placed in these memory locations. Each memory location is considered as a *constant* or *varying*. Constant means the information will not change during the execution of the algorithm and varying means it will. Constant memory locations are, strangely enough called *constants* and varying memory locations are called *variables*.

Each variable or constant has four characteristics:
1. It has a name (referred to as an *identifier* because it literally identifies which variable is being referred to). The name is not specified as a memory address but as a word which describes what the variable or constant represents and how it will be used (e.g. employeeAge).
2. It has a memory location assigned to it. This memory location is where the required information will be stored. In most languages the actual memory location is never referred to explicitly.
3. It has data. This is the actual information stored at the memory address of the variable. i.e. the collection of 1's and 0's that represent something.
4. It has a Data Type. As previously described, we must specify which representation is being used (i.e. integer, real etc.). i.e. the data type determines how the 1's and 0's will be interpreted by the computer.

This book is not intended as a programming language text and so the details specific to particular programming languages will not be discussed. In fact, the specification of exactly what variables are required in a computer program must be done at the start. When designing algorithms, the designer will simply invent variables as they go. They may revisit the variables they have specified and modify them but the exact specification is left to the coding stage. At this point we might specify that a variable is numeric or not but no more than that. The only exception would be an algorithm which was based around the properties of a particular data type (e.g. an integer).

During the course of an algorithm, variables will need to be initialised, modified and referred to. Initialisation of a variable is the act of placing the very first data item into it. ***Prior to the initialisation of a variable, it must be assumed that the value of the variable is unknown.*** There are two ways of initialising or assigning a value to a variable:
1. Input a value from the user or from a data file.
2. Evaluate a specified expression and assign the result to the variable.

Look at the algorithm below:

```
INPUT number1
INPUT number2
sum = number1 + number2
OUTPUT sum
```

The algorithm inputs two numbers from the user, stores them temporarily in two variables (number1 and number2). It then adds these two numbers together and stores the result in the third variable (i.e. sum). Finally it outputs

the contents of the sum variable to the user.

Before Algorithm Step	number1	number2	sum
`INPUT number1`	unknown	unknown	unknown
`INPUT number2`	4	unknown	unknown
`sum = number1 + number2`	4	12	unknown
`OUTPUT sum`	4	12	16

Table 2.1. The contents of the variables during the execution of the algorithm.

Let us suppose that the user enters the values 4 and 12 when asked to. Table 2.1 shows what is stored in the variables during the execution of the algorithm. Note how the steps in the algorithm never attempt to refer to the contents of a variable until *after* that variable has been initialised.

This algorithm illustrates the three ways variables are accessed:

- Initialisation: Assigning the very first value to a variable.
- Reference: Reading the value from a variable.
- Update (or mutate): Changing the value of a variable.

Variables should always be initialised before they are referenced. The initialisation should always be as close as possible to the variable's first reference. Some designers argue that all variables should be initialised at the start of an algorithm. This is not a good idea because this maximises the distance between initialisation and first reference. Also it is inefficient because variables which are going to be initialised via user input or via some calculation which cannot occur at the start cannot be initialised at the start anyway. This practice is encouraged as a way of ensuring that all variables will have a sensible value at their first reference but there is no way of knowing whether setting a variable to zero (or some other arbitrary value) will be helpful anyway. Further this implies that the algorithm is not going to be well thought out because, if it was, all the variables will be appropriately initialised at the appropriate times.

In our algorithm, notice that the use of the equals sign is different to the normal mathematical use. In mathematics if we stated:

```
x = y + 1
```

Then we are stating a *truth*. i.e. x is, in point of fact, equal to y + 1. In mathematics we could use this truth to rearrange the equation (for example to y = x – 1).

When we state a similar thing as a line in an algorithm, we are stating a desired *action*. i.e. this step of the algorithm should *make* x equal to y + 1. There are two tasks required for this step:

1. Evaluate y + 1. This involves taking a copy of the number stored in the y variable and adding 1 to it.
2. Store the result in x.

Finally the exact type of variable is not stated. When designing algorithms this is quite normal. Most of the time all we care about is that the information is numeric or character based. Sometimes, however it can be an issue. Consider the algorithm below:

```
INPUT number1
INPUT number2
ave = number1 + number2
              2
OUTPUT sum
```

In the above algorithm, the implication is that the variables would hold real numbers because real division is more accurate than integer division. If, for some reason, we wanted the division to be done as an integer division then we would have to state that in the algorithm.

There are four basic types of variables that most programming languages support:

- Integer numbers
- Real numbers
- Characters
- Boolean

As human beings, a life time of experience assists us in examining the context for our calculation and then using that as a means for selecting the right data type. Computers on the hand, have no idea, so if the choice between real and integer is crucial then the algorithm designer should state which is required for a particular variable. The following sections will cover the specific details of each of these different data types in a generic manner. i.e. The descriptions are not specific to a particular programming language.

Numeric Variables

As stated above, the two basic types are *integer* and *real*. They follow all of the basic principles from standard mathematics with the following exceptions:

- There is no way to represent infinity.
- When coding your algorithm into a particular programming language, if you attempt something which is mathematically invalid (e.g. dividing a number by zero) then your program will behave unpredictably.
- In mathematics a number can be infinitely long. In a computer a finite amount of memory will be assigned to that variable. This has the effect of placing limits on the maximum and minimum possible values allowed for integer data types. In the case of real numbers, their accuracy decreases as the numeric values get very, very small or very, very large. In the case of integers, if the amount of memory required exceeds the memory available then integer overflow (see discussion below) is said to have occurred.

Numeric expressions in algorithms can be exactly the same as that in

mathematics. As everyone reading this book should be reasonably familiar with mathematical concepts there is no need to discuss numeric expressions further.

Variables provide the storage and *literal values* provide the specific data for storage. Examples of literal values for real and integer variables are shown in Table 2.2 below.

Data Type	Examples
Integer	3, 2, 21, -128, 44
Real	3.0, 2.0, 22.4, -4.6, $2.3+10^4$

Table 2.2. The two different numeric data types and their literal values.

Notice that all of the literal values for real numbers should always include the decimal point (e.g. not 2 but 2.0). If these algorithms were not going to be converted to computer software then this distinction is not required. Many programming languages will yield incorrect results for numeric expressions when integer literal values are provided instead of literal values for real numbers (C and C++ are classics in this regard!). Therefore it is good practice to make the distinction, even in an algorithm.

Every variable will have a defined size in terms of the amount of memory allocated to it. As numbers become larger (or smaller) the algorithm designer has to keep in mind the consequences of what will happen when the memory allocated to a variable is no longer large enough.

Regardless of the programming language concerned, all data is stored in memory as binary data. Integers are stored as two's complement binary numbers. This isn't a book on machine code programming so let's not concern ourselves with exactly what is two's complement binary. It is enough to say that a collection of bytes will be allocated to each integer variable. The base 10 value is translated into binary and stored in this memory. When the base 10 value requires more memory than is available for it then the part of the number which does not fit is simply discarded. This is known as *integer overflow*. This means that if we are unsure as to whether or not integer overflow will occur, then our choice of data type, in combination with our algorithm must ensure that integer overflow will never occur. For example, suppose we wished to use an integer to count a population. If the maximum possible population was relatively small (say less than one million) then using a numeric integer will probably be okay. However suppose we wished to count the number of grains of sand in a desert (heaven knows why!) then integer overflow is guaranteed to occur. So what can be done to avoid this? Let us apply our problem definition skills! There are only three types of operations that we need to perform on a counter:

1. set it to zero
2. add one to it
3. display its current value

With that in mind, we could use a character string (see below) where each

character represents 1 digit in the integer. We would have to devise an algorithm to simulate *adding one* and, although a bit fiddly, this can be done. Armed with this we could, if we were insane enough to do so, go ahead and start counting grains of sand!

Integer overflow cannot occur with real data types. The equivalent problem is that as the number gets larger or smaller its accuracy is reduced. There is another issue with real numbers though. That is one of *required* accuracy. When dealing with real numbers the computer will always make full use of the maximum amount of precision available to it at any point in time. The problem is that this precision will vary with the size of the numeric value concerned and also the resulting precision might be inappropriate to the context of the algorithm being devised. For example dealing with money requires two decimal places.

Finally there are mathematical concepts concerning real numbers which cannot be represented with real data types. Infinite precision (e.g. irrational or recurring numbers) and infinity are the two most obvious examples. This means that the algorithm designer has to allow for these issues when designing their algorithms.

Character Variables

Character variables are fairly straight forward. Quite simply any printable character on a keyboard be it a letter of the alphabet, a digit or a punctuation symbol. Variables which hold a single character are not usually very useful and so programming languages often have a data type which can hold a collection of characters. This type of data type is often referred to as a *character string*. Normally character variables cannot be evaluated via expressions but some programming languages allow the use of the plus symbol to represent concatenation (i.e. the joining) of two character strings. For example:

```
fullName = firstName + lastName
```

Literal values for characters are represented by typing the literal characters inside single or double quotes. For example:

```
firstName = "Roger"
lastName  = "Rabbit"
fullName = firstName + " " + lastName
```

Boolean Variables

The Boolean data type is used to represent true and false. There are only two literal values (i.e. the words true and false). In algorithms they are extremely useful because we can set Boolean variables via Boolean expressions and then execute alternate steps based upon whether or not a particular variable is true or false.

Boolean expressions are simply a formal expression of things that every

human deals with on a daily basis. The simplest Boolean expressions simply compare two expressions. Table 2.3 below shows all of the Boolean operators.

Operator	Meaning	Example
=	equal to	name1 = name2
<	less than	age < 18
>	greater than	age > 65
	less than or equal to	age 12
	greater than or equal to	age 18
	not equal to	vehicle1 vehicle2

Table 2.3. The basic Boolean operators.

This allows us to ask questions about the values of variables or expressions and then make a decision based upon the truth or falseness of the Boolean expression. For example:

```
IF bloodAlcoholLevel > 0.006 THEN
   do not drive home
ENDIF
```

Often times there is the need for more complex Boolean expressions. These expressions are created by joining together simple Boolean expressions with relational operators. There are three relational operators and they are described in table 2.4

Relational Op.	Meaning
AND	Joining two Boolean expressions together with AND means that both expressions must be true for the complete expression to be true.
OR	Joining two Boolean expressions together with OR means that if either expression is true then the complete expression is true.
NOT	Reverse the meaning. i.e. true becomes false and false becomes true.

Table 2.4. The three types of Boolean relational operators.

You have been using these relational operators all of your life. You may not have thought of them in as formal a manner, but you have used them when giving instructions to others and followed instructions which contain them. Consider the examples below:

```
IF breadSlices > 0 AND jamRemaining > 0.0 THEN
   make jam on toast
ENDIF
```

We can also use Boolean expressions to initialise Boolean variables to true or false. Consider the example below:

```
equilateral = side1 = side2 AND side1 = side3
```

The variable equilateral will be set to true if all three sides are equal and false otherwise. This example also illustrates some of the subtlety that can arise with Boolean expressions. Notice that there is no need to check if side2 is equal to side3. If a third Boolean expression was added to produce:

```
equilateral = side1 = side2 AND side1 = side3 AND
              side2 = side3
```

The result would be the same but the third clause is redundant. This is because if side1 is equal to both side2 and side3 then that means that side2 and side3 are equal to the same value (i.e. side1) and hence there is no need to explicitly check. Notice that the where the equals sign is used determines whether the algorithm is asking if the left side is equal to the right side (e.g. side1 = side2) or requesting that the left side be made equal to the result of the expression on the right side (i.e. equilateral = etc). Humans can work the context out but computers cannot so all programming languages will have different operators for each. For example C/C++ use = to assign the result of an expression to a variable and == to check for equality:

```
equilateral = side1 == side2 && side1 == side3 &&
              side2 == side3;
```

Whereas Delphi uses := for assignment and equals for comparison:

```
equilateral := side1 = side2 AND side1 = side3 AND
               side2 = side3;
```

The table below is known as a Truth Table. This truth table enumerates all of the possibilities for a Boolean expression involving AND, OR or NOT.

expr1	expr2	expr1 AND expr2	expr1 OR expr2	NOT expr1
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Table 2.5. The truth table for Boolean Relational Operators.

Notice that AND and OR work in opposite ways. i.e. AND is only true when both operands are true and OR is only false when both operands are false. This is an important concept because it leads to the fact that the logical negation of AND is OR and vice versa. Consider the Boolean expression below:

```
age > 10 AND age < 60
```

Its logical negation would be:

```
NOT ( age > 10 AND age < 60)
```

which is equivalent to:

```
age <= 10 OR age >= 60
```

The first expression is checking to see if the value contained in the age variable is between 10 and 60. Therefore its negation must check to ensure that age is not in the range 10 to 60. That means that age must be either less than or equal to 10 or greater than or equal to 60.

We have to be very careful with negation because we do not always get what we expect. Consider the expression to determine if the three sides of a triangle are equal:

```
side1 = side2 AND side1 = side3
```

If we negate that expression we obtain:

```
NOT(side1 = side2 AND side1 = side3)
side1   side2 OR side1   side3
```

While this expression is the precise opposite of the original it does not determine if all three sides are different, only that all three sides are not all equal. In other words if we don't think very carefully about precisely what is required then we end up with an invalid Boolean expression. What is the Boolean expression which is true if all three sides are different lengths? That would be:

```
side1   side2 AND side1   side3 AND side2   side3
```

Which is expression is the one required. I have no idea! The reality is that it would depend upon the problem being solved and its requirements in terms of triangle side inequality.

Careful What You Wish For

The problem with dealing with variables and expressions is that it can be very easy to come up with a series of algorithm steps or an expression which is close enough to what is required that a human could perform the algorithm correctly but is not exactly right. In this case the old saying "Miss by an inch then may as well have missed by a mile!" applies. While the human will make small but critical variations to what is stated in the algorithm, a computer will not.

As an example of how easy it can be to go wrong, let's look at the very simple task of swapping the contents of two variables. This is a common step in

many algorithms. Newcomers to algorithm design usually come up with the steps below:

```
num1 = num2
num2 = num1
OUTPUT num1 and num2
```

Confusion then reigns because, rather than perform as expected, the result is that both num1 and num2 end up with the value that was stored in num2 prior to the swap. The reason as to why is best illustrated with an example. Suppose that prior to the swap num1 contained 4 and num2 contained 6. Table 2.6 below, shows what happens when the algorithm is executed.:

Before Algorithm Step	num1	num2
`num1 = num2`	4	6
`num2 = num1`	6	6
`OUTPUT num1 and num2`	6	6

Table 2.1. The contents of the variables during the execution of the algorithm.

In other words the original contents of num1 is overwritten by the contents of num2. What is required is a temporary variable as in the algorithm below:

```
tempNum = num1
num1 = num2
num2 = tempNum
OUTPUT num1 and num2
```

Before Algorithm Step	num1	num2	tempNum
`tempNum = num1`	4	6	unknown
`num1 = num2`	4	6	4
`num2 = tempNum`	6	6	4
`OUTPUT num1 and num2`	6	4	4

Table 2.1. The contents of the variables during the execution of the algorithm.

The significant point is that a human executing the first (and incorrect) swap would have got the correct answer because they knew the intent of the steps and would have remembered the old value of num1 and kept it aside for num2. The human would NOT have followed the algorithm precisely because of common sense. How much common sense does a computer have? None! so the computer would just get it wrong.

How are the variables required chosen?

The variables required will be selected as the algorithm is being developed. It would be naive to think we could come up with a list of variables before we

had worked out what the steps to the algorithm were. Probably we can come up with some but we won't know exactly until we have worked out what the steps are. Its a good idea to keep a list of variables, adding to it as you develop your algorithm. Attempting to predict the variables needed before hand often results in variables which end up not being used.

For example, suppose we wanted an algorithm which would allow us to create a budget for renting a house. The starting point is the problem definition. For that we need to think about exactly what information we want specified in our budget. Discussions with real estate agents reveal there are three costs which need to be known:

1. How much money is required to secure the lease.
2. How much money needs to be paid each fortnight.
3. What is the total cost of the lease.

The above three descriptions immediately imply three variables (i.e. deposit, rentPayment and totalLeaseCost). There will be others but at the moment we know we need these because they will be the repositories for the information that we need to calculate.

The next step is to divide our overall problem into sub problems. This problem neatly divides into three sub problems:

1. Input the data required for the calculations
2. Calculate deposit, rentalPayment and totalLeaseCost
3. Output deposit, rentalPayment and totalLeaseCost.

Note that, at this point, we do not know what precisely what to input, we don't know what the calculations are. We only know the result we want in terms of output. Instinctively the next step would be to work out the algorithm for the first step (isn't starting at the beginning the logical place to start?). That would be a wrong move because we won't know what to input until we know what the calculations require and we won't know that until we work out what the calculations are. Hence the next step is to work out the calculations. More discussions with real estate agents reveals that, to secure a lease, a tenant must pay:

- Four week's rent in advance. This is to ensure that the tenant is always paid up at least one month ahead of time.
- A Bond. The bond will be the equivalent of four week's rent and in the event of any damage being done to the property by the tenant, the cost of repairs will be taken out of the bond. The tenant will get the bond back (minus any repair costs) at the end of the lease.

After that the tenant is required to pay rental payments each fortnight. This means that:

- The deposit is equal to 8 weeks rent.
- The fortnightly rental payments are equal to 2 weeks rent.
- The total cost of the lease is the weekly rent times the length of the lease in weeks plus any repair costs taken from the bond.

Analysis of this information reveals an ambiguity. The deposit also needs to include the first fortnightly rental payment. Allowing for this observation we can now work out the calculations:

```
deposit = 10 x weeklyRent
rentalPayment = 2 x weeklyRent
totalCost = leaseDuration x weeklyRent
```

Notice that our list of variables has increased by the addition of weeklyRent and leaseDuration. If we look at the calculations, its these two variables whose values are assumed prior to the calculations. Therefore these are the values to input from the user. Hence our algorithm becomes:

```
INPUT weeklyRent
INPUT leaseDuration
deposit = 10 x weeklyRent
rentalPayment = 2 x weeklyRent
totalCost = leaseDuration x weeklyRent
OUTPUT deposit, rentalPayment and totalCost.
```

In terms of data types, the variable which is counting (i.e. leaseDuration) will end up as an integer data type in code. The variables representing money will probably also be integers because rental payments are usually expressed in dollars and not dollars and cents. The only other factor that might effect this decision is if division was involved in the calculations but it isn't.

Conclusion

Variables and data types are required in order to construct algorithms. While to concepts and principles are similar to what you have seen in Mathematics, its not exactly the same and also we have non-numeric data types. The most significant of these being Boolean. Never the less, this chapter should have provided a good insight into how variables work. The next step along the path to brilliant software designer is too look out how the actual algorithms steps are designed and expressed.

Chapter Three

"Ford!", said Arthur, "There's an infinite number of monkeys outside who want to talk to us about this script for Hamlet they've worked out."
Douglas Adams, The Hitch-Hiker's Guide to the Galaxy", 1979.

Introduction to Pseudo Code and Sub Modules.

Introduction

Algorithm design remains the most difficult process that exists in the computing world today. As was discussed in Chapter One, algorithm design employs problem solving strategies and requires a methodical and disciplined approach in order to achieve positive results. Many newcomers to software design mistakenly believe that learning to code in a programming language is all there is to building software. This misconception often leads non-computing professionals to believe that, once they understand how to write small programs, then they too can join the ranks of commercial software developers. This is the equivalent to me thinking that, because I can fly model planes I am a viable contender for the next Redbull Air Race! The complexity of the requirements that commercial software has to meet means that the actual coding is a small part of the effort involved. When designing and implementing software, there are many areas of difficulty ranging from extracting the requirements from the clients to installing and testing the software on site. At the core though, will be two tasks:
1. The design of the algorithm.
2. Implementing the algorithm in a programming language.

Professional Software Engineers can often do these tasks together. Students cannot so students are much better of designing then implementing. The basic idea is to deal with these problems separately and so one big complex problem is divided into two sub problems. The difficulties encountered in coding are usually simpler and more straight forward to overcome than those encountered in the design process. One thing which was not discussed is a mechanism for keeping our sub problem solutions separate while at the same time allowing some way of representing how these sub solutions come together to form the overall solution. In the case of algorithm design, the concept of sub modules allows this. Before we discuss sub modules though, lets take a quick look at how we represent algorithms and how this form of

expression evolved.

In order to avoid coding issues, an algorithm has to be expressed in a manner which is generic. i.e. it should not contain any information which is specific to a particular programming language. Also it should not deal with detail that can easily be added later. The latest form for expressing algorithms is called *Pseudo Code*. Put simply, pseudo code is simply the stating the steps required by the algorithm in an English like form. The major difference between pseudo code and program code is that pseudo code is intended to be read and understood by humans and program code is intended to be read and interpreted (as much as is required for compilation and execution) by a computer. This means that a lot of the detail that must be included in program code does not need to be included in pseudo code. Consider the simple algorithm below:

```
INPUT numberOne, numberTwo
sum = numberOne + numberTwo
OUTPUT sum
```

Note that an algorithm meant to be executed by humans would not require the first and last step because they are implied. An algorithm to be understood by humans and later translated into program code for a computer needs these steps because if they are missing in the final result (i.e. the program code) then they won't happen. The amount of trivial detail that must be added to translate this pseudo code into any programming language will increase the amount of text by at least doubling it! In the C programming language, the resulting program code would look like:

```
#include <stdio.h>

int main()
   {
   int numberOne, numberTwo, sum;
   fprintf( stdout, "Enter first number:");
   fscanf( stdin, "%d", &numberOne);
   fprintf( stdout, "Enter second number:");
   fscanf( stdin, "%d", &numberTwo);
   sum = numberOne + numberTwo;
   fprintf( stdout, "\n\n\tSum of input is %d\n\n", sum);
   }
```

Three lines of pseudo code mushrooms out to 11 lines of C code. An increase of 400%! Not to mention all the strange and weird uses of ampersand, curly brackets and semi-colons. Other programming languages would have even more code. While our algorithm has to deal with a lot more detail than algorithms intended for humans (e.g. cooking recipe), we do not want to bury ourselves in parenthesis, semi-colons and other programming language trivia. We need to focus on solving the problem without dealing with language syntax. Interestingly the strategy of separating out the algorithm design from the coding was not recognised in the early days of computers. This was because the processing power and memory available was so small that

complex problems could not be solved anyway. As processing power and memory increased, so did the size and complexity of software. This lead to the realisation that this separation was required but what was not known was how the algorithms being designed should be expressed.

A Historical Perspective on Algorithm Design.

The emergence of high level programming languages brought with them the promise of easing the difficulty in writing computer programs. This promise was kept to a very small extent but programmers still hit brick walls in terms of the size and complexity of problems that they could write programs to solve because they still required separation of the task of problem solving from demands of programming language grammar and syntax. The problem was in working out a programming language independent way of expressing algorithms. Flow Charts were an attempt to express algorithms via diagrams. This was driven by the idea that a graphical representation was a good vehicle for problem solving. A reasonable premise given that humans often employ diagrams to express solutions to problems. If you think about it most mathematics is expressed graphically (think about summations and integrals) The idea with flow charts was that each step in an algorithm is placed in a box and then the flow from one step to another was indicated by lines connecting the boxes.

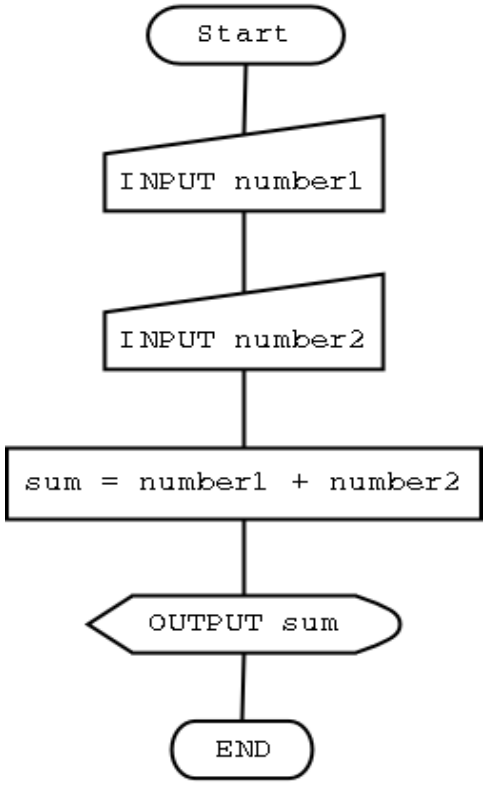


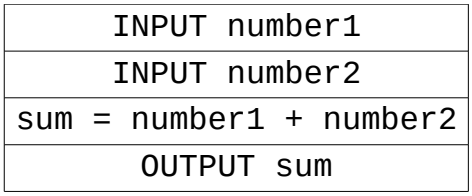
Figure 3.1 A Flowchart for a simple algorithm

Figure 3.1 shows the flow chart equivalent of the previous algorithm example. The different shaped boxes were intended to represent different types of steps

(e.g. input, output). Flow charts turned out to be a complete and total failure. They failed for two reasons:

1. The syntax for expressing flow charts is almost as complex and fiddly as an actual programming language. i.e. instead of being tangled in where the semi-colon goes, the designer is caught up with which box to use! The result is that a flow chart can only be drawn after the algorithm is designed.
2. Flow charts did not allow a number of useful mechanisms. The most significant of these is the use of loops for repeating a set of actions a number of times (loops are discussed in great detail in chapter 5).
3. Most significantly, flow charts took to long to draw and were difficult to modify. As we have already seen, the solution to a problem needs to evolve, so this last point is the major reason for the failure of flow charts as an algorithm design tool.

Of course these days, nobody in their right mind would use a flow chart. There was an attempt at refining flow charts into something better. These were known as Nassi-Schniederman Diagrams (or N-S Diagrams for short, also referred to as Nasty Spiderman diagrams by students who can't remember how to spell Schniederman). N-S diagrams were a step in the right direction because they addressed the first two limitations raised above, they also hinted at the solution to the final point. Unfortunately they were still difficult to evolve and time consuming to draw..



One advantage that N-S diagrams did have over flow charts was they encouraged designers to provide more detail in the steps than with flow charts. This lead to the idea of keeping the description of each step but to throw away the graphical representation. Hence pseudo code was born. Put simply pseudo code is simply expressing the steps in words. As a refinement tool it works well because, as mentioned previously, most humans are using to writing essays, reports and other documents in a similar manner. In other words, the idea of refining a body of text has been with us for as long as we have been able to write. The algorithms described in the previous two chapters have all been expressed in pseudo code. The idea is that the exact words are not important as long as the intent is clear and that the pseudo code needs to evolve from vague descriptions to more precise statements which can be translated into the chosen programming language.

At this point we should make a distinction between the terms algorithm and pseudo code.

- An algorithm is the set of steps which when followed, solve a problem.
- Pseudo code is the use of English like grammar to express each algorithm step.

In other words the algorithm is the solution and pseudo code is the language

that the solution is expressed in.

Step Wise Refinement

Part of the process described in chapter one, involved breaking the problem into sub problems and then developing a solution for each sub problem. This technique is known as *Step Wise Refinement*. The basic principle is to simply try to keep the number of steps being considered at any point in time to a bare minimum. Though simple in principle, the ability to practice step wise refinement well is probably the most difficult goal for any computing student to achieve. Remember from the discussion of problem solving in chapter one, we observe that, although this technique is practised in the real world, it tends to be something we do by intuition rather than conscious thought. This is essentially an artistic ability which can only be achieved through years of discipline and constant practice.

Any skill which evolves through practice is best demonstrated by example. As an example, consider the problem of determining how much paint is required to paint the walls in a room and how much the paint will cost.

The starting point is to understand the problem. The paint required comes in 4 litre tins. Each tin has a specified area coverage (specified in square metres per litre). The number of coats of paint required needs to be specified by the user. This means, in order to do the calculation, the algorithm needs to know how many coats and the area of the walls There may be other input required but, at this point, these are the obvious ones. Hence the first draft of the algorithm is:

```
INPUT numCoats
INPUT wallArea
calculate paintVolume
calculate numPaintTins
calculate totalCost
OUTPUT totalCost, numPaintTins
```

The calculations are where the refinement has to occur:

- The total cost will be the number of 4 litre tins multiplied by the cost per tin.
- The number of tins will be calculated as a function of the volume of paint required (expressed in litres).
- The volume of paint required will be calculated as a function of the total wall area, the number of coats and the area that can be covered by 1 litre of paint.

Notice that, in this case, it is easier to reason things out by starting with the last calculation and working back to the first. This is often the case but not always the case, again this is where lots of practice develops the skills required. An analysis of the above points yields the variables below:

Variable	How it gets initialised
totalCost	Calculated by algorithm
costPerTin	Input from user
totalVolume	Calculated by algorithm
wallArea	Input from user.
coveragePerLitre	Input from user.
numCoats	Input from user
numTins	Calculated by algorithm

This means our algorithm now becomes:

```
INPUT costPerTin
INPUT wallArea
INPUT coveragePerLitre
INPUT numCoats
calculate totalVolume
calculate numTins
calculate totalCost
OUTPUT totalCost, numTins
```

We still need to refine the calculations. The total volume is going to be calculated by dividing the total area by the per litre coverage. The total area is the wall area multiplied by the number of coats. Hence:

$$totalVolume = \frac{(wallArea \times numCoats)}{coveragePerLitre}$$

$$numTins = \frac{totalVolume + 1}{coveragePerLitre \times 4}$$

$$totalCost = numTins \times costPerTin$$

This means our finished algorithm is:

$$totalVolume = \frac{(wallArea \times numCoats)}{coveragePerLitre}$$

$$numTins = \frac{totalVolume + 1}{coveragePerLitre \times 4}$$

$$totalCost = numTins \times costPerTin$$

```
INPUT costPerTin
INPUT wallArea
INPUT coveragePerLitre
INPUT numCoats
```

In evolving this solution we followed the process outlined in the previous chapters. We broke the big problem down into a series of sub problems. For the calculations, we evolved our solution from a vague description to precise steps and finally we had the algorithm we needed. What would have been handy is a method for expressing the overall solution by referring to the

solutions for each sub problem and then being able to refer to each sub problem on its own. Unfortunately this sub problem isolation is lost in the final algorithm because, at the end of the process, all of the steps are grouped together in the same place. This is a very simple algorithm. A commercial software application will involve hundreds of thousands of steps! If the steps where all jammed together what a mess we would have! It would be better if we could actually preserve this isolation in our algorithm. This would be very useful when detecting or correcting errors or modifying an algorithm to perform a different task. Sub Modules are used to isolate the sub problem solutions from each other.

Sub Modules

The idea is that a set of algorithm steps are grouped together into a sub-module. This sub module can then be *called* from other parts of the algorithm. When a sub module is called the algorithm execution moves from where the sub module was called into the sub module itself. A *modular* algorithm is one which makes good use of sub modules to partition the overall solution into a cohesive set of sub modules. A sub module is *called* by another part of the algorithm. When this happens, control passes from the part of the algorithm which is doing the calling to the sub module. The sub module then executes its algorithm and then control is passed back to the calling part of the algorithm which continues its execution. There has to be a starting point for this process. This is also a sub module and is referred to as the *main* sub module. Each sub module consists of:

- A name.
- A list of information which is passed into it when it becomes active. This information is known as IMPORT because it is imported *into* the sub module when it becomes active.
- A list of information which it passes back to the calling sub module when the sub module is finished. This information is known as EXPORT because it is exported *from* the sub module to the calling module when the sub module finishes.
- The pseudo code describing the algorithm that the sub module executes.

Suppose we wished to design an algorithm which converted a measurement in inches to a measurement in centimetres. An simple algorithm would be:

```
INPUT inches
cms = inches x 2.54
OUTPUT inches, "inches is ", cms, "cm
```

Suppose we wished to do the output and the conversion in sub modules. We would then divide our algorithm up into three separate sub modules:

- A top level or main module which is where the algorithm starts and finishes.
- A sub module for converting from inches to centimetres.
- A sub module for performing the output.

This means our main algorithm would look like:

```
MAIN
INPUT inches
CALL convertToCms
CALL outputResults
```

One important point to realise is that the variables referred to in one sub module should never be referred to in another sub module. This means we have a problem. The convertToCms sub module needs to know what the value for inches is and then it needs to inform the main sub module as to the result of its centimetres calculation. Similarly, the outputResults sub module needs to be informed of the centimetres and inches values so that it can output them.

This is achieved by specifying:

- IMPORT DATA: the information required by the sub module. It is passed into a sub module when it is called.
- EXPORT DATA: the information which the sub module passes back into the calling sub module.

Note that IMPORT and EXPORT data for a particular sub module can only be determined *AFTER* the algorithm for that sub module has been designed. The variables in the algorithm which are required to already contain a value prior to the start of the algorithm (e.g. inches in the convertToCms sub module) form the list of IMPORT data and the EXPORT data is the information required by the calling sub module. Hence using sub modules our inches to cm algorithm becomes:

```
MAIN Algorithm:
INPUT inches
CALL convertToCMS<--inches -->cms
CALL outputResults<--inches, cms

SUB MODULE convertToCms:
IMPORT inInches
EXPORT outCm
ALGORITHM
outCm = inInches x 2.54

SUB MODULE outputResults
IMPORT inInches, inCms
EXPORT None
ALGORITHM
OUTPUT inInches, "inches is ", inCms, "cm"
```

The use of arrows (i.e. <---, --->) is intended to indicate the flow of information into or out of the called sub module where it is called. Note the exact notation is unimportant as long as it is clear what is coming in or going out. The second significant point is that this information can only be added to the main algorithm *AFTER* the sub modules have been designed.

At this point, I am guessing that the whole sub module thing is starting to

confuse you and the emotional part of you is seriously considering using this book as a fire lighter at this point. Well don't burn the book just yet! Read on and we will trace through the inches to centimetre algorithm to illustrate how the sub module process works.

We start in main by inputting a value for inches from the user. Let's say the user enters the value 1.0 inches. At this point the variable inches (in main) has the value 1.0 stored in it and cms has an undefined value (because our algorithm has not initialised it).

The next step in the main sub module calls the convertToCms module. The convertToCms module has one item of IMPORT data which must be passed to it from the main module. The value stored in inches is copied and then the copy is used to initialise inInches in convertToCms. Control is then passed to convertToCms.

The convertToCms sub module then multiplies inInches (set to 1.0) by 2.54 and stores the result in outCm. Control is then transferred back to the main sub module and the EXPORT information is copied and passed back. The result is the cms gets a copy of the value stored in outCms (i.e. 2.54). The exact mechanisms for transferring data to/from the calling sub module from/to the called sub module differ from one language to the next but the basic principles of IMPORT and EXPORT information remain the same.

At this point I need to remind you that the use of sub modules in this example is overkill. We ended up with a longer, more complex algorithm than the previous 3 step version which did not use sub modules. We used this simple example to illustrate what happens when a sub module is called and how information is moved in and out of it. This raises the issue of when to use a sub module.

Sub Modules, Problem Solving and Algorithm Design

So far we have discussed the following concepts in terms of design pseudo code solutions for algorithms:

- Problem Solving
- Step Wise Refinement
- Sub Modules

These three ideas can be combined together to form an overall strategy for designing algorithms:

1. Design a pseudo code solution for the problem by defining the sub problems. This means that each step will need further refinement (i.e. we are applying step wise refinement).
2. Each step that needs refinement is then converted to a sub module call.
3. The same approach is attempted for each sub module (i.e. follow step 1).
4. After the algorithm for a sub module is complete, the IMPORT and EXPORT data are determined. This step is required for all sub modules except the main sub module.

The result will be sub modules calling sub modules calling sub modules *ad infinitum.* Another point is that the algorithm for a sub module which calls another sub module can only be completely finished AFTER the algorithm for the sub module being called has been completed. This is because you will not be sure of what the IMPORT or EXPORT will be for that sub module until the algorithm is finished. For example the main sub module needs redrafting after the outputResults and convertToCms sub modules have been completed. This is a significant stumbling block for most students who attempt to write an algorithm in one draft, from start to finish. It is also interesting that the students who persist in this strategy would never dream of writing an essay from start to finish in one draft. If you think about it, the first draft of the main algorithm is a bit like working out what the headings will be in your essay. Further step wise refinement is similar to dividing up each section into sub sections in an essay. The starting for each section of an essay will be a few lines stating what the section will say. The analogy only breaks down in that when the essay is finished, all evidence of its step wise design are removed (i.e.. the few sentences stating roughly what will happen have been replaced with paragraphs of brilliantly written text).

Perfecting the art of step wise refinement and modularity only comes with lots of practice. Expect failure but with each failure you need to determine why your solution is wrong. Each realisation amounts to new knowledge. Each bit of new knowledge hones your algorithm design skills just a bit more. This means, from a student's perspective, even failed attempts at algorithm design produce a learning outcome and therefore represent progress along the path to guru algorithm designer.

Sub Module Re-Use

One advantage to sub modules is that they can be called more than once. For example in the inches to centimetres algorithm we are only converting one value from inches to centimetres but if we were converting more than one value, we could call the same sub modules over and over again. The algorithm below shows a new main algorithm which inputs two inches values and calls the conversion and output sub modules twice.

```
Main Algorithm:
INPUT inchesOne
INPUT inchesTwo
CALL convertToCMS<--inchesOne -->cms
CALL outputResults<--inchesOne, cms
CALL convertToCMS<--inchesTwo -->cms
CALL outputResults<--inchesTwo, cms

convertToCms Sub Module:
IMPORT inInches
EXPORT outCm
ALGORITHM
outCm = inInches x 2.54
```

```
outputResults Sub Module:
IMPORT inInches, inCms
EXPORT None
ALGORITHM
OUTPUT inInches, "inches is ", inCms, "cm"
```

There are two major advantages to using sub modules in this way. The first is that the algorithm for each sub module only occurs in one place. If the algorithm needs modifying or correcting we only have to go to that one place in our algorithm to fix the problem once. This is called *encapsulation*. We have built a fence around the part of the problem solved by the sub module algorithm and provided a clearly defined protocol for interacting with it (i.e. IMPORT/EXPORT). The second advantage is that without sub modules, the steps for the algorithm which are contained within the sub module must be repeated over and over again. Our inches to centimetres example is not a good example because the algorithm in each sub module only contains one step. Imagine a more complex algorithm where the calling sub module calls the sub module 20 times. The algorithm for the sub module being called is 30 lines long. Without using a sub module that 30 lines would have to be repeated 20 times in the calling sub module!

IMPORT/EXPORT Revisited

Determining IMPORT and EXPORT data for a sub module can be tricky. Especially when the sub module concerned also requires IMPORT/EXPORT data.

Data can be both IMPORT and EXPORT. Consider the increment sub module below. The variable number gets an initial value from the calling sub module, adds one to it and then its new value is EXPORTed back to the calling sub module.

```
SUB MODULE increment
IMPORT number
EXPORT number
ALGORITHM
number = number + 1
```

Also data can be Transitive. Transitive data is data which is not directly used in a sub module but is being passed from that sub module to another sub module. In the main algorithm for the inches to centimetres algorithm both inches and cms are transitive because the main algorithm never acts on them directly. Their initial value for inches is obtained from the input process. The value for inches is then passed into the convertToCms sub module which uses it to calculate the value for outCm which is then EXPORTed back to main and placed into the cms variable. The the values for inches and cms are then passed from main into the outputResults sub module. So main is acting as a conduit funnelling inches and/or cms in and out of the required sub modules but never actually manipulating them itself.

Identifying IMPORT and EXPORT for a sub module can be tricky sometimes

but following the steps below usually yields the required results:

1. Develop the algorithm.
2. Make a list of all the variables in the algorithm.
3. For each variable:
    1. classify it as IMPORT if it is not initialised by the algorithm. i.e. if the algorithm does not assign it an initial value then its initial value must come from IMPORT.
    2. classify it as EXPORT if the calling sub module needs to know its value for its algorithm.
    3. classify it as neither if the above points are not true.

Like all generalisations, the above steps are not guaranteed to work every time but again I remind you that algorithm design is art.

Testing Sub Modules

When the algorithm for each sub module is completed and the IMPORT/EXPORT data determined, the sub module should then be tested in isolation from the rest of the algorithm. Testing should consist of examining each item of IMPORT data and setting categories for testing. Each category represents a range of data values which test one facet of the algorithm. Testing should consist of all permutations of categories for all of the IMPORT data with verification that the algorithm works correctly for each category.

For example, if we have a sub module which calculates tan of an angle then it should be tested with angles of 0^o, 0.1^o - 89^o, 90^o degrees and so on. Clearly 90^o is a problem but the algorithm must have a way of tangling with infinity. In other words the test data and the results form a description of the behaviour of the sub module under all possible circumstances. Even if this behaviour isn't perfect, it is important that it is known.

If each sub module has been properly tested then the spectrum of problems which can be encountered when the entire algorithm is tested is significantly reduced. Note that I said *reduced* not *eliminated*! Testing sub modules does not mean that the overall algorithm does not need to be tested.

Algorithm Design is not a Sequential Process

The one realisation which comes out of this is that an algorithm isn't designed by starting at the first line of the main sub module and writing it out from start to end in the exact order required. Rather we start with the main algorithm. This leads us to required sub modules. each sub module leads us to more sub modules and so on. The algorithm which calls a sub module cannot be finished until the algorithm for the sub module being called is completed. For example until that algorithm for the sub module being called is completed, we do not know what the IMPORT/EXPORT information is. In other words algorithm design is an iterative process where the first draft of each sub module has many omissions and only after a lot of refinement do we end up with a well designed and complete algorithm. In other words we recognise

the fact that humans are much better at practising hindsight that foresight and make use of that by employing iteration and evolution of our algorithm as part of our design strategy.

Scope of Variables

The scope of a variable is defined by where in the algorithm the variable can be referred to. So far all the variables used in all the example algorithms have a local scope which means they can only be referred to in the sub module in which they are used. A global variable is one which is accessible anywhere in the algorithm i.e. it has a global scope. The use of global variables is not recommended because they increase the coupling between sub modules. A high degree of coupling between two sub modules means that a change in one sub module will mean the other sub module will need modification. A low degree of coupling means that a change in one sub module is not likely to require a change in any other sub modules. For the moment we will only make use of local variables. Will will revisit variable scope in the object oriented chapters.

Putting It All Together with an Example

As always, an example helps us understand all of the concepts involved. Suppose we needed an algorithm which would input a date as a eight digit integer where the first two digits, moving left to right, represent the day, the second two digits represent the month and the last four digits represent the year. Our algorithm will not validate the date (because ,at this point, we do not know enough to be able to validate).

We start by breaking the problem down into sub problems:

- Input the integer for date
- extract day, month and year
- output the date in the for day/month/year.

Hence our first draft of an algorithm is:

```
INPUT dateInteger
calculate day
calculate month
calculate year
OUTPUT day, "/", month, "/", year
```

What is now required is sub modules for calculating day, month and year. Before we start getting stuck into that we grab a piece of paper and work out the required mathematics. What we need is a way of chopping the number up into parts. Integer division is a convenient way of achieving this. A little mucking about yields the following expressions:

- day = date DIV 100000
- year = date MOD 1000

  where MOD is an operation which yields the remainder of integer division (e.g. 5 MOD 2 = 1) and
  DIV represents integer division (e.g. 5 DIV 2 = 2).

month is a bit trickier but is basically built by extending the logic from the first two expressions:

- temp = date MOD 1000000
- month = temp DIV 10000

Armed with this information we attempt the next draft of the algorithm:

```
INPUT dateInteger
CALL calculateDay
CALL calculateMonth
CALL calculateYear
```

After developing the algorithms for each sub module, we then determine the EXPORT and IMPORT and update our main algorithm to yield:

```
Main Sub Module:
ALGORITHM
INPUT dateInteger
CALL calculateDay<--dateInteger
Call calculateMonth<--dateInteger
Call calculateYear<--dateInteger
OUTPUT day, "/", month, "/", year

calculateDay Sub Module:
IMPORT inDate
EXPORT day
ALGORITHM
day = inDate DIV 1000000

calculateMonth Sub Module:
IMPORT inDate
EXPORT month
ALGORITHM
month = inDate MOD 1000000
month = month DIV 10000

calculateYear Sub Module:
IMPORT inDate
EXPORT year
ALGORITHM
year = inDate MOD 10000
```

Again I remind you that the algorithms discussed so far are relatively trivial and are simply be used to demonstrate the process. Note the order in which our algorithm was created. First came our first draft of the main algorithm. The next step was to put the algorithm aside and work out the required math. Finally came the second draft or the main algorithm and the three sub modules.

However we are not finished. The last thing that needs doing is to test each sub module individually and then the whole algorithm. In performing the tests

we know our algorithm will not perform correctly with invalid data. It is still important that we test it on invalid data and record the results. Remember know the behaviour (good or bad) of an algorithm is as important as the algorithm itself.

In terms of test data, in this case, the data will be the same for all of the modules:
negative

- < 1000
- < 100000
- > 99999999
- An assortment of integers in a valid range.

Note we are not testing the validity of the actual date in formation here because the overall purpose of the algorithm is to *extract* the date components. What could you build into the algorithm if you wished to allow for invalid dates? You would need some mechanism for selection or repetition. For example, if the date was valid we would require one set of algorithm steps to be followed and if it was invalid then we would require a different set of steps to be followed. The next two chapters discuss control structures. These control structures give us the tools we need to deal with selecting between alternate sets of pseudo code steps or to repeat a set of steps until some goal had been achieved.

Conclusion

Well, now you know all about algorithm design and sub modules. We see that pseudo code is the most effective mechanism for expressing algorithms and that sub modules can be combined with step wise refinement to produce a *modular* algorithm where each sub module *encapsulates* the solution to one section of the overall problem. Testing sub modules provides us with knowledge about the behaviour of each sub module. This knowledge provides us with confidence that, if each sub module has been thoroughly tested, then problems which can occur when testing the overall algorithm are minimised. A bit of practice should make you realise that modular algorithm design is a lot harder than it might sound! Take heart, it comes with lots of practice. Remember that if this process was easy then everyone could design their own software and therefore your career would cease to exist!

So far the algorithms that we have been able to design have been very limited in scope and complexity. This is because we need mechanisms for making selections about which statements to execute (i.e. if the door is locked then unlock it before entering, otherwise just enter). Also we need mechanisms for repeating sets of steps until we have achieved the desired result (e.g. keep sanding until the rod fits the hole). The standard mechanisms for doing this are called control structures. Chapters Four and Five discuss them.

Chapter Four

"Oh, that was easy says man, and for an encore goes on to prove that black is white and gets himself killed on the next zebra crossing"
Douglas Adams, The Hitch-Hiker's Guide to the Galaxy", 1979.

Selection Control Structures

Introduction

All of the algorithms discussed so far, have had one path of execution from start to end. The solution of many problems require:

- A mechanism for selecting which steps to execute based upon some criteria. Selection control structures provide this mechanism.
- A mechanism to repeat groups of steps until some criteria has been met. Looping control structures provide this mechanism.

Looping control structures will be discussed in the next chapter. In this chapter we will focus upon selection control structures.

Selection control structures allow an algorithm to choose between alternate paths of execution through the algorithm. As humans we have been doing this, without thinking about it, all our lives. There really is only one type of selection control structure but it does have a few sub categories (see later). The basic structure is called IF-THEN-ELSE and follows the format below:

```
IF something is true THEN
   algorithm steps to do if true
ELSE
   algorithm steps to do if false
ENDIF
```

For example:

```
INPUT x
INPUT y
IF x > y THEN
   max = x
ELSE
   max = y
ENDIF
```

If-THEN statements are relatively straight forward because we have used

them all our lives when explaining instructions to humans. The algorithm below is for a human, not a computer, to follow and uses IF-THEN to try to turn on an electrical appliance.

```
Switch power button to ON position
IF the power light does not come on THEN
   check the wall plug switch setting
   IF the wall plug is switched OFF THEN
      switch the wall plug ON
   ELSE
      check that the power cable is plugged in
      IF the power cable is not plugged in THEN
         plug the power cable in
      ELSE
         take the appliance back to the store
      ENDIF
   ENDIF
ENDIF
```

Most people, including people who have never heard of algorithms or pseudo code, would be able to follow those instructions without any trouble. In other words the IF-THEN-ELSE control structure is intuitive and, although we haven't got that formal about it, we have been using them to explain instructions to others all our lives. Notice the use of IF-THEN inside other IF-THEN structures. This is called *nesting* control structures.

The ELSE part of an IF-THEN control structure is optional. You put one in only if it makes sense to do so. The algorithm below is an example of an IF-THEN without an ELSE statement.

```
INPUT markOne, markTwo
diff = markOne - markTwo
IF diff < 0 THEN
   diff = diff x -1
ENDIF
```

Lets look at a simple example. Suppose we wanted to input a date and the output whether or not the date was valid. An initial algorithm might look like:

```
INPUT day, month, year
IF day, month and year are valid THEN
   OUTPUT day, "/", month, "/", year, " is a valid date"
ELSE
   OUTPUT day,"/",month,"/",year," is not a valid date"
ENDIF
```

The next step would be to come up with the criteria for valid and invalid so we can express the boolean expression in the IF-THEN statement more precisely. After a bit of thought we realise that we cannot come up with a simple boolean expression for validating the date. Month and year are easy but day is dependant on the specific month and if the month is 2 then we need to determine whether or not year is a leap year.

Using Boolean Sub Modules:

When the validation of data cannot be achieved via a simple boolean expression a sub module should be used. The algorithm for validation can be placed in the sub module and the call to the sub module placed in the IF THEN statement:

```
INPUT day, month, year
IF validateDate<--day, month, year THEN
   OUTPUT day, "/", month, "/", year, " is a valid date"
ELSE
   OUTPUT day,"/",month,"/",year," is not a valid date"
ENDIF
```

A full development of the date validation algorithm is included as an example at the end of this chapter.

Logical AND and Nesting IF-THEN ELSE

Any boolean expression involving the AND operator can be separated into two nested IF-THEN statements. i.e.:

```
IF booleanExpressionA AND booleanExpressionB THEN
   do something
ENDIF
```

Can be converted into:

```
IF booleanExpressionA THEN
   IF booleanExpressionB THEN
      do something
   ENDIF
ENDIF
```

The question is really when you you nest and when would you not nest? The answer is that most of the time it is a subjective choice. Not nesting might result in slightly faster machine code once the program is translated into a programming language and compiled but that may not be a valid reason. There are two circumstances when this is appropriate:

1. When the two boolean expressions are complex and separating them in this way makes the algorithm easier to understand.
2. When the first expression being false will cause the second expression to be impossible to evaluate.

For example:

```
IF numPeople ≠ 0 AND totalHeights/numPeople < 5.5 THEN
```

If numPeople is zero then the calculation of totalheights/numPeople will cause a divide by zero error. However if we use nested IF-THEN instead then the

problem does not arise:

```
IF numPeople ≠ 0 THEN
   IF totalHeights/numPeople < 5.5 THEN
```

Many modern programming languages employ a technique known as short circuit evaluation when evaluating boolean expressions. The idea is based on the principle that, under some circumstances the entire boolean expression does not need to be evaluated to know the result:

`false AND anything` will always evaluate to false
`true OR anything` will always evaluate to true.

In the previous example, short circuit evaluation would prevent the divide by zero from occurring because if numPeople was equal to zero the the first part of the boolean expression is false and so the second half does not need to be evaluated. However not all languages employ short circuit evaluation and, in some languages, it can be turned off. This means that nesting the IF-THEN statements is the better approach because it will work for all programming languages.

Linear IF -THEN Control Structure

A linear IF-THEN control structure is where:
- A series of nested IF-THEN structures occur.
- The Boolean expression inside each one is related to the same data.

Under these situations the use of nested IF-THEN is required but the nesting process yields rather long winded pseudo code. Formatting becomes an issue If we do the right thing and indent the code each time we nest the control structure. The pseudo code below illustrates the problem. Algorithmically what we are attempted to do is fairly simple but the pseudo code that we generate is clumsy.

```
IF day = 1 THEN
   dayString = "Monday"
ELSE
   IF day = 2 THEN
      dayString = "Tuesday"
   ELSE
      IF day = 3 THEN
         dayString = "Wednesday"
      ELSE
         IF day = 4 THEN
            dayString = "Thursday"
         ELSE
            IF day = 5 THEN
               dayString = "Friday"
            ELSE
               IF day = 6 THEN
                  dayString = "Saturday"
```

```
               ELSE
                  IF day = 7 THEN
                     dayString = "Sunday"
                  ELSE
                     dayString = "Not a day"
                  ENDIF
               ENDIF
            ENDIF
         ENDIF
      ENDIF
   ENDIF
ENDIF
```

An alternate approach would be to recognise that all the boolean expressions are related because they are all about alternative values of the same variable and reformat our nested IF statement to recognise this fact. One way would be to use separate IF-THEN statements. The result would look something like:

```
IF day = 1 THEN
   dayString = "Monday"
ENDIF
IF day = 2 THEN
   dayString = "Tuesday"
ENDIF
IF day = 3 THEN
   dayString = "Wednesday"
ENDIF
IF day = 4 THEN
   dayString = "Thursday"
ENDIF
IF day = 5 THEN
   dayString = "Friday"
ENDIF
IF day = 6 THEN
   dayString = "Saturday"
ENDIF
IF day = 7 THEN
   dayString = "Sunday"
ENDIF
IF day < 1 OR day > 7 THEN
   dayString = "Not a day"
ENDIF
```

Note that, not only is this solution is still clumsy, it is also inefficient. Nesting the IF-THEN statements ensures that only the minimal number of boolean expressions get evaluated. Not nesting them (as above) means that every boolean expression is evaluated every time. Also there has to be an extra boolean expression because the last clause is not nested under the others anymore, so it cannot simply be an ELSE.

The Linear IF-THEN resolves all of these issues. If you look at the linear IF version of the previous example below, you will see that it has the same flow

of control as the nested IF-THEN version but is formatted in a much more intuitive and concise manner. Remember that one of the goals of pseudo code is to provide clarity and avoid lots of trivial syntax.

```
IF day = 1 THEN
   dayString = "Monday"
ELSE IF day = 2 THEN
   dayString = "Tuesday"
ELSE IF day = 3 THEN
   dayString = "Wednesday"
ELSE IF day = 4 THEN
   dayString = "Thursday"
ELSE IF day = 5 THEN
   dayString = "Friday"
ELSE IF day = 6 THEN
   dayString = "Saturday"
ELSE IF day = 7 THEN
   dayString = "Sunday"
ELSE
   dayString = "Not a day"
ENDIF
```

It is important to realise that all we are doing is reformatting a set of nested IF-THEN statements where the Boolean expression in all of them is related to the same data. Note that in the above example, all of the boolean expressions were making equality comparisons. This does not have to be the case, any kind of boolean expression is fine as long as each expressions relates to the same information. For example, suppose we wished to check if an integer was positive, negative or zero:

```
IF number < 0 THEN
   OUTPUT "Number is negative"
ELSEIF number > 0 THEN
   OUTPUT "Number is positive"
ELSE
   OUTPUT "Number is zero."
ENDIF
```

In the above example, all of the boolean expressions were concerned with the value of the variable *number* but the expressions were range checks not equality checks.

The CASE Control Structure

In the situation where all of the checks are equality checks, there is an alternative to the linear IF statement, known as the CASE statement.. The CASE control structure is used where:

- There are a finite number of discrete possible values for an expression.
- A different set of actions are required depending upon these different possible values of the expression.

A case statement consists of:

- A specification of the expression being evaluated.
- A set of value: statement sets.

```
CASE expression
     value1: statement(s)
     value2: statement(s)
     ...
     value n: statement(s)
     otherwise: statement(s)
ENDCASE
```

For example, the linear IF statement used in the previous day of the week example would look like:

```
CASE day
     1: dayString = "Sunday"
     2: dayString = "Monday"
     3: dayString = "Tuesday"
     4: dayString = "Wednesday"
     5: dayString = "Thursday"
     6: dayString = "Friday"
     7: dayString = "Saturday"
     OTHERWISE: dayString = "Not a Day"
ENDCASE
```

Note that it is possible to list multiple values in one case clause:

```
CASE day
     2, 3, 4, 5, 6: OUTPUT "Week Day"
     1, 7:          OUTPUT "Weekend"
     OTHERWISE: OUTPUT "Invalid Day"
ENDCASE
```

Case is not an indispensable control structure, it is simply convenient to use sometimes. Remember that any CASE statement can be replaced by a linear IF statement. We have already seen what the first example looks like as a linear If statement. The second one would be:

```
IF day = 2 OR 3 OR 4 OR 5 OR 6 THEN
   OUTPUT "Week Day"
ELSE IF day = 1 OR 7 THEN
   OUTPUT "Weekend"
ELSE
   OUTPUT "Invalid Day"
ENDIF
```

Note that the first IF statement is more verbose than the CASE statement and so the CASE statement would be the preferred option whereas the second IF versus the second CASE statement is a subjective choice (i.e. either is as good as the other). Given that any CASE statement can be rewritten as a linear IF statement, is the reverse true?

In the above examples there was only one output step for each clause. In fact there can be as many steps as are required. Often the step(s) are sub module calls:

```
CASE day
     2, 3, 4, 5, 6: OUTPUT "Week Day"
                    CALL processWeekDay<--day
     1, 7:          OUTPUT "Weekend"
                    CALL processWeekend<--day
     OTHERWISE: OUTPUT "Invalid Day"
ENDCASE
```

Pre and Post Conditions

A pre-condition is a statement of fact which is true prior to the execution of a set of pseudo code statements. A post condition is a statement of fact which is true after the execution of a set of pseudo code steps. For example:

```
INPUT day

Pre-Condition: day could be any integer value
IF day < 1 THEN
   day = 1
ELSE IF day > 7 THEN
   day = 7
ENDIF
Post-Condition: 1 <= day <= 7
```

Assertion Statements

An assertion statement is a pre/post condition statement which is true at the point in the algorithm that it is encountered. If an assertion statement turns out to be false then that indicates an error in the algorithm and hence assertion statements act as validation points in any attempt to detect the errors in an algorithm.

An assertion statement can be placed anywhere in an algorithm but the most useful locations are:

- Before/after selection control structures.
- Before/after looping control structures (see next chapter).
- Before/after sub module calls.

Using Pre/Post Conditions and Assertion Statements to Find Errors in Algorithms.

When an algorithm does not work, the first step is to attempt to localise where the algorithm is going astray. The assertion statements provide a frame work for this. The algorithm is stepped through by hand and each assertion is tested when it is encountered. If an assertion is found to be false then we know there must be an error in the algorithm steps related to the assertion. Consider the simple algorithm below:

```
INPUT day
IF day > 1 THEN
   day = 1
ELSE IF day < 7 THEN
   day = 7
ENDIF
Assertion: 1 <= day <= 7
CASE day
     1: OUTPUT "Sunday"
     2: OUTPUT "Monday"
     3: OUTPUT "Tuesday"
     4: OUTPUT "Wednesday"
     5: OUTPUT "Thursday"
     6: OUTPUT "Friday"
     7: OUTPUT "Saturday"
ENDCASE
Assertion: day of the week has been output
```

The OTHERWISE clause in the Case statement is not required because the assertion just prior to it states that day is in the range 1 to 7 inclusive. However, there is an error in the algorithm which will mean that, in some cases the first assertion will not hold true. This, in turn, will cause the second assertion to be false as well. Hand testing of the algorithm would reveal the error which occurs when the input is less than 1. The greater than operator in the first IF statement should be a less than operator. This would have come to light when the an input less than one caused the first assertion to be false.

In other words the first step is to find the error(s) and once found then think about how they might be corrected.

Trick of the Trade: Eliminating dangling ELSE clauses

One issue that often arises is when an algorithm has a series of nested IF statements where:

- Each IF has an ELSE clause.
- The action to be taken in all the ELSE clauses is exactly the same.

The pseudo code below provides an example of this situation:

```
INPUT x, y
IF 0.0 <= x <= 1.0 THEN
   IF 0.0 <= y <= 1.0 THEN
      dist = √x2 + y2
   ELSE
      dist = -1.0
   ENDIF
ELSE
   dist = -1.0
ENDIF
```

The ELSE clauses can be eliminated by placing the step(s) in the ELSE

clauses just before the IF statement and removing the ELSE clauses:

```
INPUT x, y
dist = -1.0
IF 0.0 <= x <= 1.0 THEN
   IF 0.0 <= y <= 1.0 THEN
      dist = √x2 + y2
   ENDIF
ENDIF
```

This situation arises more often than you might think so its a good trick to keep in mind.

Example Algorithm One

Okay, now we know about selection control structures, lets put everything we have learned so far into practice with an example.

An algorithm is required which will:

- Input a time of day, expressed as a military time (i.e. 0000 to 2359 hours).
- The military time needs to be converted to an am/pm time.
- The am/pm time then needs to be output to the user.
- If the input time is not valid then an error message should be output instead.

As before, the starting point is to understand the problem in more detail. There are only two issues which require a look at:

1. The criteria for valid/invalid time.
2. The criteria for assigning am, pm or midday.

A valid military time has:

- hours in the range 0 to 23 and
- minutes in the range 0 to 59

This means that simply checking that the time is between 0000 and 2359 is not enough because the hours value could be in range but the minutes value could be greater than 59 (e.g. 1675). This means that the algorithm will have to separate out hours and minutes into two variables and range check each one in order to validate the time. The table below shows the required value for am/pm description required based upon the value for the input military time.

Time Range	amPm
0000 military time 1159	"am"
1201 military time 2359	"pm"
military time = 1200	"midday

Notice the way we play with different facets of the problem before we start designing the algorithm. This is an important step because it gives us a clearer understanding of what has to happen. The next step is to come up with a main algorithm which will use sub modules to accomplish the sub tasks:

```
MAIN
INPUT militaryTime
IF militaryTime is valid THEN
   convert and output militaryTime
ELSE
   OUTPUT "Military time is invalid"
ENDIF
```

Revising the algorithm by replacing the ambiguous steps with sub module calls yields:

```
MAIN
INPUT militaryTime
IF validateMilitaryTime THEN
   processMilitaryTime
ELSE
   OUTPUT "Military time is invalid"
ENDIF
```

The next step would be to develop the two sub modules referred to in MAIN and then come back and add the IMPORT/EXPORT information to the sub module calls. One observation that can be made here is that both the validation and the conversion sub modules really want the hours and minutes values of the military time in separate variables. We could do the extraction of hours and minutes in each sub module but that means we have redundant steps. The sensible thing to do is extract hours and minutes in MAIN and then feed them to each sub module. Hence the main algorithm becomes:

```
MAIN
INPUT militaryTime
militaryHours = militaryTime DIV 100
militaryMins  = militaryTime MOD 100
IF validateMilitaryTime<--militaryHours,militaryMins THEN
   processMilitaryTime<--militaryHours, militaryMins
ELSE
   OUTPUT "Military time is invalid"
ENDIF
```

Note that, in this case, we are assuming we know the IMPORT/EXPORT for the sub modules before we have developed them. In most cases we won't have a good idea and so cannot determine the information until after the sub module has been fully developed. As has been previously stated, algorithm design is an artistic process, so there are not many rules that are going to be absolutely true all the time. In this case we have a pretty good idea and as long as we are prepared to come back and change the IMPORT/EXPORT, if required, afterwards then the assumption we are making about the

IMPORT/EXPORT is fine.

The next step is to refine each sub module. We already know that the validation involves range checking the hours and minutes values so the sub module would be:

```
SUB Module validateMilitaryTime
IMPORT hours, minutes
EXPORT timeIsValid

IF 0 <= hours <= 23 THEN
   IF 0 <= minutes <= 59 THEN
      timeIsValid = true
   ELSE
      timeIsValid = false
   ENDIF
ELSE
   timeIsValid = false
ENDIF
```

The above algorithm came out of dealing with hours and minutes separately. It will work but involves a nested IF-THEN when and the AND operator could eliminate the need for the inner IF-THEN. Hence the sub module becomes:

```
SUB Module validateMilitaryTime
IMPORT hours, minutes
EXPORT timeIsValid

IF 0 <= hours <= 23 AND 0 <= minutes <= 59 THEN
   timeIsValid = true
ELSE
   timeIsValid = false
ENDIF
```

Notice that we didn't stop refining the algorithm at the first draft, even though the first version would work. This happens often with selection control structures. The intuitive first draft isn't the most optimal algorithm. Hindsight is used to analyse the resulting algorithm and see if it can be improved. The three most common improvements are:

1. Eliminating nested IF statements by moving the inner boolean expression to the outer IF statement and adding it using an AND operator.
2. Reversing the logic so that the ELSE clause becomes the IF clause and vice versa. For reasons I cannot explain most peoples' first attempt has the boolean logic and the If-ELSE the wrong way around.
3. Completely eliminating the IF-THEN statement in cases where the IF and ELSE clause are assigning true and false respectively to a boolean variable.

We can further refine our algorithm by employing improvement 3 from above. Look at the IF-THEN statement carefully. If the boolean expression in the IF statement is true then we assign true to the boolean variable. If the boolean

expression in the IF statement is false, then we assign false to the boolean variable. Why not simply assign the result of the boolean expression to the boolean variable? The end result will be exactly the same. Hence the algorithm becomes:

```
Sub Module validateMilitaryTime
IMPORT hours, minutes
EXPORT timeIsValid

timeIsValid = 0 <= hours <= 23 AND 0 <= minutes <= 59
```

Finally we need to develop the second sub module. There are three tasks to be accomplished:

1. Convert the hours from 0 to 23 to 0 to 12.
2. Determine the time description (i.e. "am", "pm" or "midday").
3. Output the time an am/pm format.

Hence the first draft of the algorithm looks like:

```
Sub Module  processMilitaryTime
IMPORT hours, minutes
EXPORT None

calculate amPmHours
calculate timeDescription
OUTPUT amPmHours, "." minutes, timeDescription
```

The output statement is fine. The other two steps each require a sub module. Again, because of the simplicity of the situation we can guess at the IMPORT/EXPORT:

```
Sub Module  processMilitaryTime
IMPORT hours, minutes
EXPORT None

Assertion: 0 <= hours <= 23
amPmHours = calcAmPmHours<--hours
Assertion: 0 <= amPmHours <= 12
timeDescription = calcTimeDescription<--hours
OUTPUT amPmHours, "." minutes, timeDescription
```

Notice that the algorithm depends upon the assertion being true. It won't work correctly if the assertion is not upheld. The two sub modules are fairly straight forward:

```
Sub Module calcAmPmHours
IMPORT milHours
EXPORT amPmHours
Assertion: 0 <= milHours <= 23
IF milHours = 0 THEN
   amPmHours = 12
ELSE IF 12 < milHours THEN
   amPmHours = milHours - 12
ELSE
   amPmHours = milHours
ENDIF

Sub Module calcTimeDescription
IMPORT milHours
EXPORT timeDesc
Assertion: 0 <= milHours <= 23
IF milHours < 12 THEN
   timeDesc = "am"
ELSE IF milHours = 12 THEN
   timeDesc = "midday"
ELSE
   timeDesc = "pm"
ENDIF
```

Notice in both sub modules, the algorithms will only work in all cases if the assertion stated holds. If the IMPORT hours value is not in the range specified in the assertion statement then its possible that both algorithms will not work as required. Hence if testing of the algorithm revealed invalid output being made, these assertion statements would need to be tested. If they were found to not hold true in all cases then the algorithm must be modified to ensure that they will hold true in all cases.

Example Algorithm Two

Earlier in the chapter we saw a MAIN algorithm which inputs a date and then outputs whether or not the date is valid:

```
INPUT day, month, year
IF validateDate<--day, month, year THEN
   OUTPUT day, "/", month, "/", year, " is a valid date"
ELSE
   OUTPUT day,"/",month,"/",year," is not a valid date"
ENDIF
```

What remains to be done with this algorithm is to develop the validation sub module. A first draft looks like:

```
Sub Module validateDate
IMPORT day, month, year
EXPORT dateIsValid

dateIsValid = false
IF day valid THEN
   IF month valid THEN
      IF year valid THEN
         dateIsValid = true
      ENDIF
   ENDIF
ENDIF
```

Note that before we continue we must put our algorithm aside and do a bit of analysis of what opens and shuts in a date. Validating month is easy. Year is a bit ambiguous (i.e. do we allow for AD/BC or are we just interested in dates in the current century?). Let's assume, for the purposes of this algorithm, that we are just interested in the 21st century (i.e. 2000 <= year <= 2099). The validation of day is also okay as long as we know how many days are in the specified month. If we use a sub module for calculating the number of days in the month then the next version of our algorithm is:

```
Sub Module validateDate
IMPORT day, month, year
EXPORT dateIsValid

dateIsValid = false
IF 1 <= day <= calcdaysInMonth<--month THEN
   IF 1 <= month <= 12 THEN
      IF 2000 <= year <= 2099 THEN
         dateIsValid = true
      ENDIF
   ENDIF
ENDIF
```

Note that, as with the military time example, we think we know what the IMPORT to calcDaysInMonth will be. In this case we won't be exactly correct but we can return and modify our pseudo code once we have finished the rest of the algorithm. The calcDaysInMonth sub module makes use of a CASE statement:

```
Sub Module calcdaysInMonth
IMPORT month
EXPORT days
CASE month
   1, 3, 5, 7, 8, 10, 12: days = 31
   4, 6, 9, 11: days = 30
   2: IF year is a leap year THEN
         days = 29
      ELSE
         days = 28
   OTHERWISE: days = -1
ENDCASE
```

It is at this point that we realise that this sub module needs to know the year as well as the month. We also need to determine if the year is a leap year so we will use a sub module for that:

```
Sub Module calcDaysInMonth
IMPORT month, year
EXPORT days

CASE month
   1, 3, 5, 7, 8, 10, 12: days = 31
   4, 6, 9, 11: days = 30
   2: IF determineIfLeapYear<--year THEN
         days = 29
      ELSE
         days = 28
   OTHERWISE: days = -1
ENDCASE
```

Notice that the IMPORT to calcDaysInMonth was expected to just be month but now we realise that it has to include year as well. To develop the determineIfLeapYear submodule we need to determine the criteria for a leap year. A bit of research (okay a bit of Googling) reveals that a year is a leap year if:

- it is a multiple of 4.
- if the year is a century it must be a multiple of 400.

Hence the required boolean expression has two parts. The first dealing with it being a multiple of 4 and the second dealing with the century criteria. This means a rough draft of the required boolean expression would be:

(year is not a century AND year is a multiple of 4) OR year is a multiple of 400

Refinement of the boolean expression above yields:

(year MOD 100 ≠ 0 AND year MOD 4 = 0) OR year MOD 400 = 0All that is need now is to place the boolean expression in the sub module:

```
Sub Module determineIfLeapYear
IMPORT year
EXPORT isALeapyear

isALeapyear = ( year MOD 100   0 AND year MOD 4 = 0 ) OR
              ( year MOD 400 = 0)
```

All that now remains is to revise our validateDate sub module to allow for the IMPORT of year into the calcDaysInMonth sub module:

```
Sub Module validateDate
IMPORT day, month, year
EXPORT dateIsValid

dateIsValid = false
IF 1 <= day <= calcdaysInMonth<--month, year THEN
   IF 1 <= month <= 12 THEN
      IF 2000 <= year <= 2099 THEN
         dateIsValid = true
      ENDIF
   ENDIF
ENDIF
```

Conclusion

Both of the example algorithms would not have been possible without selection control structures. The ability to choose alternate sets of algorithm steps is an essential algorithm development tool. Note also that everything we have discussed prior to this chapter is used in earnest in the algorithms presented here. In fact, now that we have added control structures to the mix, step wise refinement becomes an essential tool because our sub module algorithms are more complex. However, our algorithms are still limited. For example, in the military time conversion algorithm, suppose we wanted to convert more than one military time? We need control structures that will allow our algorithms to repeat sections as many times as is required. Iteration or looping control structures are used for this purpose and are discussed in the next chapter.

Chapter Five

"There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is another theory which states that this has already happened."
Douglas Adams, The Restaurant at the End of the Universe, 1980.

Iteration Control Structures

Introduction

In the previous chapter we examined control structures aimed at providing a means of selecting different execution paths through our algorithms. Another important type of control structure provides a means of repeating sections of algorithms until some desired goal has been reached. As human beings performing task in the real world we often repeat actions to achieve a goal. A simple example would be that of repeatedly striking a nail with a hammer until the nail has been driven fully into the surface it was being nailed into. These control structures are called looping or iteration control structures. There are two main types of looping control structures. The difference between them is how the loop decides when to stop repeating.

Each looping control structure must specify:

- What type of loop it is (see below for loop types).
- What its termination condition is. In other words what stops the loop.
- What statements are executed each time through the loop.

The termination condition of a loop is a Boolean expression which depending upon whether or not it is true or false, determines whether or not the loop repeats again or terminates. The two types of looping control structures differ in when the stopping condition is checked:

- The WHILE loop checks at the start of the loop.
- The DO-WHILE loop checks at the end of the loop.

The While Loop

A While consists of:

- The statement that it is a While loop
- A Boolean expression which is placed at the top of the loop.
- An indication of which statements are inside the loop.

The Boolean expression is checked at the start of each loop. If the expression is true then the loop is repeated once. If the expression is false then control moves to the next statement after the loop. If the Boolean expression is false the very first time it is checked then it is possible that the statements within the loop are never executed. Hence a While loop will repeat zero or more times.

The template below illustrates how a While loop could be expressed in pseudo code:

```
WHILE  boolean expression DO
    statements inside the loop
ENDWHILE
```

For example:

```
INPUT number
WHILE number < 0 DO
   OUTPUT "Your number should not be negative"
   INPUT number
ENDWHILE
```

In the above example, if the user entered a negative number then the Boolean expression would evaluate to true and the loop would be entered. The first time the user entered a non-negative number then the next time the Boolean expression was checked it would then evaluate to false and the loop would be terminated. Notice that if the very first number that the user entered was non-negative then the statements inside the loop would never be executed because the first time the Boolean expression was checked it would evaluate to false and the loop would be terminated before it started.

The DO-WHILE Loop

The DO-WHILE loop follows the same basic principle as the While loop except that the Boolean expression is placed at the bottom of the loop. This means that the statements within the loop are executed at least once because the Boolean expression won't be checked until execution reaches the bottom of the loop.

The template below illustrates how a Do-While loop could be expressed in pseudo code:

```
DO
    statements inside the loop
WHILE  boolean expression
```

For example:

```
DO
   INPUT number
   IF number < 0 THEN
      OUTPUT "Your number should not be negative"
   ENDIF
WHILE number < 0
```

Notice that, no matter what the user enters as input, the statements inside the loop are executed at least once because they have to be in order to get to the Boolean expression at the bottom of the loop. Hence DO-WHILE loops execute at least once.

An alternative way of expressing DO-WHILE loops is REPEAT UNTIL. A repeat until loop will continue to loop until the Boolean expression becomes true whereas a DO-WHILE loop will continue to loop until the Boolean expression becomes false.

For example:

```
REPEAT
   INPUT number
   IF number < 0 THEN
      OUTPUT "Your number should not be negative"
   ENDIF
UNTIL number >= 0
```

REPEAT-UNTIL and DO_WHILE are two versions of the same kind of looping structure because they both loop at least once. Which one you decide to use is simply a matter of personal preference. However you should be consistent don't alternate between the two, use one or the other all the time. These days DO-WHILE is more prevalent in programming languages than REPEAT-UNTIL.

The FOR Loop.

There is a third looping control structure which is really just a WHILE loop in disguise. Many algorithms have loops which use a variable to count from some starting value to an end value. A While loop can be used to achieve this:

```
number = 1
WHILE number < 100 DO
   OUTPUT number
   number = number + 1
ENDWHILE
```

It turns out that the need to count in a loop is very common and so a specialised control structure was introduced to make this easier. The above pseudo code initialises number to 1, checks to see if number has reached its final value and increments number by one each time through the loop. A FOR loop control structure has a counting variable (known as the loop index) which it sets to a starting value, increments by a specified value each time through the loop and stops at an ending value.

The template below illustrates how a FOR loop should be expressed in pseudo code:

```
FOR loopIndex = startVal TO endVal INC BY incValue
   statements inside loop
ENDFOR
```

For example:

```
FOR number = 1 TO 100 INC BY 1
   OUTPUT number
ENDFOR
```

The initialisation and incrementing of the loop index (number in the above example) is completely under the control of the FOR loop. This eases the burden on the part of the algorithm designer because they no longer have to bother with those issues. Because a starting, stopping and increment must be specified in a FOR loop, the number of times the loop will repeat is always known. Note that the loop index does not have to count upwards, a negative increment will cause the loop counter to count downwards. For example:

```
FOR number = 100 TO 1 INC BY -1
   OUTPUT number
ENDFOR
```

Also the increment can be any integer value. it is often 1 but does not have to be. For example the loop below will output all of the even numbers between 1 and 100:

```
FOR number = 2 TO 100 INC BY 2
   OUTPUT number
ENDFOR
```

This means we can always calculate the exact number of times a FOR loop will repeat using the formula:

$$\left|\frac{(endValue - startValue)}{increment}\right| + 1$$

It is important that you keep in mind that a FOR loop is a specialised form of WHILE loop with the following properties:

- The FOR loop index must be an integer.
- The start, end and increment values must be specified and must be integer.
- The FOR loop index is not to be modified inside the FOR loop.
- The FOR loop index is not to be referred to outside of the FOR loop.

Sometimes a FOR loop might be close to what is needed but the above properties cannot be preserved. In those cases we would fall back to using a normal WHILE loop instead.

For example, suppose we wanted to loop 10 times, each time inputting a number. However at any point if the user enters a negative number we want the loop to terminate. This means that we are no longer sure how many times we want the loop to repeat so we have to use a WHILE loop instead:

```
INPUT number
count = 1
WHILE count <= 10 AND number >= 0 DO
   sum = sum + number
   INPUT number
   count = count + 1
ENDWHILE
```

In then above example, the steps in bold show the sections of the algorithm which is achieving the FOR loop functionality. Because the termination condition involved more than the final value of the FOR loop index, a FOR loop cannot be used. Many old time C,/C++ code cutters violate these properties and basically use a FOR loop for any kind of loop. They get away with this because of the incredibly crude implementation provided by C for a FOR loop. They are not to be considered a role model. Rather they are prime candidates for early retirement because the rest of their code is often just as atrocious and unmaintainable.

In other cases we can still use a FOR loop but have to ensure the above properties are preserved. For example suppose we wanted to count from 0.0 to 1.0 in increments of 0.1. We cannot cycle a FOR loop index from 0.0 to 1.0 because the FOR loop index must be an integer. We can still use a FOR loop by setting the loop to iterate 10 times and using a different variable which we increment from 0.0 to 1.0:

```
num = 0.0
FOR count = 1 TO 10 INC BY 1
   OUTPUT num
   num = num + 0.1
ENDFOR
```

Alternatively we could use the FOR loop index and divide its value by 10 to get the desired output:

```
FOR count = 1 TO 10 INC BY 1
   OUTPUT count/10
ENDFOR
```

Neither version is superior to the other and both are acceptable because they achieve the desired result while preserving the generic properties of a FOR loop.

Choosing the Most Appropriate Loop

Once the need for a loop has been determined, the next step is to decide which type of loop should be used. In simple cases the choice is driven by simply determining how many times the loop will repeat:

- Fixed number of times then use a FOR loop.
- Must run at least once then use a DO-WHILE loop
- Under some circumstances the loop may not execute at all (i.e. repeats zero or more times) then use a WHILE loop.

Under many situations though, the determination of how many times the statements inside the loop need to repeat is not a simple question. For Example consider the DO-WHILE loop below:

```
DO
   INPUT number
   IF number < 0 THEN
      OUTPUT “Number should be non-negative”
   ENDIF
WHILE number < 0
```

The INPUT statement has to happen at least once (i.e. DO-WHILE) but the OUTPUT statement happens zero or more times (i.e. WHILE). This is resolved by using and IF statement to prevent the OUTPUT statement happening the last time through the loop. The point is that, in this situation, the statements inside the loop need to repeat a different number of times. This means that either a DO-WHILE or a WHILE can be used because neither is a perfect fit. The WHILE loop version is below:

```
INPUT number
WHILE number < 0 DO
   OUTPUT “Number should be non-negative”
   INPUT number
ENDWHILE
```

The DO-WHILE loop used an IF statement to resolve the conflict whereas the WHILE loop simply repeats the steps that need to go one more time just before the loop. The DO-WHILE version has the same Boolean expression being evaluated twice, each time through the loop but the WHILE loop version has the same steps repeated. This means that under these circumstances the choices is subjective and must be made in the context of the required algorithm. In the above case the WHILE loop version is better because there is only one repeated step above the loop.

Infinite Loops

An infinite loop is a loop which will never terminate. FOR loops will never become infinite loops because the number of times a FOR loop repeats is predetermined. WHILE loops and DO_WHILE loops however have to be designed so that the boolean expression used to decide whether or not to repeat the loop, will at some point, become false. If not then the loop will never terminate and we have an infinite loop. The two major causes of infinite loops are:

- Forgetting to include logic in the body of the loop which will eventually cause the loop Boolean expression to become false.
- Making a mistake in the Boolean logic so that the expression will always be true.

```
count = 1
WHILE count < 10 DO
   OUTPUT count
ENDWHILE
```

In the above example, the body of the loop never modifies the value of count and the Boolean expression for the loop will only become false if count's value is changed to a value which is greater than or equal to 10. Hence this loop will never end.

```
count = 2
WHILE count   99
   OUTPUT count
   count = count + 2
ENDWHILE
```

In the above example count will never be equal to 99 because it will start at two and increment by two each time through the loop so count will never contain an odd number. Both of the above examples are somewhat contrived but the point is clear. Care must be taken to ensure that we do not accidentally come up with an infinite loop.

Pre/Post Condition Assertion Statements Revisited

The end of loops is an ideal place for a post condition assertion statement. The loop can only terminate when the Boolean expression used by the loop

becomes false. Hence using the negated Boolean expression in an assertion statement immediately following the loop is a good way of ensuring that the Boolean expression being used is valid and correct.

For example:

```
INPUT oddNumber
WHILE oddNumber MOD 2 = 0 DO
   OUTPUT "Number must be odd"
   INPUT oddNumber
ENDWHILE
ASSERTION: oddNumber contains an odd number.
```

Consider the pseudo code below:

```
INPUT number
WHILE number < 1 AND number > 10 DO
   OUTPUT "Number must be between 1 and 10"
   INPUT number
ENDWHILE
ASSERTION 1 <= number <= 10
```

The AND in the Boolean expression is wrong and should be an OR. Testing the assertion at the end of the loop would reveal that it is not holding and that means there is an error in the Boolean expression for the loop. In this way the assertion statement has helped discover the error in the Boolean expression for the loop.

Nesting Loops.

In Chapter Four we saw that selection control structures can be nested, one inside the other. The same is true for looping control structures. If we have one loop nested inside another loop then each repetition of the outside loop will involve the inside loop cycling from start to end. This means that the number of times the inside loop is repeated is calculated by multiplying the number of times the inside loop repeats in one cycle of the loop by the number of times the outside loop is repeated.

For Example:

```
totalSteps = 0
FOR outsideCounter = 1 TO 5 INC BY 1
   FOR insideCounter = 1 TO 5 INC BY 1
      totalSteps = totalSteps + 1
      OUTPUT “Outside = “ outsideCounter,
             “  Inside = “, insideCounter,
             “ Total Steps = “, totalSteps
   ENDFOR
ENDFOR
```

The inside loop in the above example loops 5 times on each cycle. The outside loop also loops 5 times. Hence the output statement gets executed 5 x 5 = 25 times. So the output produced by the above algorithm would look

like:

```
Outside = 1  Inside = 1 Total Steps = 1
Outside = 1  Inside = 2 Total Steps = 2
Outside = 1  Inside = 3 Total Steps = 3
Outside = 1  Inside = 4 Total Steps = 4
Outside = 1  Inside = 5 Total Steps = 5
Outside = 2  Inside = 1 Total Steps = 6
Outside = 2  Inside = 2 Total Steps = 7
Outside = 2  Inside = 3 Total Steps = 8
Outside = 2  Inside = 4 Total Steps = 9
Outside = 2  Inside = 5 Total Steps = 10
Outside = 3  Inside = 1 Total Steps = 11
Outside = 3  Inside = 2 Total Steps = 12
Outside = 3  Inside = 3 Total Steps = 13
Outside = 3  Inside = 4 Total Steps = 14
Outside = 3  Inside = 5 Total Steps = 15
Outside = 4  Inside = 1 Total Steps = 16
Outside = 4  Inside = 2 Total Steps = 17
Outside = 4  Inside = 3 Total Steps = 18
Outside = 4  Inside = 4 Total Steps = 19
Outside = 4  Inside = 5 Total Steps = 20
Outside = 5  Inside = 1 Total Steps = 21
Outside = 5  Inside = 2 Total Steps = 22
Outside = 5  Inside = 3 Total Steps = 23
Outside = 5  Inside = 4 Total Steps = 24
Outside = 5  Inside = 5 Total Steps = 25
```

Often times when developing algorithms, the desired algorithm will require nested loops. In these cases the problem can be subdivided into devising the outside loop and ignoring what has to occur within the loop and the focussing on what has to happen in one iteration of the outside loop. Often a good strategy is to use sub modules such that the body of the outside loop calls a sub module to perform the inside loop. For example suppose we required an algorithm which would output the 1 to 12 times tables. The problem can be sub divided into the loop which cycles from the one times table to the 12 times table and the sub problem which involves just outputting a single times table. The main algorithm would look like:

```
MAIN
FOR timesTable = 1 TO 12 INC By 1
   outputTimesTable<--timesTable
ENDFOR
```

The outputTimesTable sub module will deal with the task of outputting a single times table:

```
SUB MODULE outputTimesTable
IMPORT timesTable
EXPORT None

FOR number = 1 TO 12 INC BY 1
   product = number x timesTable
   OUTPUT number, " times ", timesTable,
          " = ", product
ENDFOR
```

The use of a sub module for the inner loop has provided a clear sub division of the problem which means the potential to be confused between the two loops is minimised. If we removed the sub module and put the entire algorithm in main we would have:

```
MAIN
FOR timesTable = 1 TO 12 INC By 1
   FOR number = 1 TO 12 INC BY 1
      product = number x timesTable
      OUTPUT number, " times ", timesTable,
             " = ", product
   ENDFOR
ENDFOR
```

The danger is the potential to get confused between the inner and outer loops. Admittedly in this simple example such confusion is not likely but in a more complex examples ti is very easy to get the loops confused. Remember also that the nesting of loops can go beyond two loops. As many loops as is required can be nested one inside the other:

```
numSteps = 0
FOR loop1 = 1 TO 10 INCBY 1
   FOR loop2 = 2 TO 24 INCBY 2
      FOR loop3 = 20 TO 50 INC BY 1
         FOR loop4 = 4 TO 24 INC BY 4
            FOR loop5 = -10 TO 100 INC BY 1
                numSteps = numSteps + 1
            ENDFOR
         ENDFOR
      ENDFOR'
   ENDFOR
ENDFOR
OUTPUT "Total times inner loop executed was ", numSteps
```

Care must be taken to ensure that the nesting of loops is absolutely necessary because of the increase in the number of times the inner most loop is executed as a function of the outer loops.

Example One.

Okay, so now we know about problem solving, step wise refinement, sub modules, selection and looping control structures let's put it all together with some examples.

Suppose we needed an algorithm which would calculate the total floor area of the rooms of a house. For the sake of this example, we will assume that all rooms are rectangular in shape. The basic steps are:

- Input the number of rooms
- For each room:
    - Input the room width.
    - Input the room length.
    - Calculate the room area
    - Add the room area to the total area for the house.

Also each time a number is to be input, it must be positive so a sub module which loops until the number input was positive should be used and called each time input was required. Hence main would consist of two sub module calls:

```
MAIN
numRooms =
   inputNumber <--”Please enter then number of rooms”
totalArea = calcCombinedRoomArea<--numRooms
OUTPUT “Total room area for the house is “, totalArea
```

The inputNumber sub module would involve a WHILE loop which repeated until the user input was positive:

```
SUB MODULE inputNumber
IMPORT userPrompt
EXPORT number

OUTPUT userPrompt
INPUT number
WHILE number <= 0 DO
   OUTPUT”Input must be positive. Try again.”
   OUTPUT userPrompt
   INPUT number
ENDWHILE
```

Note we IMPORT the user prompt so we can call the same sub module to input the length and width of a room. The calcCombinedRoomArea sub module will loop through and process each room:

```
SUB MODULE calcCombinedRoomArea
IMPORT numRooms
EXPORT totalArea

totalArea = 0
FOR room = 1 TO numRooms INC BY 1
   length = inputNumber <-- “Enter room length”
   width  = inputNumber <-- “Enter room width”
   roomArea = length x width
   totalArea = totalArea + roomArea
ENDFOR
```

If we look at the above pseudo code carefully we can see that loop nesting does occur because the inputNumber sub module contains a while loop and it is called twice within the FOR loop.

Example Problem

Suppose we required an algorithm which processed the student results for a course. We can apply all the concepts and skills we have learnt so far and come up with an algorithm. The algorithm needs to output the final grade for each student along with the minimum, maximum and average mark for the course. The assessment for the course consists of two tests, each worth 15% and a final exam which is worth 70%. All marks will be input as a percentage (i.e. an integer in the range 0 to 100). The final mark is calculated by converting each test mark to a score out of 15, the exam mark to a score out of 70 and then adding the three marks together. The table below shows the relationship between the final mark and the grade awarded to the student.

Final Mark	Grade
90-100	A+
80-89	A
70-79	B+
60-69	B
50-59	C
0-49	F

Analysis of the above problem description reveals that for each student the algorithm has to:

- Input the three marks
- Calculate the final mark.
- Calculate the grade.
- Output the grade.

The algorithm also has to find the maximum and minimum final mark, as well as calculate the average mark. The first version of our algorithm will not deal with the maximum, minimum and average mark. Once we have a working

algorithm for processing each student, we can add the steps required for the rest. Temporally reducing the scope of an algorithm so that it becomes easier to solve is another commonly used algorithm design technique.

Notice that the use of the phrase "for each" in a problem definition will more than likely result in a FOR loop being required (though not always). In this case its "For each student". That immediately implies that a loop is required which will loop from the first student to the last student. As long as we know how many students are being processed then we know how many times the loop will repeat. Therefore a FOR loop is required:

```
FOR nextStudent = 1 TO numStudents CHANGEBY 1
   INPUT test and exam marks
   Calculate final mark
   Calculate grade from final mark
   OUTPUT grade
ENDFOR
```

The FOR loop needs to know how many students, therefore this must be input from the user beforehand. The input of the number of students needs to be validated, therefore we need a loop. It makes sense to use a sub module to do the input:

```
numStudents = inputNumStudents
FOR nextStudent = 1 TO numStudents CHANGEBY 1
   INPUT test and exam marks
   Calculate final mark
   Calculate grade from final mark
   OUTPUT grade
ENDFOR

SUB MODULE inputNumStudents
IMPORT None
EXPORT numStds

INPUT numStds
WHILE numStds <= 0 DO
   OUTPUT “Number of students must be > 0.”
   INPUT numStds
ENDWHILE
```

Similarly the input of each mark has to be validated to ensure that it is in the range 0 to 100. The algorithm for the input of each mark is the same, therefore we develop one sub module which we call three times. Hence the main algorithm becomes:

```
numStudents = inputNumStudents
FOR nextStudent = 1 TO numStudents CHANGEBY 1
   testOne = inputMark<--"Enter Test One mark"
   testTwo = inputMark<--"Enter Test Two mark"
   exam    = inputMark<--"Enter Test Exam mark"
   Calculate final mark
   Calculate grade from final mark
   OUTPUT grade
ENDFOR
```

The inputMark sub module is almost the same as the inputNumber sub module from the previous example:

```
SUB MODULE inputMark
IMPORT userPrompt
EXPORT mark

OUTPUT userPrompt
INPUT mark
WHILE mark <= 0 OR mark > 100
   OUTPUT
        "Mark must be between 0 and 100.  Try again."
   OUTPUT userPrompt
   INPUT mark
ENDWHILE
```

Notice that, as we develop our sub modules, more and more of the steps in MAIN get converted from vague descriptions to precise sub module calls. The calculation of the final mark means converting each test mark to a mark out of 15 and the exam mark to a mark out of 70 and then adding them together. Again a sub module for doing this is the appropriate choice. Hence the Main algorithm becomes:

```
numStudents = inputNumStudents
FOR nextStudent = 1 TO numStudents CHANGEBY 1
   testOne = inputMark<--"Enter Test One mark"
   testTwo = inputMark<--"Enter Test Two mark"
   exam    = inputMark<--"Enter Test Exam mark"
   finalMark = calcFinalMark<--testOne, testTwo, exam
   Calculate grade from final mark
   OUTPUT grade
ENDFOR
```

Now we need to look at the maths required to rescale a mark. In the case of the test marks:

$$\frac{x}{15} = \frac{y}{100}$$

Hence:

$$x = \frac{(y \times 15)}{100}$$

Similarly for the exam mark: $$x = \frac{(y \times 70)}{100}$$

Notice that before we develop the sub module, we put our algorithm aside and got the maths right in our head. This is an essential step that you should never skip. Hence the calcFinalMark sub module is:

```
SUB MODULE calcFinalMark
IMPORT testOne, testTwo, exam
EXPORT finalMark
ASSERTION: All IMPORT values are in the range 0 to
100

testOneScaled = (testOne x 15.0) / 100.0
testtwoScaled = (testTwo x 15.0) / 100.0
examScaled    = (exam x 70.0 ) / 100.0
finalMark = testOneScaled + testTwoScaled +
            examScaled
```

The remaining step is to calculate the grade. Again a sub module is an obvious choice. Hence the Main algorithm becomes:

```
numStudents = inputNumStudents
FOR nextStudent = 1 TO numStudents CHANGEBY 1
   testOne = inputMark<--"Enter Test One mark"
   testTwo = inputMark<--"Enter Test Two mark"
   exam    = inputMark<--"Enter Test Exam mark"
   finalMark = calcFinalMark<--testOne, testTwo, exam
   grade = calcGrade<--finalMark
   OUTPUT grade
ENDFOR
```

A Linear IF statement is the most sensible way to convert from final mark to grade:

```
SUB MODULE calcGrade
IMPORT mark
EXPORT grade
ASSERTION: 0 <= mark <= 100

IF 100 >= mark >= 90 THEN
   grade = "A+"
ELSE IF mark >= 80 THEN
   grade = "A"
ELSE IF grade >= 70 THEN
   grade = "B+"
ELSE IF grade >= 60 THEN
   grade = "B"
ELSE IF grade >= 50 THEN
   grade = "C"
ELSE  ASSERTION: grade <= 50 and >= 0
   grade = "F"
ENDIF
```

Our algorithm is almost complete. The final step is to add the processing required to find the maximum and minimum and calculate the average. All three must be in the range 0 to 100. The average is calculated by dividing the sum of the marks by the number of students. That means that we need to calculate the sum of the marks by adding the next mark to the sum each time through the loop. To find the maximum and minimum we need to use and IF-THEN statement to compare the current maximum with the next mark and similarly the current minimum with the next mark. The only problem is what to we initialise maximum and minimum to? If we set minimum to a value which is larger than any possible mark then it will be updated by the first mark because the first mark is going to be less than the current minimum. Similarly if we set maximum to a value which is smaller than any valid mark then it will be updated to the first mark because the first mark is going to be larger than the maximum. Again, notice we have thought our way through the problem at a human level before trying to come up with an algorithm. Hence our Main algorithm becomes:

```
numStudents = inputNumStudents
maximum = -1
minimum = 101
sum = 0
FOR nextStudent = 1 TO numStudents CHANGEBY 1
   testOne = inputMark<--”Enter Test One mark”
   testTwo = inputMark<--”Enter Test Two mark”
   exam    = inputMark<--”Enter Test Exam mark”
   finalMark = calcFinalMark<--testOne, testTwo, exam
   grade = calcGrade<--finalMark
   OUTPUT grade
   IF finalMark < minimum THEN
      minimum = finalMark
   ENDIF
   IF finalMark > maximum THEN
      maximum = finalMark
   ENDIF
   sum = sum + finalMark
ENDFOR
average = sum / numStudents
OUTPUT average, maximum, minimum
```

Placing the FOR loop in MAIN seemed like a good idea but now the size of MAIN is starting to mushroom. Its still not too large but we could decide to place the FOR loop in its own submodule.

Hence main becomes:

```
numStudents = inputNumStudents
processStudents<--numStudents
```

Then we move the FOR loop and associated processing into its own sub module:

```
Sub Module processStudents
IMPORT numStudents
EXPORT None
ASSERTION: numStudents > 0
maximum = -1
minimum = 101
sum = 0
FOR nextStudent = 1 TO numStudents CHANGEBY 1
   testOne = inputMark<--”Enter Test One mark”
   testTwo = inputMark<--”Enter Test Two mark”
   exam    = inputMark<--”Enter Test Exam mark”
   finalMark = calcFinalMark<--testOne, testTwo, exam
   grade = calcGrade<--finalMark
   OUTPUT grade
   IF finalMark < minimum THEN
      minimum = finalMark
   ENDIF
   IF finalMark > maximum THEN
      maximum = finalMark
   ENDIF
   sum = sum + finalMark
ENDFOR
average = sum / numStudents
OUTPUT average, maximum, minimum
```

Conclusion

Looping controls structures complete the set of tools required for designing simple algorithms. We now have all the basic tools required to design algorithms. Another observation is the importance of slowly refining our algorithms. Could you have come up with the last version of our algorithm in one attempt?

For many years this was enough. However, over time, the need to group related information together was recognised. For example, rather than having all the information about a customer in separate variables, it would be great to be able to place them in one big composite variable. Also software developers found they were repeating themselves a lot. For example, if a software application had code to sort a list of integers and then needed to also sort a list of real numbers, the original code could not be reused. It had to be copied and then the variable declarations changed from integer to real. The other form of repetition is where there is existing code to perform some task and then a slight variation is required as well. Again the initial code has to be copied and then modified. For many years there was no answer to these problems. Then Object Orientation was put forward as the magic solution. As you will see in the last few chapters, OO is not a magic solution but it is a viable solution to a lot of these problems. The remaining chapters will discuss the basics of Object Orientation. However, as you delve deeply into the depths of OO, don't forget that EVERYTHING we have discussed so far still applies.

Chapter 6

"The History of every major Galactic Civilisation tends to pass through three distinct and recognisable phases, those of Survival, Inquiry and Sophistication, otherwise known as the How, Why and Where phases. For instance, the first phase is characterised by the question How can we eat? the second by the question Why do we eat? and the third by the question Where shall we have lunch?"
Douglas Adams, The Hitch-Hiker's Guide to the Galaxy", 1979.

Introduction to Object Orientation

Introduction

The purpose of this book is to introduce students to Object Oriented Algorithm Design. This is not a book on Object Oriented Software Engineering (OOSE) and students should realise that once they have understood the concepts presented in this book, they still have much more to learn about OO. So far we have treated variables as repositories of simple information such as numbers or Boolean. Object variables are capable of much more. Under OO, the algorithm is divided up into a collection of sub components called classes. Each class has an assigned role to play and consists of a set of variables and sub modules. Different classes communicate with each other by calling each other's sub modules. Different classes share relationships with each other which define how they interact. The determination of what classes are required in a software system is outside the scope of this book. Texts on Object oriented Software Engineering cover the processes and techniques for determining the classes required. The goal of OO was to provide a means of minimising the effort required to fix or modify software by maximising the re-use of code and minimising the impact that changes to one part of the code can have on the rest of the code (i.e. minimise *coupling* and maximise *cohesion*). Prior to OO, the only sub division of the code was sub modules. Under OO the sub modules are collected into different classes. OO provides mechanisms for specifying the visibility of sub modules. i.e. a sub module can be visible to the entire software system, to a sub set of classes within the software system or to just the class in which it is defined. If changes are made to a sub module then only the sections of the code in which it is visible need to be checked for possible modifications. In other words control of visibility is a tool which is used to reduce coupling as much as possible. This is known as *Information Hiding* and OO provides the ideal infrastructure for employing this strategy. OO uses a class relationship known as Inheritance to increase the re-usability of code and minimise the amount of new code that

needs to be written when modifying or extending the functionality of software. Under OO, we call sub modules methods, so for the remainder of this book, when discussing OO, sub modules will be called methods.

Basic Classes

Up until now we have used sub modules to divide our algorithms up into sub steps. Now we are going to use classes for a similar purpose. Each class consists of:

- A set of variables which have a scope that makes them global to the class but invisible to any code outside of the class. These variables are known as class *fields* (also known as class *members* or class *attributes*).
- A set of methods which are visible to code outside the class (also known as class *operations*).
- A set of methods which are invisible to code outside of the class.

An object is a variable in memory which is made according to the specifications defined in the class definition. A good analogy would be a set of house plans are the equivalent of a class and an actual house constructed from the plans is the equivalent of an object. The class specifies what should be available via the object variable. The methods in a class are there to enable objects of the class to service their class fields and to fulfil whatever role or responsibility has been assigned to the class. The class fields represent the information needed by the class methods to perform their function. For example, consider a class which administers a date. It would have class fields for day, month and year and a set of methods for initialising, modifying or retrieving day, month and year. Unlike the variables we are used to (i.e. local to a sub module), these variables will remain in memory for as long as the object remains in memory. They will *persist* in memory between method calls.

The *State* of an object is the values that the class fields have been set to for that object. For example, if we had a date class with the class fields for day, month and year and we had two date objects, the first representing a date of 5th March, 2006 and the second representing a date of 9th April 1973 then the state of the first object would be day = 5, month = 3, year = 2006 and the state of the second object would be day = 9, month = 4 and year = 1973. In other words a class has no state, just a specification of what the object state should be and each object has its own state. This is a very important concept because all of the methods in a class interact with object state. They either initialise it, modify it or refer to it. When an object is created from a class we say the object has been *constructed*. Variables which are not objects are referred to as primitive variables because they contain one piece of data and no functionality. The state of an object consists of more than one piece of data and methods can be called via an object variable so objects have both state and functionality.

The methods in a class fit into one of the following categories:

- Constructors: These methods are used to create the object. Their main purpose is to initialise the state of the object being constructed.
- Destructors: These methods are responsible for freeing up all the resources assigned to an object and removing it from memory. Many OO languages (e.g. Java, Python) have garbage collection mechanisms which automatically remove objects which are no longer required from memory. Destructors are not required for this type of language. As most new OO languages have Garbage Collection we will will dispense with any discussion of destructors.
- Accessors: These methods EXPORT information which is either part of the object state or derived from the object state.
- Mutators: These methods are used to modify or *mutate* part or all of the object state.
- Imperative: The word imperative is derived from the Latin word *imperita*, which means *to do*. These methods perform some task but do not EXPORT, initialise or modify object state information.
- Private: These are sub modules called by methods in the above five categories to perform some sub task. They are invisible to any code outside of the class.

Constructors, accessors, mutators and imperative methods are categorised as *public* meaning they are visible to the entire code for the software application.

When referring to public information from outside the class, we state the object we are dealing with followed by the name of the public method that we wish to invoke. For example, suppose we had a class for representing date, within that class suppose we have a public method called getDay which exports the day value from the date object then we might have pseudo code which looks like:

```
day = myDate.getDay
```

In other words execute the getDay method from the Date class and use it to pass a copy of the day value from the myDate object and place it in the variable called day. The use of a period as punctuation between the name of the object and the name of the public method being referenced is used fairly universally across OO languages but any punctuation could be used in pseudo code, as long as you are consistent. Some examples might be:

- day = myDate->getDay
- day = myDate:getDay

One has to be careful though. For example what is wrong with:

day = myDate-getDay

This is the point where you are probably starting to feel a bit lost. As Douglas Adams said, "Don't Panic", it will all become clear. Lets look at a simple example of a class and then go over the six categories of methods in more detail.

Suppose we need a class to represent peoples' height in metres and cm. Such a class would have class fields for metres and cm. The constructors would initialise metres and cm, the accessors would allow the metres and cm values to be EXPORTed to code outside the class and the mutators would allow metres and cm to be modified. At first very little of the pseudo code below will make any sense at all to you but we will discuss each method in detail and the see how objects of this class are created and used.

```
CLASS: Height

CLASS FIELDS: metres, cm

Public Default Constructor: IMPORT None

metres = 0
cm = 0

Public Alternate Constructor IMPORT inMetres, inCm

IF validateMetresAndCm<--inMetres, inCm THEN
   metres = inMetres
   cm = inCm
ELSE
   metres = 0
   cm = 0
ENDIF

Public Copy Constructor: IMPORT inHeight (Height Object)

metres = inHeight.getMetres
cm = inHeight.getCm

Public MUTATOR: setHeight IMPORT inMetres, inCm
                          EXPORT None

IF validateMetresAndCm<--inMetres, inCm THEN
   metres = inMetres
   cm = inCm
ENDIF

Public ACCESSOR getMetres IMPORT None  EXPORT metres
Public ACCESSOR getCm     IMPORT None  EXPORT cm

Public ACCESSOR equals IMPORT inHeight (Height Object)
                       EXPORT isEqual

IF metres=inHeight.getMetres AND inCm=inHeight.getCm THEN
   isEqual = true
ELSE
   isequal = false
ENDIF
```

```
Public ACCESSOR toString  IMPORT None  EXPORT stateString

stateString = "Meters = ", metres, " and cm = ", cm

IMPERATIVE outputState  IMPORT None  EXPORT None

OUTPUT metres, " m and ", cm, " cm."

Private validateMetresAndCm IMPORT m, cm  EXPORT isValid

IF m >= 0 AND cm >= 0 THEN
   isValid = true
ELSE
   isValid = false
ENDIF

ENDCLASS
```

The first point to remember is that you need to have a good understanding of sub module IMPORT and EXPORT or the rest of this chapter will be almost impossible to understand. We will now discuss each of the categories of methods (i.e. constructors, accessors, mutators, imperative and private) using the height class as an example.

Constructors

The first thing to notice is there is more than one constructor. The difference between each constructor is defined by the IMPORT for the constructor. The idea is that different types of information might be used to initialise the state of the object being constructed. The three constructors in the height class example illustrate the three categories of constructor. Before an object can be used, it must be constructed by calling a constructor. These means that the life of every object begins with a constructor call.

The *Default Constructor* has no IMPORT information and simply initialises the class fields to sensible defaults. The default constructor has to be provided as part of the class design. All OO languages will automatically insert a default constructor if one has not been provided. When this happens the default constructor will not initialise the class fields, it needs to be there for a different reason (which we will discuss in the chapter on inheritance). The use of a default constructor is loosely equivalent to setting a numeric variable to 0.0 or a character variable to a blank space.

The Second category of constructor does not have a formal name, I simply refer to them as alternate constructors (i.e. not a default constructor and not a copy constructor). This type of constructor has IMPORT which is used to initialise the class fields. We always want to ensure that every object constructed has a valid object state which is why this type of constructor will always validate its IMPORT. If the IMPORT turns out to be valid then it is used to initialise the class fields. If not then the class fields are initialised to the same default values used in the default constructor. OO languages do not

force designers to structure their method algorithms so that the object state is always valid. However ensuring the object state is valid in this manner means that the rest of the software which is using objects of that class can always assume the objects have a valid state. In other words responsibility for validating the state of the object rests with that class and no other. This helps reduce coupling and simplify any algorithms making use of objects of this class. As you can see from the algorithms for all three constructors in the Height class, the goal of every constructor is to make sure the state of the object being constructed is initialised. The state of the object is defined by the values of the class fields, so every constructor must ensure that the class fields have been initialised to valid values. Note that in all cases the constructors always ensure that metres and cm have been initialised to valid values.

The **Copy Constructor** is used to create a duplicate object which has exactly the same state as the IMPORT object. When copying the contents of primitive variables this is accomplished with a simple assignment statement:

```
ralph = joe
```

We cannot use an assignment statement when copying objects because we are not simply talking about copying a couple of bytes from one memory location to another. The contents of an object are more complex. The height class is an example of a very simple class but often classes and hence objects can be far more complex. Having said that, there are OO languages which will allow the use of an assignment operator and will perform the equivalent steps to calling a constructor (sometimes referred to as implicit constructor calls). As you are learning OO, we will assume this is not the case. You need to understand the overheads involved in copying objects and, as a student, you need to be clear on when you are calling a constructor and when you are not.

```
Copy Constructor: IMPORT inHeight (Height Object)

metres = inHeight.getHeight
cm = inHeight.getCm
```

As we can see from the algorithm used in the height class example (see above), the accessors are used to extract the state information from the IMPORT object (metres and cm in the case of Height class) and use those values to initialise the class fields. Validation is not required if we assume that the IMPORT object will always have a valid state.

Mutators

Mutators are similar in purpose to constructors. The fundamental difference is that constructors are assigning the initial values to the class fields whereas mutators are updating them. If the IMPORT to a mutator is not valid then the mutator algorithm can leave the class fields unchanged because they have already been initialised whereas the constructors must set them to a value

because they have no initial value when the constructor is called. The other difference is that a mutator does not have to mutate the entire object state, it might only mutate part of the object state whereas a constructor must initialise the entire object state. As with the alternate constructors, the IMPORT is validated to ensure that after the mutation, the state of the object is still valid. In the Height class example there is one mutator for mutating the entire object state (i.e. metres and cm). An alternate approach would have been to have individual mutators for mutating metres and cm. i.e.:

```
MUTATOR setMetres  IMPORT inMetres  EXPORT None

IF inMetres >= o THEN
   metres = inMetres
ENDIF

MUTATOR setCm  IMPORT inCm  EXPORT None

IF inCm >= o THEN
   cm = inCm
ENDIF
```

Neither approach is incorrect and in fact there would be nothing wrong with including all three mutators in the Height class. Sometimes the validation of the IMPORT is dependant upon the other IMPORT items. In those cases the use of individual mutators would not be appropriate. For example, in a class representing date, the validation of the day value, requires knowing the month and year, hence individual mutators for mutating day, month and year would not be appropriate. It is very common for the name of a mutator to start with the word *set* and for the IMPORT to be a direct reflection of the class fields. It is equally important to realise that, while this is common, it is not always the case. It is possible to have mutators which have no IMPORT and use some form of algorithm to internally calculate the new state information. For example suppose, in a class which generated lottery numbers, there was a mutator called generateLotteryNos which used a random number generator to generate the numbers. It is mutating the object state (by changing the lottery numbers) but its name doesn't start with the word *set* and it has no IMPORT.

Accessors

As stated previously, the role of an accessor is to EXPORT information which is either object state information or is derived from object state information. As with mutators, it is common to have a set of accessors whose name starts with the word *get* and who EXPORT a copy of some of the object state. In the Height class example the accessors *getMetres* and *getCm* are examples of this type of accessor. Note that because they are simply EXPORTing state information, there is no algorithm. As with mutators it is important to realise that not all accessors have a name that starts with *get* and simply EXPORT some object state information. Some accessors will EXPORT information which has been calculated using the object state information. In those cases the accessor might not have a name which starts with the word *get* and will involve a (possibly complex) algorithm. For example suppose we had a class

which was responsible for representing a colour picture. We might have an accessor which EXPORTed a black and white version of the image, in which case its name might be something like generateBWPicture and an algorithm for processing the colour pixels into black and white would be required.

The use of a toString accessor is a convention introduced in the Java class library that comes with the Java compiler and interpreter. The main use is for debugging. The idea is the EXPORT string describes the state of the object in string form. Any string is easily output and so handy in debugging where we might want to know what the state of an object at some point in the algorithm.

The need for an equals accessor method is not so straight forward. It is quite common to compare to primitive variables for equality. When doing so we simply use an equals sign in a Boolean expression. Because the equals sign is in a Boolean expression we know it is a comparison and not an assignment:

```
IF x = y THEN
   a = b
ENDIF
```

In the above pseudo code, x = y, is comparing x to y and evaluating to true if they are equal and false if not whereas a = b is assigning a copy of the contents of b to the memory associated with a. The problem with objects is that we are not simply comparing one piece of data with another we are comparing object states. The second problem is that the criteria for equality with primitives is simple (are the bit patterns in memory identical), with objects it is subjective. For example, suppose we had a class for representing a student in a university. The class fields for this class would describe the student's name, date of birth, address etc. However the only item of information which would be guaranteed to be unique would be the student number that the student was assigned when they joined the university. Hence an equals method for this class would only compare student numbers. It would be pointless to compare the other information because it is possible. however unlikely, to have two students where all the other information was the same. In the case of the Height class, we do want to compare both metres and cm but it is important to realise that it is not always the case that all the state information is compared. That means that if I wanted to compare two Height objects for equality the Boolean expression would need to call the equals method. i.e:

```
IF heightOne.equals( heightTwo) THEN
```

OR

```
IF heightTwo.equals( heightOne) THEN
```

Imperatives

An imperative method is one which is public but is not a constructor, accessor or mutator. In the Height class example we have a imperative method called

outputState which outputs the metres and cm values for the object to the user. Imperative methods are rare in small classes but quite common in more complex classes. Note the difference between toString, which EXPORTs a string describing the object state to the part of the algorithm that called it, and outputState which outputs the same information to the user. outputState is not an accessor because it does not EXPORT state to wherever it was called.

Private Methods

In chapter three, we discussed the idea of using sub modules to accomplish sub steps in an algorithm. This idea is not discarded once we involve OO, it is as vital as ever. However because these sub modules are sub steps we want to hide them from the rest of the software system because they are part of the implementation of a public method and as such we want the option of changing or even removing them without requiring modifications anywhere else in the software (i.e. low coupling). The Height class is a very simple class so the use of a sub module to validate metres and cm provides no real advantage over simply placing the Boolean expression (i.e. metres >=- 0 AND cm >= 0) where the validation sub module is called. I really only included it as an example of a private method. As with non OO algorithms, we only need sub modules when our algorithms involve more than a few lines of pseudo code. In commercial software, classes would be more complex and so the use of private sub modules is quite common.

An Introduction to The Unified Modelling Language (UML)

Around about the time when OO started to take hold in industry, the need for a visual OO design language arose. The idea was to use diagrams to represent the various OO artefacts and to show how they interact to achieve the gaols of the software being designed. The three most prominent Software Engineering academics in this area at the time were Grady Booch, James Rumbaugh and Ivar Jacobson. All three had proposed different types of diagrams and notations for representing OO software systems. Towards the end of the 20th Century, the three combined forces (they were nick named the *Three Amigos*) and combined their ideas into one visual language which they called the *Unified Modelling Language* or UML. UML has proved very successful and so powerful that it has been adopted for use in many non software related disciplines. UML has become the standard way for software producers to visualise their designs. There is a lot to teach with regard to UML and that is not the purpose of this book. However UML provides some useful tools for illustrating OO designs so we will use a small amount of UML in this book. Specifically the UML Class Diagram which shows classes and their relationships. For the moment we will not consider class relationships, that will be dealt with in future chapters. A UML class diagram is made up of a set of class icons with lines joining them. Each connecting line on a UML class diagram connects two class icons and represents a relationship between the two classes. Each class icon consists of a rectangle, inside the rectangle is the name of the class. Optionally, the class fields (aka members, attributes) and the class methods (aka operations) are also listed. In front of

each class field or method is a plus or minus sign. The plus sign means that the method is public, the minus sign means the class field or method is private. There is one other category of visibility known as *protected* (represented by the # character) but we will look at that in future chapters. Hence the class icon for the Height class would look like:

Height
-metres -cm
+Constructor IMPORT None +Constructor IMPORT inMetres, inCm +Constructor IMPORT inHeight (Height) +setHeight IMPORT inMetres, inCem EXPORT None +getMetres IMPORT None EXPORT metres +getCm IMPORT None EXPORT cm +equals IMPORT inHeight (Height) EXPORT isEqual +toString IMPORT None EXPORT stateString +outputState IMPORT None EXPORT None -validatemetresAndCm IMPORT m, cm EXPORT isValid

The ordering must be class name followed by the class fields followed by the methods. Strictly speaking the use of IMPORT/EXPORT is not part of UML and so the class icon above is not quite standard UML. In Standard UML it would like:

Height
-metres -cm
+Height() +Height(inMetres: integer, inCm:real) +Height(inHeight: Height) +setHeight(inMetres:integer, inCm:real): void +getMetres(): integer +getCm(): real +equals(inHeight:Height): boolean +toString() : String +outputState() : void -validateMetresAndCm(m:integer,cm:real):boolean

Note also the constructors are not explicitly labelled as such but have a name which is the same as the name of the class. At this point, you are probably

still struggling a bit with the concept of data flowing in and out of a method and are most definitely struggling with what a constructor is used for and so I will use my *not quite* UML style for class icons.

Creating and Using Objects

I didn't explicitly state it, but when using primitive variables, there are only three basic forms of interaction:

- Initialisation: Giving the variable its first value (e.g. x = 0).
- Mutation: Updating the value of a variable (e.g. x = 42). Note with primitive variables the first assignment of a value to a variable is initialisation and any subsequent assignments count as mutations.
- Reference: the value of the variable is read and used in some way (e.g. x = y + 5, the y variable is being read and used to update or initialise the x variable).

If you think back on all the examples you have seen so far, every reference to a variable can be placed into one of the above three categories. Algorithms which make use of object variables will also require the same three categories of functionality:

- Initialisation: Use a constructor to initialise the object state.
- Mutation: Use a mutator to modify part or all of the object state.
- Reference: Use the appropriate accessor to match the type of reference (e.g. use the equals accessor if comparing two objects for equality).

In other words, where we simply referred to a primitive variable, we have to use a method to do the equivalent thing with an object. Lets use a simple algorithm as a means of illustrating the difference between how we manipulate objects and how we manipulate primitive variables.

Suppose we wanted an algorithm which would:

- Input a set of integers from the user.
- Stop inputting numbers when the user enters -1 for the input.
- Calculate and output the average of the numbers to the user.

Using primitive variables, we can easily come up with a simple algorithm for this:

```
sum = 0
count = 0
INPUT number
WHILE NOT (number = -1 ) DO
   count = count + 1
   sum = sum + number
ENDWHILE
average = sum / count
OUTPUT average
```

Now suppose instead of using primitive variables, we had a class called Integer and all of our integer variables in our algorithm were represented as

Integer objects. Firstly the class icon for Integer shows the functionality provided by our Integer class:

Integer
-number
+Constructor IMPORT None +Constructor IMPORT inNumber +Constructor IMPORT inInteger (Integer) +setInteger IMPORT inNumber EXPORT None +getInteger IMPORT None EXPORT number +equals IMPORT inInteger (Integer) EXPORT isEqual +toString IMPORT None EXPORT stateString

This means our algorithm would become:

```
construct sum using 0
construct count using 0
construct negOne using -1
INPUT tmpInt
construct number using tmpInt
WHILE NOT (number.equals(negOne) ) DO
   count.setInteger( count.getInteger + 1)
   sum.setInteger( sum.getInteger + number)
ENDWHILE
construct average using sum / count
OUTPUT average.toString
```

The initialisation of sum and count now involves calling a constructor with the initial value (i.e. 0) as the IMPORT to the constructor. The equality comparison is now done with an equals method. Notice we had to provide an Integer object with a state of -1 to the equals method to make the equality comparison possible. Note also the we actually need a notEquals method but we got around that by using the equals method with the Boolean NOT operator.

The increment of count and sum needs a bit more explaining. To add one to count we need to extract the integer value from the count object using the getInteger accessor, add 1 to it and use the result as the IMPORT to the setInteger mutator. An alternative to this would have been to include and increment mutator:

```
PUBLIC MUTATOR increment  IMPORT None  EXPORT None
number = number + 1
```

The the line:

```
count.setInteger( count.getInteger + 1)
```

would become:

```
count.increment
```

Finally we use to toString accessor to provide the required output for the OUTPUT statement. Keep in mind that the basic functionality that we require is the same as with primitives, its just that with objects we use the various methods provided in the class to achieve the same result.

Method Overloading and Polymorphism.

ALL OO languages provide a mechanism known as method overloading so that alternate versions of the same method can be provided in a class and then the appropriate version is determined where the method is called. The different versions of a method are distinguished by their signatures. The signature of a method is composed of the name of the method and the method parameters. In our case the IMPORT and EXPORT. When the method is called, the version of the method which matches the IMPORT/EXPORT of the call is the one that is executed. Consider the integer class. We could add a second equals method which IMPORT a primitive integer value instead of an Integer object. Then we would have two equals methods:

```
Public ACCESSOR equals IMPORT inNumber(Integer object)
                      EXPORT isEqual
isEqual = false
IF number = inNumber.getInteger THEN
   isEqual = true
ENDIF

Public ACCESSOR equals IMPORT inInt  EXPORT isEqual
isEqual = false
IF number = inInt THEN
   isEqual = true
ENDIF
```

When the equals method is called with a primitive integer as its IMPORT then the second version is executed. If the equals method is called with an Integer object as IMPORT then the first version of the equals method is executed. There are two times when the process of matching method signature to method call is done. The first is at compile time when the compiler for the OO programming language is converting the human readable code into machine executable code. Sometimes it is not possible to determine which method is being called at compile time, in those case the matching of method to method call is done at execution time (i.e. as the program is running). When the compiler determines which method is being called we call it method overloading. When the determination is made at run time we call it polymorphism. At this point it is not possible for you to understand why the compiler could not determine the required version of the method. The need for polymorphism is driven by a class relationship called Inheritance. Inheritance is *THE* most significant tool OO provides us with. However the effects of inheritance are both profound and subtle and so we will spend an entire chapter discussing inheritance.

Operator Overloading.

There are some OO programming languages that use a technique known as *operator overloading* which means that, if the code is written to allow it, object variables can be treated the same as primitives. Traditional operators (e.g. +, - /, *) are overloaded to have a specified behaviour when used with objects of a particular class. For example the plus operator might be overloaded in our Integer class so that we could have steps like:

```
a = b + c
```

where a, b and c are Integer objects. What actually happens is that the use of the overloaded operator (plus in the above example) results in a method being called in the class (i.e. the Integer class in our example). Unfortunately when this technique can be misused to create cryptic and often unstable code. For example, suppose the plus operator could be overloaded such that the object state of the operands is mutated as well as the variable on the left hand side of the equals sign. This violates the basic mathematical concept of an operator and so leads to code that has an unpredictable side effect.

From a learning perspective, being able to use objects in this way hides the fact that they are objects and that accessor and mutator methods are being called, something we need to be very clear about when trying to understand OO algorithm design. How would we achieve the same functionality without using operator overloading? Simply by providing methods to do the required operation. One way would be to provide an Imperative method:

```
Public IMPERATIVE plus IMPORT a, b (Integer)
                      EXPORT c
tmpInt = a.getInteger + b.getInteger
construct c using tmpInt
```

Another version could be a mutator where the result of the addition is used to mutate the object state:

```
Public MUTATOR plus IMPORT a(Integer)
                    EXPORT None
number = number + a.getInteger
```

Given three Integer objects called ralph, joe and betty we could use the two methods as shown below:

```
betty = joe.plus<--ralph

joe.plus<--betty
```

Note because of method overloading so that we could include both versions of the plus method in the Integer Class.

Pure Versus Impure Object Oriented programming Languages

So far all of the example classes and algorithms have assumed that the OO programming language chosen for implementation have both primitive and object variables available. Not all OO programming languages allow this. A pure OO programming language has no primitive date types (e.g. Eiffel). All variables are object variables. An impure programming language has both primitive and object data types (e.g. Java). Impure OO programming languages acknowledge that the use of objects has overheads which are perhaps not needed when doing simple tasks (e.g. a FOR loop index just needs a few bytes of memory for the counter). The first OO programming languages were all pure but more recent languages are impure. I know impure sounds like the coders should wear rubber gloves and masks and that the language is in someway inferior to pure OO languages. The reality is that impure languages are more pragmatic in terms of overheads and a more realistic choice in terms of execution speeds and coding.

An Example Class and an Example Algorithm.

Suppose we needed to design a software application that acted as a calculator for complex numbers. The application would display a menu and allow a user to:

- Input new values for the complex numbers.
- Add the two numbers together and output the result.
- Subtract the first from the second and output the result.
- Multiply the two numbers together and output the result.
- Divide the first by the second and output the result.

Before we start in on what classes and objects will be required, we need to think about the basic steps required. Also, in the real world, the software would have a graphical user interface. Being new to the discipline of software design and implementation, you are not ready for the issues involved in event handling and graphical user interfaces so we will settle for a non-graphical user interface. The basic algorithm would be:

```
INPUT complex numbers
quit = false
DO
   OUTPUT menu (choices being add, subtract. multiply,
                divide or input new numbers)
   INPUT choice
   CASE choice OF
     'a', 'A': add numbers and output result.
     's', 'S': subtract numbers and output result
     'm', 'M': multiply numbers and output result
     'd', 'D': divide numbers and output result
     'i', 'I': INPUT complex numbers
     'q', 'Q': quit = true
WHILE NOT quit
```

Using the principles discussed in earlier chapters with regard to the use of sub

modules and algorithm refinement, it makes sense to have a sub module for each operation (do the operation and output the result) and a sub module for inputing a complex number (remember two numbers need to be input (the real and imaginary part). It also seems sensible to have a class for representing complex numbers.

Lets start with the class and the return to the main algorithm afterwards. The class would look like:

Complex
-realPart -imagPart
+Constructor IMPORT None +Constructor IMPORT inReal, inImag +Constructor IMPORT inNumber (Complex) +setNumber IMPORT inReal, inImag EXPORT None +getReal IMPORT None EXPORT realPart +getImag IMPORT None EXPORT imagPart +toString IMPORT None EXPORT stateString +equals IMPORT inNumber (Complex) EXPORT isEqual +add IMPORT inNumber (Complex) EXPORT None +minus IMPORT inNumber (Complex) EXPORT None +multiply IMPORT inNumber (Complex) EXPORT None +divide IMPORT inNumber (Complex) EXPORT None +add IMPORT inA, inB(Complex) EXPORT result(Complex) +minus IMPORT inA, inB(Complex) EXPORT result(Complex) +multiply IMPORT inA,inB(Complex) EXPORT result(Complex) +divide IMPORT inA, inB(Complex) EXPORT result(Complex)

You can see from the above class icon that we are using both mutators and imperative methods for the mathematical operations (i.e. x = x + y as well as x = y + z). The pseudo code for the actual class is shown below:

```
CLASS Complex

Class Fields: realPart, imagPart

Default Constructor IMPORT None
realPart = 0.0
imagPart = 0.0

Alternate Constructor IMPORT inReal, inImag
realPart = inReal
imagPart = inImag
```

```
Copy Constructor IMPORT inNumber (Complex Object)
realPart = inNumber.getReal
imagPart = inNumber.getImag

Public Mutator setNumber
IMPORT inReal, inImag
EXPORT None
realPart = inReal
imagPart = inImag

Public Accessor getReal IMPORT None EXPORT realPart

Public Accessor getImag IMPORT None EXPORT imagPart

Public Accessor toString IMPORT None EXPORT stateString
stateString = realPart, " + ", imagPart, "i"

Public Accessor equals IMPORT inNumber (Complex)
                       EXPORT isEqual

NOTE: when comparing real numbers for equality we must
use a tolerance.  For our purpose we will assume a
tolerance of 0.0001.
isequal = false
IF | realPart - inNumber.getReal| < 0.0001 THEN
   IF | imagPart - inNumber.getImag | < 0.0001 THEN
      isEqual = true
   ENDIF
ENDIF

Public Mutator add IMPORT inNumber (Complex) EXPORT None
imagPart = imagPart + inNumber.getImag
realPart = realPart + inNumber.getReal

Public Mutator minus IMPORT inNumber (Complex)
                     EXPORT None
imagPart = imagPart - inNumber.getImag
realPart = realPart - inNumber.getReal

Public Mutator multiply IMPORT inNumber (Complex)
                        EXPORT None
imagPart = imagPart x inNUmber.getReal +
           realPart x inNumber.getImag
realPart = realPart x inNumber.getReal -
           imagPart x inNumber.getImag

Public Mutator divide  IMPORT inNumber (Complex)
                       EXPORT None
otherReal = inNumber.getReal
otherImag = inNumber.getImag
realPart = (realPart x otherReal + imagPart x otherImag)/
          (otherReal x otherReal + otherImag x otherImag)
imagPart = (imagPart x otherReal - realPart x otherImag)/
          (otherReal x otherReal + otherImag x otherImag)
```

```
Public Imperative add IMPORT inA, inB (Complex)
                      EXPORT result
imag = inA.getImag + inB.getImag
real = inA.getImag + inB.getReal
construct result using real and imag

Public Imperative minus  IMPORT inA, inB (Complex)
                         EXPORT result
imag = inA.getImag + inB.getImag
real = inA.getImag + inB.getReal
construct result using real and imag

Public Imperative multiply  IMPORT inA, inB (Complex)
                            EXPORT result
imag = inA.getImag x inB.getReal +
       inA.getReal x inB.getImag
real = inA.getReal x inB.getReal -
       inA.getImag x inB.getImag
construct result using real and imag

Public Imperative divide IMPORT inA, inB (Complex)
                         EXPORT result

AReal = inA.getReal
AImag = inA.getImag
BReal = inB.getReal
BImag = inB.getImag
real = (AReal x BReal + AImag x BImag)/
       (BReal x BReal + BImag x BImag)
imag = (AImag x BReal - AReal x BImag)/
       (BReal x BReal + BImag x BImag)
```

Notice that we have included all the methods that we think might be useful, not just the ones that we know ill be needed. That is standard practice when designing and implementing classes. Each class is being designed to service a specified role within the software system. At the time it is being designed, the specifics of object communication may not yet be fully understood. Another issue to consider is that each class should be made as generically useful as possible. Other software designs can then make use of classes which have been designed and implemented for previous projects.

Previously we had determined the need to develop a sub module which would input a complex number from the user and sub modules which would do the adding, subtracting etc. Now that we have the Complex class we can design these sub modules:

```
Sub Module inputComplex
IMPORT None
EXPORT number (Complex Object)

INPUT realPart
INPUT imagPart
construct number using realPart and imagPart
```

```
Sub Module add
IMPORT numOne, numTwo (Complex Objects)
EXPORT None
construct oldNumOne using numOne
numOne.add<-- numTwo
OUTPUT oldNumOne.toString, " + ", numTwo.toString, " = ",
       numOne.toString

Sub Module subtract
IMPORT numOne, numTwo (Complex Objects)
EXPORT None
construct oldNumOne using numOne
numOne.minus<-- numTwo
OUTPUT oldNumOne.toString, " - ", numTwo.toString, " = ",
       numOne.toString

Sub Module multiply
IMPORT numOne, numTwo (Complex Objects)
EXPORT None
construct oldNumOne using numOne
numOne.multiply<-- numTwo
OUTPUT oldNumOne.toString, " x ", numTwo.toString, " = ",
       numOne.toString

Sub Module divide
IMPORT numOne, numTwo (Complex Objects)
EXPORT None
construct oldNumOne using numOne
numOne.divide<-- numTwo
OUTPUT oldNumOne.toString, " / ", numTwo.toString, " = ",
       numOne.toString
```

The main algorithm can then be modified to call these sub modules:

```
numOne = inputComplex
numTwo = inputComplex
quit = false
DO
   OUTPUT menu (choices being add, subtract. multiply,
                 divide or input new numbers)
   INPUT choice
   CASE choice OF
     'a', 'A': add<--numOne, numTwo
     's', 'S': subtract<--numOne, numTwo
     'm', 'M': multiply<--numOne, numTwo
     'd', 'D': divide<--numOne, numTwo
     'i', 'I': numOne = inputComplex
               numTwo = inputComplex
     'q', 'Q': quit = true
WHILE NOT quit
```

Notice that the changes to our main algorithm are small and only involve specifying the required sub modules and supplying them with the appropriate

IMPORT/EXPORT. The two variables used to represent the two complex numbers are object variables and so when they are manipulated then this is done via the methods defined in the Complex class. Finally note how all responsibility for the maths related to complex numbers happens in the Complex class. i.e. This class is solely responsible for complex numbers and so all things related to manipulating complex numbers should reside in this class and nowhere else. Finally see that we have methods in Complex which have the same name as the sub modules used in conjunction with the main algorithm. This is allowable because each method is identified by its signature and which class it has been defined in. Some OO languages will allow sub modules to exist outside of a class definition (e.g. C++) while others insist that all methods must be defined within a class (e.g. Java). The latter is a better arrangement because it ensures that all methods have been assigned to a class and prevents the software from becoming a hybrid of OO and non-OO code. We can see from the main algorithm that all of the concepts and skills required to evolve an algorithm from an initial draft to a finished algorithm have not disappeared once we start using classes and objects. The two changes are that some of the variables can now be object variables instead of primitives and that each class has a fairly standard set of requirements in terms of constructors, accessors and mutators.

The algorithms for the constructors, accessor, mutators, imperatives and private methods for all of the examples so far have been very simple. This is because all of the examples are of very simple classes. Clearly this is not always the case and in the real world is more of an exception than a rule. Like learning anything else we need to understand simple examples before we go on to more complex examples.

Types of Classes

There are many different categories of classes and if this was a text book on OO Software Engineering we would be spending a substantial amount of time discussing the different types of classes in detail. The most obvious type of class is called a *container* class. Its role is to contain and administer a set of information. All of the examples so far are container classes. Unlike the examples presented here, container classes are usually more complex because they are normally used to represent information which has been arranged into some form of complex data structure. Depending upon what course you are studying, you will probably be studying data structures in your next semester so further discussion at this point is not appropriate. Another type of class is called a *boundary* class. The role of a boundary class is to sit on the edge of the software and is responsible for communicating with some type of external entity. For example a class which is responsible for opening and closing an electronic door lock. There are many other types of classes but we will focus on reasonably simple container classes in this book as a means of understanding the basic OO concepts.

Conclusion

In this chapter we have dealt with the basic concepts behind simple classes under OO. Our example application only involved the use of a single class. Commercial software projects will involve hundreds of classes. Different sets of classes in a software project will share relationships with each other. These relationships is where the true power of OO reveals itself. There are many different ways in which classes can be related. This book will only deal with the two most important. They are known as aggregation and inheritance. The easiest of these two relationships to understand is aggregation so the next chapter will discuss the aggregation class relationship and the following chapter will discuss inheritance.

Chapter 7

"The fact that we live at the bottom of a deep gravity well, on the surface of a gas covered planet going around a nuclear fireball 90 million miles away and think this to be normal is obviously some indication of how skewed our perspective tends to be."
Douglas Adams, Speech, Digital Biota 2, Cambridge, UK, 1998

The Aggregation Class Relationship.

Introduction

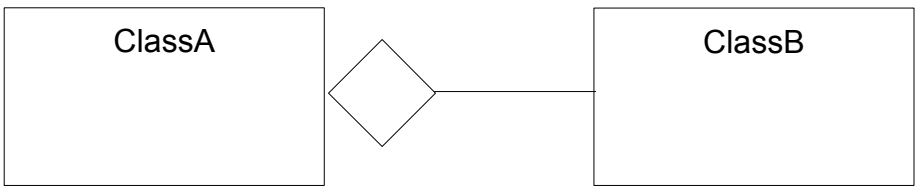
The previous chapter described the basic components of a class. The assumption that was made was that class fields would always be primitive variables. If we relax that assumption then our class fields can also be objects. When a class field is an object of a particular class then the class that contains the class field and the class that the class field is an object of, are related by the class relationship aggregation. The Merrium-webster online diction defines the word aggregation to mean:

“A group, body, or mass composed of many distinct parts or individuals” (http://www.merriam-webster.com/dictionary/aggregation).

So generally an aggregation is a collection of things which combine to represent something else. Under OO, things are objects. In other words part of the object state of one class is composed of the object state of another class. Note that this changes nothing about the structure and purpose of classes as described in the previous chapter. The only thing that changes is how we manipulate the class fields because they are now objects and not primitives.

UML Notation.

If class A has at least one class field which is an object of class B then, in UML, we link the class icons with an aggregation relationship. This consists of an arrow whose head is a diagonal diamond. The arrow is always pointing towards the class that contains the object of the other class.



Note that in the specification for class A there is no explicit statement about aggregation. The aggregation relationship is implied by the declaration of class fields in class A which are specified as being objects of class B.

Aggregation versus no Aggregation

Ninety nine percent of the time a class will have class fields of some sort (the only time I can think of when a class would have no class fields is when we have an extreme version of an abstract class (see next chapter). If the class fields are primitive then things work as explained in the previous chapter and there is no aggregation. If some or all of the class fields are objects of some class then there is aggregation. Whether aggregation is present or not, the overall structure and intent of the class remains the same. There are two differences is the steps for the algorithms for the class methods:

1. If a class field is an object then it is initialised, accessed and updated differently to a primitive variable (as explained in the previous chapter).
2. If a class field is an object and the class that it is an object of is designed so that the state of the object is always valid then the methods in this class can always assume the object is valid and so validation is not required.

Class Responsibility

Before we dive into an example there is one last factor to keep in mind. In the last chapter we talked about preserving class responsibility (i.e. no other class does the jobs that are the responsibility of another class). The temptation to break this rule is very strong when aggregation is present. The way around this temptation is to invoke a method in the class responsible rather than repeating the algorithm in another class.

Confused yet? Don't stress. All we are doing is using all of the concepts and mechanisms that were discussed in the previous chapter in a slightly more complex manner. The easiest way to understand is to use examples which is what we will now do.

A Simple Example of Aggregation

We start by considering a class where all its class fields are primitive. We will then change some of the class fields to objects and observe how the rest of the class changes to compensate. This is an ideal way of understanding the differences in algorithm methods when aggregation is or isn't present.

Let us consider a class for containing the time of day (TimeOfDay). The class will have two class fields: hours and minutes. In the first version of the class they will both be integers. In the second version of the class they will become objects so that we can observe the impact of that change on the TimeOfDay class. For now though lets have a look at version one of our class:

```
CLASS TimeOfDay

Class Fields: hours, minutes

Default constructor: IMPORT None
hours = 0
minutes = 0

Alternate Constructor: IMPORT inHours, inMinutes
IF 0 <= inHours <= 23 AND 0 <= inMinutes <= 59 THEN
   hours = inHours
   minutes = inMinutes
ELSE
   hours = 0
   minutes = 0
ENDIF

Copy Constructor IMPORT inTime (TimeOfDay)
hours = inTime.getHours
minutes = inTime.getMinutes

Mutator: setHours IMPORT inHours  EXPORT None
IF 0 <= inHours <= 23 THEN
   hours = inHours
ENDIF

Mutator: setMinutes IMPORT inMinutes  EXPORT None
IF 0 <= inMinutes <= 59 THEN
   minutes = inMinutes
ENDIF

Mutator: setTime IMPORT inHours, inMinutes  EXPORT None
IF 0 <= inHours <= 23 AND 0 <= inMinutes <= 59 THEN
   hours = inHours
   minutes = inMinutes
ENDIF

Accessor getHours  IMPORT None EXPORT hours
Accessor getMinutes  IMPORT None  EXPORT minutes

Accessor toString  IMPORT None  EXPORT timeStr
timeStr = hours, " Hours and ", minutes, "minutes."

Accessor equals  IMPORT inTime (TimeOfDay) EXPORT same
same = false
IF hours = inTime.getHours AND
   minutes = inTime.getMinutes THEN
   same = true
ENDIF

Accessor getAmPmTime  IMPORT None  EXPORT amPmTime
IF hours < 12 THEN
   timeString = hours, ":", minutes, " a.m."
ELSE
   IF hours = 12 THEN
```

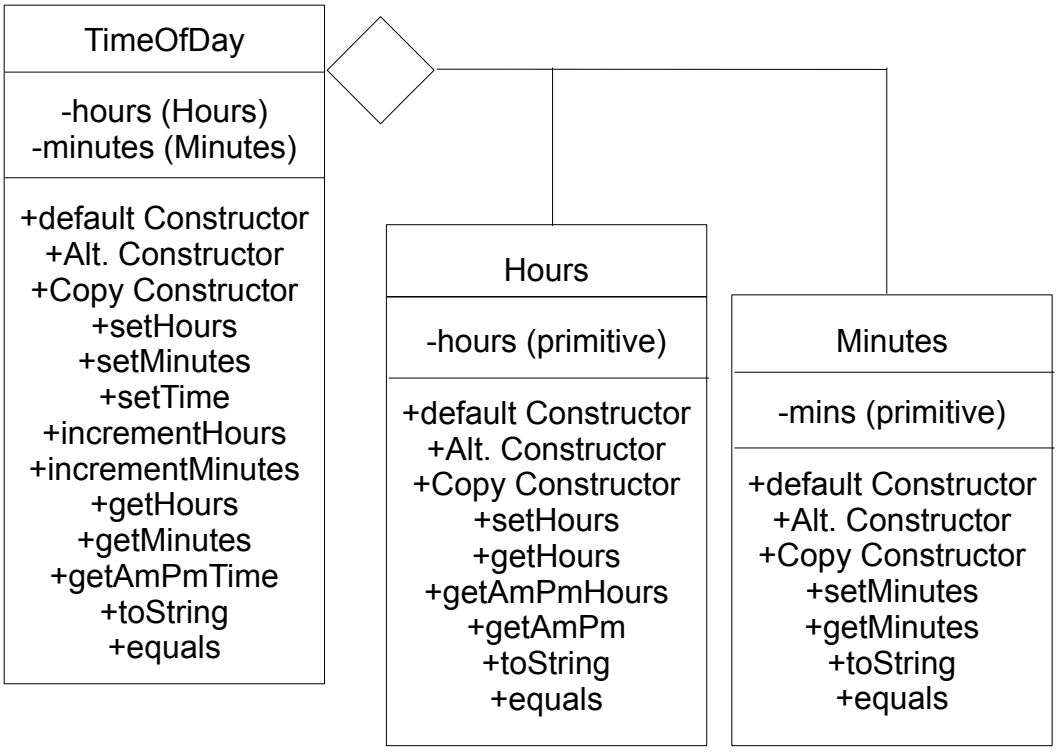
```
      timeString = hours, ":", minutes, " p.m."
   ELSE
      timeString = hours - 12, ":", minutes, " p.m."
   ENDIF
ENDIF

Mutator incrementHours  IMPORT None  EXPORT None
hours = hours + 1
IF hours = 24 THEN
   hours = 0
ENDIF

Mutator incrementMinutes  IMPORT None  EXPORT None
minutes = minutes + 1
IF minutes = 60 THEN
   minutes = 0
   incrementHours
ENDIF
```

As can be seen the above class reflects all of the concepts described in the previous chapter. This includes the last two mutators whose names don't start with the word *set* and have no IMPORT but do mutate the object state (in the previous chapter we described this but did not include an example).

We will now develop separate classes for hours and minutes. These two classes won't involve aggregation but will take over some of the responsibility currently held in the class TimeOfDay. We will then modify TimeOfDay to compensate for these changes. What we will see is this modification will involve a reduction to TimeOfDay in both size and complexity, clearly an advantage. The UML will now look like:



Note that the Hours and Minutes classes involve the same issues and intent as all of the previous examples we have looked at so far:

```
CLASS Hours

Class Field: hours

Default constructor: IMPORT None
hours = 0

Alternate Constructor: IMPORT inHours
IF 0 <= inHours <= 23 THEN
   hours = inHours
ELSE
   hours = 0
ENDIF

Copy Constructor IMPORT inHours (Hours)
hours = inHours.getHours

Mutator: setHours IMPORT inHours  EXPORT None
IF 0 <= inHours <= 23 THEN
   hours = inHours
ENDIF

Accessor getHours  IMPORT None EXPORT hours

Accessor toString  IMPORT None  EXPORT hourStr
hourStr = hours, " Hours"

Accessor equals  IMPORT inHours (Hours) EXPORT same
same = false
IF hours = inHours.getHours AND THEN
   same = true
ENDIF

Accessor getAmPm  IMPORT None  EXPORT amPm
IF hours < 12 THEN
   amPm = " a.m."
ELSE
   amPm = " p.m."
ENDIF

Accessor getAmPmHours  IMPORT None  EXPORT amPmHours
IF hours <= 12 THEN
   amPmHours = hours
ELSE
   amPmHours = hours - 12
ENDIF
```

```
CLASS Minutes

Class Field: minutes

Default constructor: IMPORT None
minutes = 0

Alternate Constructor: IMPORT inMinutes
IF 0 <= inMinutes <= 59 THEN
   minutes = inMinutes
ELSE
   minutes = 0
ENDIF

Copy Constructor IMPORT inMinutes (Minutes)
minutes = inMinutes.getMinutes

Mutator: setMinutes IMPORT inMinutes  EXPORT None
IF 0 <= inMinutes <= 59 THEN
   minutes = inMinutes
ENDIF

Accessor getMinutes  IMPORT None  EXPORT minutes

Accessor toString  IMPORT None  EXPORT minStr
minStr = minutes, "minutes."

Accessor equals  IMPORT inMinutes (Minutes) EXPORT same
same = false
IF minutes = inMinutes.getMinutes THEN
   same = true
ENDIF
```

Now we need to modify the TimeOfDay class so that the hours and minutes class fields change from being primitive variables such that hours becomes an object of the class Hours and minutes becomes an object of the class Minutes. There are three categories of changes:

1. Variable Access: variables must be treated like objects and not primitives (e.g. hours = 0 becomes construct hours using default constructor).
2. Removal of validation of IMPORT: Both the Hours and Minutes classes ensure that their objects will always have a valid state. This means that there is never any need to validate incoming Hours and Minutes objects.
3. The Hours class takes over all responsibility of hours so some of the algorithm steps in the TimeOfDay class gets replaced with method calls to methods in the Hours class. This reduces complexity in the TimeOfDay class and preserves class responsibility. The same goes for the Minutes class.

The modified TimeOfDay class is shown below:

```
CLASS TimeOfDay

Class Fields: hours (Hours), minutes (Minutes)

Default constructor: IMPORT None
construct hours using default constructor
construct minutes using default constructor

Alternate Constructor: IMPORT inHours (Hours),
                               inMinutes (Minutes)
construct hours using inHours
construct minutes using inMinutes
(i.e. call the Hours and Minutes copy constructors).

Copy Constructor IMPORT inTime (TimeOfDay)
hours = inTime.getHours
minutes = inTime.getMinutes

Mutator: setHours IMPORT inHours (Hours)  EXPORT None
construct hours using inHours

Mutator: setMinutes IMPORT inMinutes (Minutes)
                    EXPORT None
construct minutes using inMinutes

Mutator: setTime IMPORT inHours (Hours),
                        inMinutes (Minutes)
                 EXPORT None
construct hours using inHours
construct minutes using inMinutes

Accessor getHours  IMPORT None EXPORT hours
Accessor getMinutes  IMPORT None  EXPORT minutes

Accessor toString  IMPORT None  EXPORT timeStr
timeStr = hours.toString, " and ", minutes.toString

Accessor equals  IMPORT inTime (TimeOfDay) EXPORT same
same = false
IF hours.equals<--inTime.getHours AND
   minutes.equals<--= inTime.getMinutes THEN
      same = true
ENDIF

Accessor getAmPmTime  IMPORT None  EXPORT amPmTime

timeString = hours.getAmPmHours, ":", minutes.toString,
             hours.getAmPm
```

```
Mutator incrementHours  IMPORT None  EXPORT None
tmpHours = hours.getHours
tmpHours = tmpHours + 1
IF tmpHours = 24 THEN
   tmpHours = 0
ENDIF
hours.setHours<--tmpHours

Mutator incrementMinutes  IMPORT None  EXPORT None
tmpMinutes = minutes.getMinutes
tmpMinutes = tmpMinutes + 1
IF tmpMinutes = 60 THEN
   tmpMinutes = 0
   incrementHours
ENDIF
minutes.setMinutes<--tmpMinutes
```

Note that most of the algorithms have become simpler because they can either assume the Hours and Minutes classes have ensured validity or they can simply ask Hours and Minutes class to do the sub task for them. You should analyse the first and second versions of the TimeOfDay class to make sure you understand what has occurred. We will pick some of the methods out to help you along.

Starting with the default constructor. Note that in the first version hours and minutes were primitive so the thing to do was set them both to zero. In the second version, hours and minutes are objects so it doesn't make sense to set them to zero. What we need to do is initialise them (i.e. use a constructor to create them) such that they represent an hours and minutes time of zero. In each case the default constructor does exactly that so the algorithm for the TimeOfDay default constructor calls the Hours default constructor to initialise the hours object and the Minutes default constructor to initialise the minutes object.

The alternate constructor in the first version of TimeOfDay had to validate the IMPORT values for inHours and inMinutes. In the second version inHours becomes an Hours object and inMinutes becomes a Minutes object. The Hours class ensures that an Hours object will always have a valid value for hours. The same is true for inMinutes and the Minutes class. This means that the alternate constructor for the TimeOfDay class can assume that inHours and inMinutes are valid and so the validation part of the algorithm is removed. The second change is because hours and minutes are now objects so we cannot simply assign inHours to hours and inMinutes to minutes. As with the default constructor we have to initialise them by calling constructors. In each case we use the copy constructor because we wish to create new objects for our class fields which have the same state information as the import objects (i.e. hours will have the same state as inHours and minutes will have the same state as inMinutes).

The copy constructor and the *get* accessors are the same (except that with the *get* accessors we are now exporting objects not primitives).

In a similar manner to the constructors, the changes to the mutators are centred around the removal of validation and the use of constructors to re-initialise the hours and minutes objects (for exactly the same reasons).

The changes to the toString and equals accessors is related to preserving class responsibility. It is no longer the responsibility of the TimeOfDay class to compare hours for equality. It must use the equals method in the Hours class for that. The same is true for minutes. Note that it **is** the responsibility of the TimeOfDay class to compare the times for equality (i.e. that BOTH hours AND minutes are the same). In order to achieve that goal and preserve class responsibility it uses the equals methods in the Hours and Minutes class to achieve its goal. Exactly the same issues occur in the toString method which now builds its string of the time using the toString methods in the Hours and Minutes classes.

As a consequence of passing responsibility for issues relating to whether or not the time is in the a.m. or p.m. to the Hours class, the getAmPmTime in the TimeOfDay class becomes much simpler.

The incrementHours and incrementMinutes mutators raise some issues. We could have transferred the responsibility for incrementing hours to the Hours class but we could not have done the same for the increment minutes mutator because it needs to refer to hours to accomplish its task. Consistency in OO design is ALWAYS preferable so, if we cannot transfer responsibility for both mutators then we keep them both in the TimeOfDay class. Also, in both cases we want to increment the values concerned (i.e. hours and/ or minutes). We cannot simply add one to an object so we must extract the value out of the object using an accessor, manipulate it and then put it back in the object using a mutator. The effect is that the second version s of incrementHours and incrementMinutes have become more complex in stead of less! Nobody said that OO was perfect! It isn't and these two mutators are a good example of when it can be a bit annoying! But hey, keep in mind that if OO design was simple, straight forward and non-subjective then I wouldn't have a job and you wouldn't have a highly successful career to look forward to!!

The *Main* Class and the Chicken and the Egg

All of these classes require some other part of the algorithm to call their constructors so that objects can be constructed from them. The problem is, when the main algorithm starts, we don't have any objects yet. The main algorithm will drive the steps which will cause objects to be created, manipulated and then destroyed. Some programming languages (e.g. C++) allow main and other sub modules to be defined outside of class definitions while others (e.g. Java) require all sub modules to be contained within a class definition. In either case, the high level algorithm in main (and possibly a few sub modules called by main) will not map to an object. In other words it will be defined in the same way as described prior to chapter six. If it must be placed in a class (e.g. Java) then that class will have no class fields,

constructors, accessors or mutators. This might seem like some sort of sacrilegious violation of the OO religion. Fortunately OO is not a religion, it is a tool and there is nothing wrong with having the highest level of your algorithm setup this way. Some languages do provide ways of getting around this but students should not go that route if the simplest, most intuitive solution is to simply provide main in a section of code which will never be mapped to an object. In many areas of design there is a phrase:

"Keep it simple stupid!"

In other words the incorporation of unnecessary complexity is a stupid idea. This saying should be engraved upon placards and placed upon the desk of every software engineer on the planet. Complexity is often required but it should always be kept to a minimum. Returning to main, it is often simpler to include the main sub module in a section of the code not mapped to an object.

Putting It All Together

The last two chapters has focused, not on complete software applications, but on individual classes. When software engineers develop commercial software, they do it in collections of classes which are tested and then integrated together. One of the advantages that OO gives us is that the class is a convenient way of dividing up the task. However we must remember that, at some point, all the classes are integrated into a complete software package. This means all of the concepts that were discussed earlier involving step wise refinement and evolving algorithms to achieve some goal still apply. With that in mind, let's take our time example one step further and use the three classes that we have developed as part of a complete software package. We still need to keep things simple so let's say we wish to input two times and output the difference, in minutes, between them. For the sake of convenience, we will assume that both times occur on the same day. As always, we start with a simple draft of our main algorithm. However we also need to consider what class will contain main and any sub modules it might require. Hence we will add a fourth class to our design. This class will look more like the algorithms we developed prior to considering OO. It will not contain constructors, accessors and mutators and no objects of that class will be constructed. It will simply contain the high level algorithm which will drive the other classes in order to accomplish the required task. Our first draft of the main algorithm is below:

```
MAIN
   INPUT time1 and time2
   calculate the difference between them
   OUTPUT difference
```

The next revision will involve incorporating sub modules and assuming the three classes previously developed (i.e. TimeOfDay, Hours, Minutes):

```
MAIN
   time1 = inputTime
   time2 = inputTime
   difference = calcTimeDifference<--time1, time2
   OUTPUT difference
```

Our main algorithm is now complete! Next we consider the two sub modules that we have just invented (i.e. inputTime and calcTimeDifference). The input will be the easiest so we start with that first:

```
Sub Module inputTime
IMPORT not sure?
EXPORT time (TimeOfDay)

INPUT numHours and numMinutes
construct hours using numHours
construct minutes using numMinutes
construct time using hours and minutes
```

The input of hours and minutes should include a validation loop so that hours and minutes will be valid. We know that if they are not, the TimeOfDay class alternate constructor will assign defaults but we don't really want that in this algorithm, hence the validation is required. The input and validation is identical except for the upper bound (23 for hours and 59 for minutes). This means it makes sense to use a sub module and feed it the upper bound:

```
Sub Module inputNumber
IMPORT  upper
EXPORT value

DO
   INPUT value
WHILE value < 0 OR > upper
```

We can now finalise our inputTime sub module by incorporating calls to the inputNumber sub module. At this point we are also confident that inputTime will not require any IMPORT so we end up with:

```
Sub Module inputTime
IMPORT None
EXPORT time (TimeOfDay)

numHours= inputNumber<--23
numMinutes = inputNumber<--59
construct hours using numHours
construct minutes using numMinutes
construct time using hours and minutes
```

Note that the EXPORT is a TimeOfDay object and not a primitive. As we

wade further and further into the ocean of OO this will be more common than the IMPORT or EXPORT of primitives.

We now move onto the calcTimeDifference sub module. We know we need the two times to calculate the difference and we know the difference is to be expressed in minutes. Hence the IMPORT will be the two TimeOfDay objects and the EXPORT will be a primitive representing minutes. As always, we start with a rough draft of the algorithm:

```
Sub Module calcTimeDifference
IMPORT time1, time2 (TimeOfDay)
EXPORT difference

IF time1 < time2 THEN
   difference = time1 - time2
ELSE
   difference = time2 - time1
ENDIF
```

Given that time1 and time2 are TimeOfDay objects, we have some issues to resolve:

- We cannot subtract two objects. We need to convert the hours and minutes in each to an elapsed time, in minutes, since midnight. Then we can subtract the elapsed times.
- We have no way of doing a less than comparison between the two objects.

This is where we need to consider class responsibility. We could develop a lessThan and a calcElapsedTime sub module to be incorporated with the other sub modules in the class containing main. However these tasks are really the responsibility of the TimeOfDay class. To preserve class responsibility we return to the timeOfDay class and add these two methods:

```
Accessor calcElapsedTime  IMPORT None  EXPORT elapsed

elapsed = hours.getHours x 60 + minutes.getMinutes

Accessor lessThan  IMPORT otherTime (TimeOfDay)
                   EXPORT isLess
thisElapsed = calcElapsedTime
isLess = false
IF thisElapsed < otherTime.calcElapsedTime THEN
   isLess = true
ENDIF
```

The addition of these two accessors to the TimeOfDay class illustrate a number of significant points:

- They are both accessors which export information which is derived from the class fields but is not simply a copy of a class field.
- Neither starts with *get*.
- The need for them was not known at the time the TimeOfDay class was

developed. Although good software engineering practice will minimise the need for adding or modifying class designs after they have been initially completed, this type of situation cannot be avoided. We could have avoided it by putting the sub modules with main but violating class responsibility will cause long term pain whereas this way only adds a small, on time only overhead.

Allowing for our revised TimeOfDay class (with the two new accessors incorporated into it), our final version of the calcTimeDifference sub module is:

```
Sub Module calcTimeDifference
IMPORT time1, time2 (TimeOfDay)
EXPORT difference

IF time1.lessThan<--time2 THEN
   difference = time1.calcElapsedTime -
                time2.calcElapsedTime
ELSE
   difference = time2.calcElapsedTime -
                time1.calcElapsedTime
ENDIF
```

Note that, because our task is so simple, we also end up with considering whether or not the calcTimeDifference sub module should also be moved into the TimeOfDay class. The argument for this is, of course, class responsibility. The argument against is one of practicality. The TimeOfDay class is pretty generic and so, once developed, is quite likely to be used in many different software applications. If we put everything to do with time that all the applications require then, over time (pun not intended), the TimeOfDay class will grow to include hundreds of methods. A large number of those methods will only be used in one application. Hence, this issue is not so clear cut in the real world. The Inheritance class relationship could be used to resolve this issue but we will leave that discussion for the next chapter.

Complexity through Aggregation

Note that in the TimeOfDay example, the total amount of algorithm for the first version involved only 1 class and a whole lot less code. This may start you wondering whether or not OO is really worth the effort. It may surprise you to learn that sometimes it isn't! OO helps us enormously when the system under development is large and its required functionality is complex. Recently I had to implement a standard algorithm called the Travelling Salesman algorithm on a PDA. I had the choice of a non OO design and implementation using the C programming language or an OO design and implementation using the Java programming language. The algorithm was small and pretty straight forward and the software concerned was never going to be incorporated into anything larger. I chose a non OO approach in C because, in that situation, an OO approach would not provide me with any advantages and would have higher design and implementation overheads. However that type of situation is almost never going to come up in a commercial software development situation. If the software is small and simple then the clients will probably

write it themselves using a simple non-OO programming language. The point is keep in mind that OO pays off when the software is large and complex. However as first year students you are absolutely not ready for large and complex so all of the examples presented in this book are relatively small and simple. A good analogy would be the difference between learning to fly light aircraft and learning to fly a jet fighter. In both cases the initial flight training would occur in a small, simple light aircraft but the fighter pilot training would be more rigorous and involve more complex techniques which will become useful in later training. The light aircraft pilot will never need to know these techniques so his training will be more simple and straight forward.

Relating class through aggregation can produce extremely elegant and easy to understand code while at the same time achieving great complexity. For example, suppose we had a class for representing a continent as well as a class for representing bodies of water. We could then have a class for describing a planet which had class fields for describing all of its continents and bodies of water. In other words aggregation is used to distribute the information describing the planet amongst continent and bodies of water objects. We could then represent a solar system with a class which used planet objects to describe the planets in the solar system. We would also have a class for describing the star(s) for that solar system. Again aggregation is being used to encapsulate the information into appropriate objects. Next we could describe a galaxy using a class which used solar system objects to describe the solar systems contained within it. Finally we could have a class which using galaxy objects to describe the universe. We could then construct a representation of the known universe by constructing one universe class object. This one constructor call will trigger all the other constructor calls. The result is that we can reference anything in the universe via one object that we created! Feeling *God like* yet?

Conclusion

In this chapter we have dealt with the simplest class relationship. i.e. aggregation.. Hopefully you are starting to see how objects of different classes can work together to produce the required functionality. Aggregation across a large number of classes can produce incredibly complex and powerful functionality while at the same time encapsulating sections of algorithm into intuitive collections (i.e. classes). This makes the resulting algorithm easier to understand and modify. The second class relationship we will discuss is called Inheritance. It is even more powerful than aggregation. However it is also a lot more subtle and great care must be taken to ensure that it is used appropriately.

Chapter 8

"The Universe speaks in many languages, but only one voice.
...
It is the voice of our ancestors speaking through us.
And the voice of our inheritors waiting to be born.
It is the small, still voice that says we are One."
G'Kar, The text of the second draft of the Interstellar Alliance Declaration of Principles, Babylon 5.

The Inheritance Class Relationship.

Introduction

We come at last to the most significant aspect of object orientation: The Inheritance class relationship. The basic goal of inheritance is to maximise code re-use and minimise code repetition. As an example, suppose we had a pre-existing class for managing dates. This is a generic class intended to handle all types of dates. Now consider what we would do if we needed a class to manage birth dates. The overall functionality would be the same but there would be differences. For example birth dates would always be in the past but not too far in the past (i.e. nobody lives to be 1000 years old!). Without inheritance we would have to create a DateOfBirth class by copying the code from the original Date class and then modifying it to suite the needs of maintaining birth dates. Most of the code from both classes would be identical but there would be differences. This is exactly the situation which causes nightmares when maintaining software. Corrections and updates need to be made to both classes but the difference in the role of each class needs to be kept in mind. This is a recipe for disaster. The use of inheritance would allow us to develop a DateOfBirth class which inherits from the original Date class. This would mean that ONLY the code that was different to the original Date class would need to be placed in the DateOfBirth class. Anything in the DateOfBirth class which was the same as the original class is automatically assumed and does not need to be repeated. This means there is no repetition of code across classes and, because the DateOfBirth class only contains the code that is different, it is much less likely for errors to occur and the overheads of maintaining the code are minimised.

Again, I remind you that object orientation is only beneficial to large, commercial software systems. We will only be looking at small collections of classes related by inheritance but you need to remember the impact of this on a large system.

In principle, inheritance is all good news. In reality there are negatives and overheads which need to be considered. We will discuss these issues as we look at inheritance in more depth but for the moment we need to understand the terminology and the basic mechanics of how inheritance is implemented in code.

Terminology

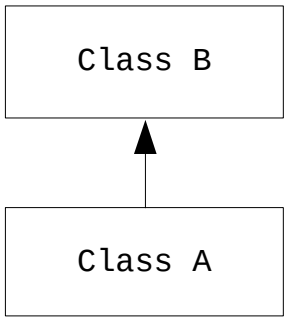
Inheritance is a more complex class relationship than aggregation and so there is some terminology that needs to be understood. For the moment don't get to obsessed with exactly what each term means and what the purpose of it is. We will flesh out the details as the chapter progresses. For the moment you just need to get a rough idea.

If class A inherits from class B Then:

- Class A automatically has all of the functionality defined in class B.
- Class A has some knowledge of class B.
- Class B has no knowledge of class A.
- Class B is the super class of class A. Also known as the ancestor of class A.
- Class A is the sub class of class B. Also known as the child class of class B.

A class which has no sub classes is called a leaf class or concrete class. Normally (though not always) a sub class has only one super class. This is known as *Single Inheritance*. When a sub class has more than one super class, the relationship is known as *Multiple Inheritance*. Multiple Inheritance is a can of worms all by itself and so, given that this book is intended for first year, we will not be discussing it in detail.

Under UML we model inheritance with an arrow which points from the sub class towards the super class.



The Base Class

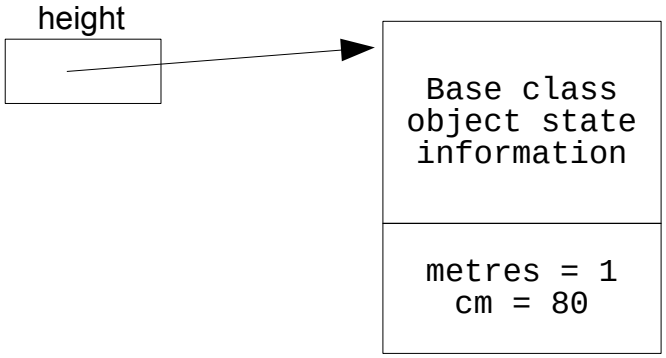
In any object oriented programming language there is only one class which does not inherit from any other class. It is known as the *Base Class*. If class A is not the base class and it's super class is not stated, then the implication is that class A inherits from the base class. The base class is responsible for

looking after all of the low level requirements of placing and maintaining an object in memory (e.g. calculating the size and allocating the memory required, etc.). This type of house keeping is the same for all objects of all classes so it makes sense to capture it in one class and have all other class, directly or indirectly inherit it from it. This means that every time we construct an object of a particular class we automatically get all of the base class functionality. Otherwise this housekeeping code would have to be copied into every class we designed! The other advantage to the base class is that the details of what happens are hidden from the rest of the classes. This is a good thing because we don't need to know and we don't want to deal with it. We are concerned with ensuring our classes fulfil their intended roles and we don't want to waste time writing the same code over and over again.

This means that every time we construct an object of a class that we have designed, we are, not only executing the constructors for our class, we are also executing the base class constructors. This means that the resulting object will contain the object state of our class as well as the object state of a base class object. Normally we refer to this combined object state as one object but, for now, its probably better for you to think of them as two objects that have been created and glued together. For example, remember the Height class example from chapter 6. It had class fields of metres and cm. When a Height class object is constructed, for example:

```
construct height using 1, 80
```

The first thing that the Height constructor will do is call the default constructor for the base class. We never put that constructor call in our algorithm for any of the Height class constructors so how did this happen? It happened automatically. The compiler for whatever OO language you are using knows a base class object has to be constructed so it inserted a line of code calling the base constructor into the Height class constructors as it compiled them. That way we end up with the height variable referring to an object which was composed of the object state of a base class object as well as a Height class object containing values of 1m and 80 cm. The diagram below illustrates what I mean:

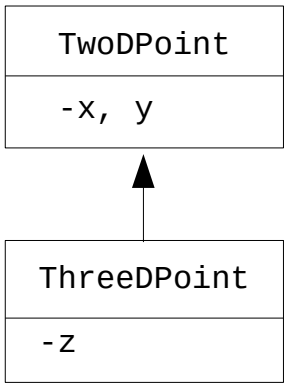


Note that we have not discussed exactly what is contained within a base class object. The good news is that we don't really care! We simply assume that whatever is there is whatever is required to support an object in memory and

leave it at that. Final year subjects on programming languages and compiler design would be interested in what is inside a base class object and how it got there but we are not. However, now that we are tangling with inheritance we can never afford to forget that the base class object is there and must be constructed first.

A Simple Example

Let's look at a simple inheritance example before we wade in any deeper. Suppose we wanted classes to represent points in 2 and 3D Cartesian space. We will have two classes, one for 2D points and one for 3D points. The 2D class will keep track of x and y values and the 3D class will look after x, y and z values. If you think about it the 3D class has the same data and functionality as the 2D class for x and y but in addition it also has the z value and related functionality. This is an ideal situation for inheritance because the 3D class needs the all of the data and functionality as the 2D class plus a bit extra. If we developed the classes without inheritance we would have the 2D code repeated in both classes. What we can do now is develop the 2D class and then extend its functionality in the 3D class by adding the z value and related functionality.



The 2D class is straight forward, its basically along the lines presented in chapter 6. We don't even need to worry about validation because all values (negative, positive, zero) are valid for coordinate values.

```
CLASS TwoDPoint
Class Fields: x, y

Default constructor IMPORT None
x = 0
y = 0

Alternate Constructor IMPORT inX, inY
x = inX
y = inY

Copy Constructor IMPORT inPoint (TwoDPoint)
x = inPoint.getX
y = inPoint.getY
```

```
Mutator setX IMPORT inX EXPORT None
x = inX

Mutator setY IMPORT inY EXPORT None
y = inY

Mutator setPoint IMPORT inX, inY  EXPORT None
x = inX
y = inY

Accessor getX IMPORT None  EXPORT x

Accessor getY IMPORT None  EXPORT y

Accessor equals IMPORT inPoint (TwoDPoint) EXPORT same

IF x = inPoint.getX AND y = inPoint.getY THEN
   same = true
ELSE
   same = false
ENDIF

Accessor toString  IMPORT None EXPORT stateStr

stateStr = "X = ", x, " Y = ", y
```

So, as can be seen from the above example, there is nothing extra in the super class. For all intents and purposes it looks like your average, garden variety container class. This is quite typical for super classes. There is one exception which is connected with abstract classes and abstract methods. What the heck is an abstract class or an abstract method you ask? For the moment forget I mentioned them. We will discuss them last.

The sub class will look very different and the inheritance will manifest itself throughout the class. The main thing to consider is that the code outside of these two classes should not need to concern itself with which class looks after x and y and which class looks after z. It should be possible to treat a ThreeDPoint object as if it contains x, y and z. This means that the algorithms for the methods in ThreeDPoint class have to cater to this. The communication between ThreeDPoint and TwoDPoint classes will be via super class method calls. In other words the communication is always from the sub class to the super class. It is never the other way around. The word super is used in all OO languages to refer to something in the super class. When used by itself this means a reference to a super class constructor. When used with a super class method name then this means you are calling that particular super class method (e.g. super.getX). Lets have a look at ThreeDPoint class and then discuss each method to highlight the inheritance issues.

```
CLASS ThreeDPoint INHERITS FROM TwoDPoint
Class Field: z

Default Constructor:  IMPORT None
super
z = 0

Alternate Constructor IMPORT inX, inY, inZ
super<--inX, inY
z = inZ

Copy Constructor  IMPORT inPoint( ThreeDPoint)
super<--inPoint
z = inPoint.getZ

Mutator setZ  IMPORT inZ  EXPORT None
z = inZ

Mutator setPoint  IMPORT inX, inY, inZ  EXPORT NOne
super.setPoint<--inX, inY
z = inZ

Accessor getZ  IMPORT None  EXPORT z

Accessor equals  IMPORT inPoint (ThreeDPoint) EXPORT same

IF z = inPoint.getZ AND super.equals<--inPoint THEN
   same = true
ELSE
   same = false
ENDIF

Accessor toString  IMPORT None  EXPORT stateStr

stateStr = super.toString, " Z = ", z
```

Note that everything in ThreeDPoint class which is related to inheritance and the super class is represented in bold. Let's now examine the inheritance embodied in the above class in more detail.

Inheritance and Object Construction.

Before we look at the above code in more detail, we need to understand what we are creating in memory we we construct a ThreeDPoint object. Each ThreeDPoint object will consists of the object states of three objects:

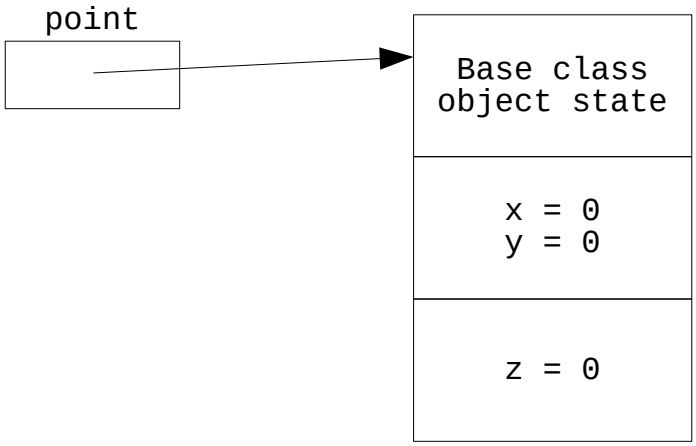
- The Base class
- TwoDPoint (Containing x and y).
- ThreeDPoint (containing z).

This means that when we construct a ThreeDPoint object we trigger constructor calls for TwoDPoint class which in turn calls the base class default constructor.

Hence if we had the following pseudo code in some other part of a software application:

```
construct point using defaults
```

We would create an object that looks like:



Outside of these two classes we can ignore where the information has been placed inside the point object. For example a call to point.getX doesn't need to know where x or the getX accessor have been defined (i.e. in TwoDpoint or ThreeDPoint). The inheritance relationship will ensure that the method is located and executed (we will re-visit this issue a bit further on). Starting with constructors we need to have each sub class constructor call the appropriate super class constructor. If we do not specify a super class constructor call then the compiler will automatically insert a call to the super class default constructor as the first line of the sub class constructor. The super class constructor call has to be the first line of the sub class constructor call because we want the order of object construction to be from the base class down to the sub class. Starting to feel a tad light headed yet? Don't panic. Lets look at what happens when we construct a ThreeDPoint class object using a default constructor. The pseudo code for the ThreeDPoint default constructor is:

```
Default Constructor:  IMPORT None
super
z = 0
```

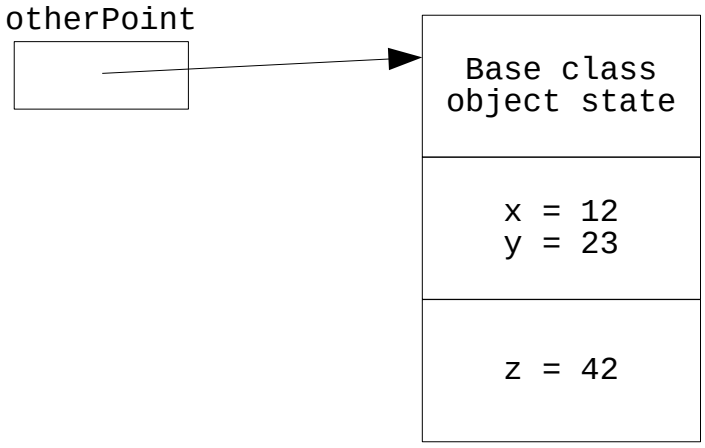
The word super is a call to the super class default constructor. If we had left that line out then the compiler would have inserted it anyway because, in order to construct a ThreeDPoint class object we have to construct a TwoDPoint class object. This means we have to call a TwoDPoint class constructor. Hence, if we don't then the compiler will, otherwise the code cannot function. So, the first thing that happens is that control is passed up to the TwoDPoint default constructor. For exactly the same reasons, the first thing that happens in the TwoDPoint class constructor is to call the default base class constructor. The base class constructor creates the base class object and initialises its state before returning control to the TwoDPoint class constructor. The TwoDPoint class constructor initialises x and y to zero and

then passes control back to the ThreeDPoint class constructor which sets z to zero. For the code that called the ThreeDPoint class constructor it appears that one, ThreeDPOPint class object has been constructed. However, what has really occurred is that three objects have been constructed (base class, TwoDPoint class and ThreeDPoint class) and combined into what appears as one object to the code that called the ThreeDPoint constructor.

Exactly the same process occurs when using the alternate constructor except that we don't want to call the default super class constructor. We want to call the alternate super class constructor. Suppose we created a ThreeDPoint object using the pseudo code below:

```
construct otherPoint using 12, 23, 42
```

The resulting object would look like:



Note that when calling the ThreeDPoint class alternate constructor we can ignore the details related to inheritance. We simply feed in the x, y and z values and ask it to generate the ThreeDPoint class object. This means that the IMPORT to the ThreeDPoint alternate constructor has to IMPORT values for x, y and z, pass the x and y values up to the super class alternate constructor and then using the z IMPORT to initialise its z class field:

```
Alternate Constructor IMPORT inX, inY, inZ
super<--inX, inY
z = inZ
```

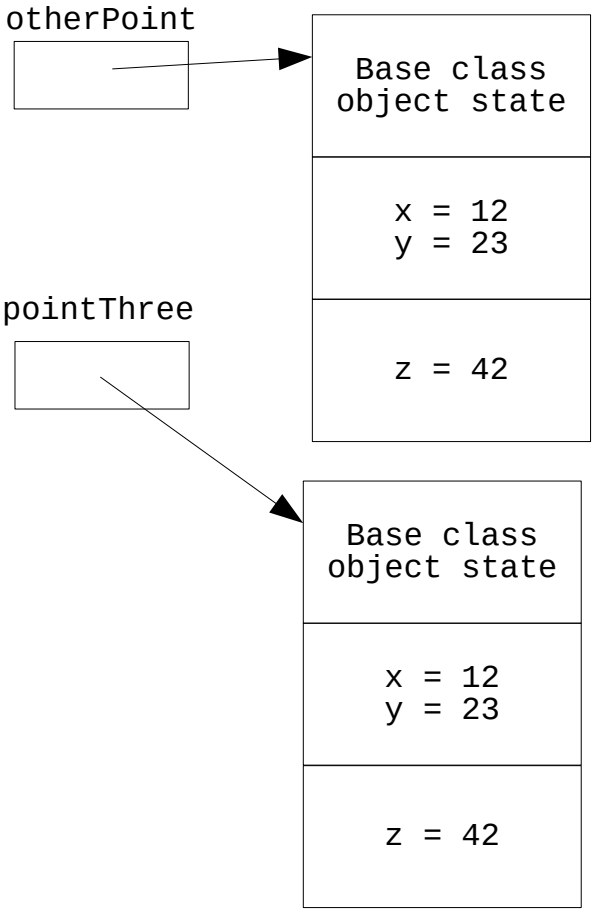
We must put the call to the alternate super class constructor because if we do not, then the compiler will automatically insert a call to the super class default constructor. The effect will be that x and y will be set to zero instead of the values in inX and inY. So, in order to ensure that the object has inX and inY for its x and y values, we have to make sure that the ThreeDPoint alternate constructor calls the TwoDPoint alternate constructor instead of the default constructor. We can see that from the code outside of the inheritance related classes, the inheritance seems automatic and largely invisible. To create that illusion we must ensure that we get the required super/ sub class functionality

to achieve the desired result.

The copy constructor is a little bit trickier, although it does involve the same idea. For example suppose we used our otherPoint object to create another object with the same x, y and z values by using a copy constructor:

```
construct otherPoint using 12, 23 and 42
construct pointThree using otherPoint
```

would result in the following objects:



The algorithm for the copy constructor for ThreeDPoint class is:

```
Copy Constructor  IMPORT inPoint( ThreeDPoint)
super<--inPoint
z = inPoint.getZ
```

The idea is exactly the same as with the alternate constructor, call the super class copy constructor which will extract x and y from the object and then extract the z value from the IMPORT object and use it to initialise z. The only issue is that the super class copy constructor is expecting its IMPORT to be a TwoDPoint class object. Here in the ThreeDPoint copy constructor we are calling the TwoDPoint class copy constructor and feeding it a ThreeDPoint class object rather than a TwoDPoint class object. At first this seems like it

should not work. However, if you look at the object diagrams for a ThreeDPoint class object, it has the TwoDPoint class object information as well as the ThreeDPoint class object information. What will happen is that the copy constructor for TwoDPoint class will accept the incoming ThreeDPoint class object and treat it as if it was a TwoDPoint class object by simply ignoring all of the ThreeDPoint class information.

This is a very important point which will come up often when inheritance is involved. Any time we have a variable specified to be an object of class X, we can initialise it as an object of Class X OR as an object of *any* of the descendant (i.e. sub) classes. We can get away with this because we know that a sub class object will always contain the state information for all of its ancestor classes. However the object variable will not be able to access any of the sub class information.

Preserving Class Responsibility.

We do not have to define getX/setX and getY/setY accessors and mutators in ThreeDPoint class because they already exist in the super class. Note the setPoint mutator IMPORTs values for x, y and z, calls the super class setPoint mutator handing it the x and y values and then updates the z value. Again from code outside of both classes, the distinction between where the x and y values end up and where the z value ends up is invisible. The use of super in the setPoint call (i.e. super.setPoint<--inX, inY) is explicitly stating that it is the super class setPoint mutator which we want to call.

Notice in the toString and equals methods we do not repeat the functionality in their super class equivalents. We call the super class equivalents and simply add the functionality related to z. For example in the toString accessor we have:

```
Accessor toString  IMPORT None  EXPORT stateStr

stateStr = super.toString, " Z = ", z
```

The call to super.toString will EXPORT back a string containing x and y and then the z information is added to it so that the resulting stateStr will list the x, y and z values. Similarly the equals method compares the z values and then calls the super class equals method to compare the x and y values. Notice that we have the same issues with handing a ThreeDPoint class object to a method which is expecting a TwoDPoint class object but the same principle applies. The TwoDPoint class equals method will accept the incoming ThreeDPoint class object and treat it as if it is a TwoDPoint class object by simply ignoring the ThreeDPoint class state and functionality.

At no point in any method in ThreeDPoint class have we included code which directly manages x and y because that is the responsibility of TwoDPoint class. This actually means that the ThreeDPoint class methods are easier to write because every time we need to deal with x an y we simply pass the burden on up to TwoDPoint class by calling the appropriate method. The

overall effect is that we don't have algorithms repeated across the classes, the overhead in developing the sub class is restricted to simply dealing with the extra functionality and the sections of the code which use objects of the sub class can remain blissfully ignorant of all the inheritance related issues.

Lets look at a simple algorithm that deals with 3D points to illustrate this. Suppose we wanted to input two 3D points and, if they are different, calculate the distance between them. Our main algorithm would look something like:

```
MAIN
INPUT x, y and z
construct point1 using x, y and z

INPUT x, y and z
construct point2 using x, y and z

IF point1.equals<--point2 THEN
   OUTPUT "The two points have the same location"
ELSE
   dist = calcDistance<--point1, point2
   OUTPUT "The distance between ", point1.toString,
          " and ", point2.toString, " is ", dist
ENDIF

Sub Module calcDistance
IMPORT point1, point2 (ThreeDPoint)
xSqr = (point1.getX - point2.getX)²
ySqr = (point1.getY - point2.getY)²
zSqr = (point1.getZ - point2.getZ)²

dist =  √(xSqr + ySqr + zSqr)
```

Notice that point1 and point2 are treated as if they each contain x, y and z. There is no allowance for what specified in the TwoDpoint or ThreeDPoint classes. In other words, from outside the classes related by inheritance, the which of the inheritance related classes contains what is irrelevant. This is exactly how we want things to be. It ensures high cohesion and low coupling everywhere except between the two inheritance related classes.

The above example shows the basics but what happens when there is more than one sub class? The answer is the same process applies but the more sub classes then the larger the benefit we gain from using inheritance.

Inheritance: The *Kind Of* Relationship.

Now that we have a basic idea of the mechanics, lets revisit the intent of Inheritance. We have discussed some of the benefits and now we will look at when it is or isn't appropriate to use inheritance. If you think about it, both aggregation and inheritance are mechanisms which combine objects of different classes together to form a combined entity. The major difference is the form of communication that takes place between the objects concerned.

Aggregation enables communication between an object and other objects which are declared as its class fields whereas inheritance is enabling communication between the super and sub class. In both cases the communication is in one direction. With aggregation its from the object to the object it contains via the class field and with inheritance its from the sub class object to the super class object. This means that whenever we want objects of two classes to be able to communicate in this way, we have a choice between the two relationships. Both will be possible, the decision will be driven by which one results in the most maintainable and understandable code.

As previously described, aggregation is a *Contains* relationship. If class A is aggregated to class B then:

- Objects of class A are declared as class fields within class B.
- The state of class A is partially composed of objects containing the state information specified in class B.

Inheritance is a *Kind Of* relationship. What this means is that each sub class should represent a different, and more specialised, version of the super class. For example, suppose we had a super class for representing 2D shapes. We might then have a collection of sub classes for each specific 2D shape (e.g. circle, square, triangle). Each sub class is a *kind of* shape and each is more specialised.

This means that when considering whether to use inheritance we need two criteria to be satisfied:

1. There must be some functionality and data which is common to all the classes. This common functionality and data would be placed in the super class.
2. Each sub class must represent a more specialised version of the super class.

If you think about our 2D and 3D point example, the x/y functionality and data is common to both and therefore placed in the super class. Mathematicians might argue with me but, from a software design perspective, a 3D point is more specialised than a 2D point because it has extra data (i.e. the z value) and extra functionality.

Another Example

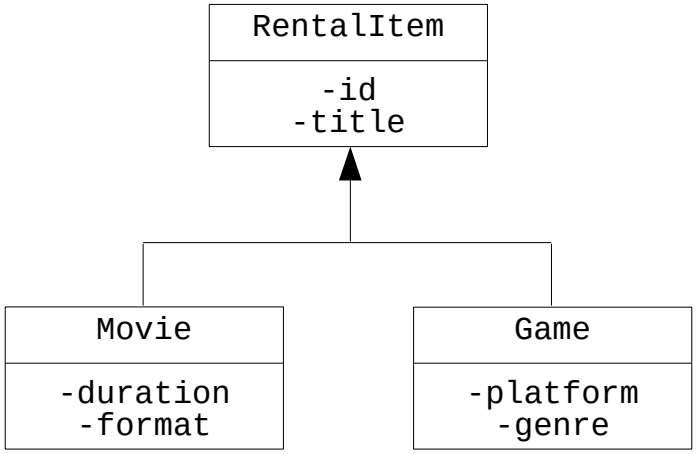
Suppose we were developing software for a video library rental system. We would need a collection of classes to represent the information about each rental item. Our fictitious rental library rents out movies and computer games. The information describing a movie is:

- Identity number
- Title
- Duration (minutes).
- Disk type (DVD or Blu-Ray).

The information describing a computer game is:

- Identity number
- Title
- Platform (e.g. PlayStation 3).
- Genre (e.g. First Person Shooter).

The first thing we see is that computer games and movies are things that the video library rents out to customers. They are both different types of rental items. Secondly we can see that there is some information which is common to both (i.e. identity number and title). Inheritance is an ideal choice here because we have commonality (which is placed in the super class) and specialisation (movies and computer games are more specialised versions of rental items). At this point we can map out the classes and the class fields for each class:



As with our 2D and 3D points example, the super class will look like an ordinary container class and each sub class will only contain the extra information. the id has to be an integer in the range 1000 to 9999, the duration of a movie must be positive and all the other information is represented via character strings and cannot be validated. Also, many OO languages have a class for looking after character strings. We will assume that in our case, there is a pre-existing class called String which is used for this purpose. We will also assume that the String class has all the usual stuff (i.e. normal constructors, equals method etc).

```
CLASS RentalItem
Class Fields:  id, title.
Class Constants: MINID=1000, MAXID = 9999

Default Constructor IMPORT None
id = MINID
construct title using default
```

```
Alternate Constructor  IMPORT inId, inTitle
IF MINID <= inId <= MAXID THEN
   construct title using inTitle
   id    = inId
ELSE
   id = MINID
   construct title using default
ENDIF

Copy Constructor IMPORT inRental (RentalItem)
id    = inRental.getId
title = inRental.getTitle

Mutator setID  IMPORT inId  EXPORT None
IF MINID <= inId <= MAXID THEN
   id    = inId
ENDIF

Mutator setTitle  IMPORT inTitle  EXPORT None
construct title using inTitle

Mutator setRental  IMPORT inId, inTitle
IF MINID <= inId <= MAXID THEN
   construct title using inTitle
   id = inId
ENDIF

Accessor getId  IMPORT None  EXPORT id

Accessor getTitle  IMPORT None  EXPORT title

Accessor toString  IMPORT None EXPORT stateStr
stateStr = "Id = ", id, "  title = ", title

Accessor equals  IMPORT inRental (RentalItem) EXPORT same
IF id = inRental.getId AND
   title.equals<--inRental.getTitle THEN
   same = true
ELSE
   same = false
ENDIF
```

Note that the super class looks just like an ordinary container class. All of the work in supporting the inheritance happens in the sub classes so let's have a look at them now.

```
Class Movie INHERITS FROM RentalItem
Class Fields: duration, format

Default Constructor  IMPORT None
super
duration = 90
format = "DVD"

Alternate Constructor
IMPORT inId, inTitle, inDuration, inFormat
super<--inId, inTitle
IF inDuration <= 0 THEN
   duration = 90
ELSE
   duration = inDuration
ENDIF
construct format using inFormat

Copy Constructor  IMPORT inMovie (Movie)
super<--inMovie
duration = inMNovie.getDuration
format = inMovie.getFormat

Mutator setDuration  IMPORT inDuration  EXPORT None
IF inDuration > 0 THEN
   duration = inDuration
ENDIF

Mutator setFormat  IMPORT inFormat  EXPORT None
construct format using inFormat

Mutator setMovie
IMPORT inId, inTitle, inDuration, inFormat
super.setRental<--inId, inTitle
IF inDuration > 0 THEN
   duration = inDuration
ENDIF
construct format using inFormat

Accessor getDuration  IMPORT None  EXPORT duration
Accessor getFormat  IMPORT None  EXPORT  format
Accessor toString  IMPORT None  EXPORT stateStr

stateStr = super.toString, " duration = " duration,
           " format = " format

Accessor equals IMPORT inMovie (Movie)  EXPORT same
IF super.equals<--inMovie AND
   duration = inMovie.getDuration AND
   format.equals<--inMovie.getFormat THEN
   same = true
ELSE
   same = false
ENDIF
```

The Movie class looks a lot like the 3DPoint class because it has the same issues in ensuring the inheritance is going to work the way we want to. However there is another issue related to validating the IMPORT which has resulted in a flawed design. The issue is some of the IMPORT (id) is validated in the super class and some (duration) is validated in the sub class. The problem being is that the super class has no way of communicating to the sub class whether or not the IMPORT turned out to be valid or not. This is important because if some of the IMPORT to either the alternate constructor or the setMovie mutator is invalid, then the other IMPORT will be used and the invalid IMPORT will not. What we want is that the IMPORT to be used only if ALL the IMPORT is valid. To enable this we need to go back and modify the super class by adding an imperative method which IMPORTs an id and EXPORTs a boolean specifying whether or not the IMPORT is valid. We would then modify the alternate constructor and some of the mutators in RentalItem to use that method for validation. The revised version of the class is below. All of the modified lines are in bold.

```
CLASS RentalItem
Class Fields:  id, title.
Class Constants: MINID=1000, MAXID = 9999

Default Constructor IMPORT None
id = MINID
construct title using default

Alternate Constructor  IMPORT inId, inTitle
IF validateID<--inId THEN
   construct title using inTitle
   id    = inId
ELSE
   id = MINID
   construct title using default
ENDIF

Copy Constructor IMPORT inRental (RentalItem)
id = inRental.getId
title = inRental.getTitle

Mutator setID  IMPORT inId  EXPORT None
IF validateID<--inId THEN
   id    = inId
ENDIF

Mutator setTitle  IMPORT inTitle  EXPORT None
construct title using inTitle

Mutator setRental  IMPORT inId, inTitle
IF validateID<--inId THEN
   construct title using inTitle
   id    = inId
ENDIF

Accessor getId  IMPORT None  EXPORT id
```

```
Accessor getTitle  IMPORT None  EXPORT title

Accessor toString  IMPORT None EXPORT stateStr
stateStr = "Id = ", id, "  title = ", title

Accessor equals  IMPORT inRental (RentalItem) EXPORT same
IF id = inRental.getId AND
   title.equals<--inRental.getTitle THEN
   same = true
ELSE
   same = false
ENDIF

PROTECTED Imperative validateID
IMPORT inId  EXPORT isValid
IF MINID <= inId <= MAXID THEN
   isValid = true
ELSE
   isValid = false
ENDIF
```

Note the visibility category protected is a halfway house between public and private. It means that the method is public to all descendant classes of RentalItem and private to all other classes. This way we can refer to it in the sub classes and, at the same time, hide it from all other classes. Now we can modify the Movie class to call the validateID method whenever it needs to. Hence we end up with (again the modifications are on bold):

```
Class Movie INHERITS FROM RentalItem
Class Fields: duration, format

Default Constructor  IMPORT None
super
duration = 90
construct format using "DVD"

Alternate Constructor
IMPORT inId, inTitle, inDuration, inFormat
super<--inId, inTitle
IF super.validateID<--inId AND inDuration > 0 THEN
   duration = inDuration
   construct format using inFormat
ELSE
   duration = 90
   construct format using "DVD"
   super.setRental<--super.MINID, "No Title"
ENDIF

Copy Constructor  IMPORT inMovie (Movie)
super<--inMovie
duration = inMNovie.getDuration
format = inMovie.getFormat
```

```
Mutator setDuration  IMPORT inDuration  EXPORT None
IF inDuration > 0 THEN
   duration = inDuration
ENDIF

Mutator setFormat  IMPORT inFormat  EXPORT None
construct format using inFormat

Mutator setMovie
IMPORT inId, inTitle, inDuration, inFormat
IF super.validateID<--inId AND inDuration > 0 THEN
   super.setRental<--inId, inTitle
   duration = inDuration
   construct format using inFormat
ENDIF

Accessor getDuration  IMPORT None  EXPORT duration
Accessor getFormat  IMPORT None  EXPORT  format
Accessor toString  IMPORT None  EXPORT stateStr

stateStr = super.toString, " duration = " duration,
           " format = " format

Accessor equals IMPORT inMovie (Movie)  EXPORT same
IF super.equals<--inMovie AND
   duration = inMovie.getDuration AND
   format.equals<--inMovie.getFormat THEN
   same = true
ELSE
   same = false
ENDIF
```

Note that we have still preserved class responsibility. The Movie class never directly deals with tasks which are the responsibility of the RentalItem class. The Game sub class involves exactly the same concepts and issues. Try your hand at developing it yourself. If you can do it without any problems then you most likely understand inheritance so far. if you have trouble then go back over the chapter and try to discover what the problem is.

DO NOT PROCEED until you have verified that you are happy with all of the concepts so far.

Method Overloading, Method Overriding and Polymorphism.
We have seen in some of our examples situations where we have different versions of the same method (e.g. setPoint in TwoDPoint and setPoint in ThreeDPoint). In previous chapters, we discussed the concept of a sub module (or method) signature as being the name of the sub module and its IMPORT/EXPORT.

Method Overloading is where we have multiple versions of the same method defined within the same class. When the method is called, the version whose signature matches the call is the version used.

Method Overriding is where we have multiple versions of the same method spread across a collection of classes related by inheritance. In these cases the method signature and the search order will determine which method is invoked. The search for a method with a matching signature will start in the class that the object has been constructed from. If a match is not found then the search will move up to its super class. The search will continue until either a match is found or the search has progressed all the way up to the base class and a match has not been found. Note that in this situation the sub class method can have the same signature as the super class method. If it does then we will always end up with the sub class method being the one invoked. i.e. the sub class method has over ridden the super class method.

Mostly the determination of which method is being invoked is resolved at compile time. This is because, most of the time, the compiler can determine which class the object is by analysing the coding and verifying which constructor was called.

However sometimes this is not possible. Remember from our previous discussion, we know that an object variable which has been declared as being of a particular class can be initialised as an object of that class or as an object of any descendant of that class. In these situations the compiler may not be able to tell which class in the inheritance hierarchy the object has been constructed from. This means that the method matching must be done as the program is executing instead of as it is being compiled. We call this *polymorphism*. For example suppose we had a super class called Shape and the sub classes triangle, rectangle and circle. Suppose we had a main algorithm as below:

```
MAIN

INPUT shapeType
IF shapeType = "c" THEN
   construct shape using circle default constructor
ELSE IF shapeType = "r" THEN
   construct shape using rectangle default constructor
ELSE IF shapeType = "t" THEN
   construct shape using triangle default constructor
ENDIF
Rest of algorithm
```

In the above main algorithm we don't know which class will construct the object until we have the user input. This means that any method overloading has to resolved at run time. This is polymorphism. At this point the only really important issue is that you understand that the search for a matching method signature proceeds from sub class to super class and stops as soon as it finds a match.

Abstract Classes.

At an introductory level, abstract classes can be a bit of a struggle to get a handle on so this section will first describe what defines an abstract class and then we will talk about why and how they are used. Again, I remind you that, like most OO features, abstract classes are only really useful in a large, complex software system. However many would argue that in a large, complex OO system, abstract classes are an essential design tool. For now though, let's simply look at what an abstract class is and how it works.

An abstract class is a super class. What differentiates it from an ordinary super class is that it does not contain enough information and functionality to be useful by itself. An abstract class only becomes useful when combined with a sub class. This means that objects created from abstract classes never exist by themselves in the software system as it executes. Abstract class objects only exist as part of the object state of a sub class object. An easier way to think of it is that the constructors for an abstract class are only ever called by sub class constructors. In other words you only ever construct an abstract class object as part of the process of constructing a sub class object.

In the 2D and 3D point example, the super class (TwoDPoint) was not an abstract class because a TwoDPoint class object is a useful thing by itself. It can be used to represent a 2D Cartesian point without any help from its sub class. The RentalItem class from the video library example is an abstract class because RentalItem objects by themselves cannot serve a useful purpose in the software system. They are only useful when combined with sub class objects (i.e. Movie or Game objects). All OO languages have a means of specifying whether or not a class is an abstract class. In C++ this is an exercise in torture. In others, such as Java, it is simply a matter of including the word *abstract* in the class header. Whatever the OO language, the compiler will not allow constructor calls to an abstract class constructor anywhere but as the first line of a sub class constructor. This prevents accidental constructor calls and is the whole point of going to the trouble of specifying the class as abstract in the first place.

What is the fuss about? You might be wondering why I am even bothering to tell you if that is the only difference. Once again you have to try to see beyond first year and you alone building three or four classes. Imagine yourself as one of a team of Software Engineers who are spending months developing hundreds of classes. Given that these classes will share hundreds (possible thousands) of class relationships we need strategies to keep things organised, intuitive and to minimise repetition of code etc.

Remember that we said that when we have a collection of similar classes where some of the data and functionality is common, we can extract the common data and functionality up into a super class. Often this commonality is not useful by itself and must be combined with the sub class data and functionality to serve a useful purpose in the software system. Remember also that we can always initialise a super class object variable with a sub class object. In those situations we usually expecting the object variable to be initialised as an object of one of the sub classes and do not want a super class object. Abstract classes give us a mechanism for ensuring this cannot happen. Think about the shape example from the polymorphism discussion, The main algorithm is assuming that the shape object variable is always going to be an object created by one of the sub classes and will never simply exist as a Shape object. i.e. the object variable will be declared as an object of the super class but the algorithm expects that it will always be constructed as one of the sub class objects.

So, at this level, that is all there is to an abstract class. A design decision is made regarding whether or not its appropriate for objects of a super class to exist on their own or not. If they can then the class is not an abstract class whereas if they shouldn't then the class should be defined as an abstract class. This means that TwoDPoint class should not be declared as an abstract class but the RentalItem class should (see if you can figure out why).

Abstract Methods

Abstract methods are another design tool that can be used with abstract classes. We will look at mechanically what is an abstract method, then we will discuss why the heck we would bother with them. Finally we will illustrate the concept with a simple example.

An abstract method:

- Can only be defined in an abstract class.
- Consists of only a specification that:
    - It is an abstract method
    - A signature for the method.
    - Does not have an algorithm or body attached to it. i.e. the method is not implemented.

If an abstract class has an abstract method defined in it then, somewhere in the inheritance chain between the abstract class and the leaf class, a full implementation of the method must be made. This implementation must match the signature specified in the abstract method.

For example, with the Shape class and its descendants, if we defined an abstract method called calcArea in the Shape class which had no IMPORT and had a real number as EXPORT then all of the sub classes of shape would have to have an implementation of the calcArea method which had no IMPORT and had an real number as EXPORT.

The mechanics of this will become more clear when we look at an example. before we do that lets consider why we would want to play with abstract methods at all (especially considering that, at this point you might be feeling about the same as you do when involved in a tax audit).

In a large, complex software system with many classes, it is likely that there will be inheritance hierarchies of considerable size. Abstract methods are an ideal way of ensuring that all sub classes have a particularly important functionality and that the method of accessing this functionality is consistent across all of the sub classes. Remember that if the required functionality works the same for all the sub classes then we don't need to use abstract methods, we can simply put the method in the super class. But what happens when the implementation is different for each sub class but we do want to ensure that every sub class has that functionality and that, from the public view of the class, it works the same way.

There are three broad categories of abstract class:

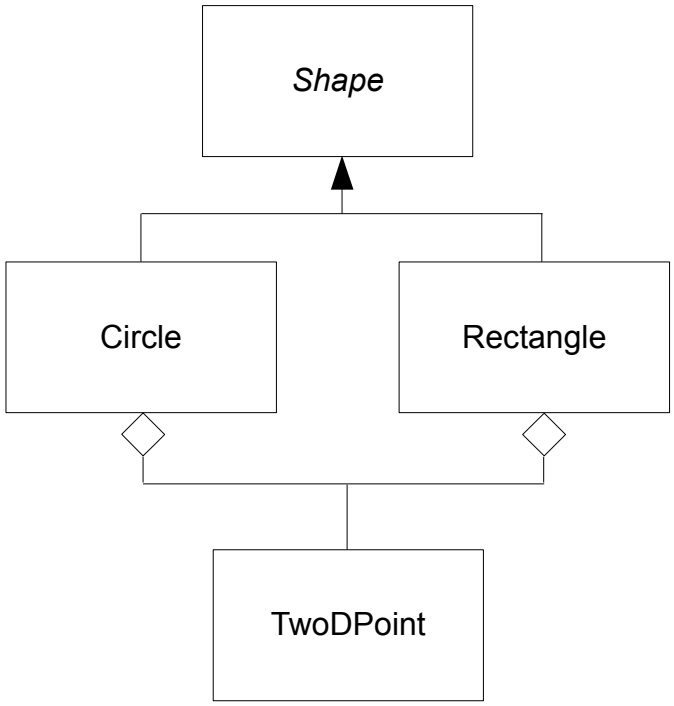
- Abstract classes which do not have any abstract methods (e.g. RentalItem).
- Abstract classes that have the normal stuff (class fields, accessors mutators etc) and at least one abstract method.
- Abstract classes which consist of nothing but abstract methods.

If we revisit the Shape example, we will see that, not only is Shape an abstract class, it will fit the last category. Consider a super class for 2D shape and the sub classes circle and rectangle. Circle and rectangles are both a kind of 2D shape, so relating these classes to the Shape super class via inheritance is valid. However a circle is defined by its centre and its radius and a rectangle by its four corner points. There is no common functionality with regard to the information that describes each shape. Lets suppose that we do want to obtain information related to area from all sub classes of Shape:

- We want to the area of the shape.
- We want to compare the areas of two shapes to see if one is less than, greater than or equal to the other.

Hence we use abstract methods in the Shape class to capture that functionality. This ensures that each sub class has to implement the methods and that they have to have matching signatures. We will also make use of the TwoDPoint class for representing centre and corner points. For simplicity's sake we will only deal with Circle and Rectangles.

From a UML perspective we have:



Notice that the name of the super class (i.e. Shape) is in italic. Under UML this is how an abstract class is specified. Notice also that the aggregation relationship with TwoDPoint is linked to the sub classes and not the super class because the TwoDPoint class fields will exist in the sub class and not the super class.

As stated previously, the Shape abstract class will simply consist of a specification for abstract methods which will accomplish the required functionality:

- A method called calcArea for calculating the area of the shape.
- A method called areaLessThan for comparing two shapes to see if one has an area which is less than the other.
- A method called areaGreaterThan for comparing two shapes to see if one has an area which is greater than the other.
- A method called areaEquals for comparing the areas of two shapes and determining if they are equal. Note this is not the same as the standard equals method which would compare the state information describing the shape and not simply its area.

Hence our abstract Shape class becomes:

```
ABSTRACT CLASS Shape:
Class Fields: None

default constructor  IMPORT None
super

ABSTRACT Accessor calcArea  IMPORT None  EXPORT area

ABSTRACT areaLessThan    IMPORT inShape (Shape)
                         EXPORT isLess

ABSTRACT areaGreaterThan IMPORT inShape (Shape)
                         EXPORT isGreater

ABSTRACT areaEquals      IMPORT inShape (Shape)
                         EXPORT isSame
```

Notice that a default constructor is included but that it serves only to call the super class default constructor. if this had been left out the compiler would have added it to ensure that all the objects, from the base class object to the leaf class object, were created and packaged together.

The sub class will look the same as the others we have seen, with the usual collection of constructors, accessors and mutators. In addition they will have full implementations of the four abstract methods defined in the super class.

```
CLASS Circle INHERITS FROM Shape
Class Fields, radius, centre (TwoDPoint)
Class Constant PI = 3.14159

Default constructor IMPORT None
radius = 1.0
construct centre using default constructor

Alternate Constructor IMPORT inRadius, inCtre (TwoDPoint)

IF inRadius > 0.0 THEN
   radius = inRadius
   construct centre using inCtre
ELSE
   radius = 1.0
   construct centre using default constructor
ENDIF

Copy Constructor IMPORT inCircle (Circle)

radius = inCircle.getRadius
centre = inCircle.getCentre
```

```
Mutator setRadius IMPORT inRadius EXPORT None
IF inRadius > 0.0 THEN
   radius = inRadius
ENDIF

Mutator setCentre IMPORT inCtre (TwoDPoint) EXPORT None
construct centre using inCtre

Mutator setCircle IMPORT inRadius, inCtre (TwoDPoint)
                  EXPORT None
IF inRadius > 0.0 THEN
   radius = inRadius
   construct centre using inCtre
ENDIF

Accessor getRadius IMPORT None EXPORT radius

Accessor getCentre IMPORT None EXPORT centre

Accessor toString IMPORT None  EXPORT stateStr
stateStr = "Center is ", centre.toString,
           "  radius is ", radius

Accessor equals IMPORT inCircle (Circle) EXPORT same
IF centre.equals<--inCircle.getCentre AND
   withinTolerance<-- radius, inCircle.getRadius THEN
   same = true
ELSE
   same = false
ENDIF

PRIVATE withinTolerance IMPORT num1, num2 EXPORT inTol
IF | num1 - num2 | < 0.001 THEN
   inTol = true
ELSE
   inTol = false
ENDIF

Accessor calcArea  IMPORT None  EXPORT area
area = PI x radius x radius

Accessor areaLessThan IMPORT inShape(Shape) EXPORT isLess
IF calcArea < inShape.calcArea THEN
   isLess = true
ELSE
   isLess = false
ENDIF

Accessor areaGreaterThan IMPORT inShape (Shape)
                         EXPORT isGreater
IF calcArea > inShape.calcArea THEN
   isGreater = true
ELSE
   isGreater = false
ENDIF
```

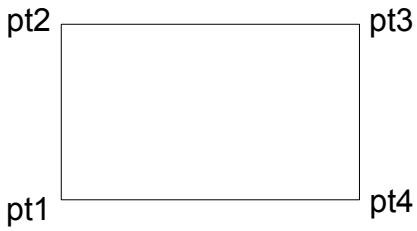
```
Accessor areaEquals IMPORT inShape (Shape) EXPORT isSame
myArea = calcArea
otherArea = inShape.calcArea
IF withinTol<--myarea, otherArea THEN
   isSame = true
ELSE
   isSame = false
ENDIF
```

Before we go onto the Rectangle class lets zoom in on a few things regarding the Circle class. Firstly, remember when we discussed basic data types we talked about the issue of comparing two real numbers for equality. We said that what we do in this situation is measure the difference between the two numbers and then, if that difference is small enough, we consider the two numbers to be equal. In our Circle class we need to compare radius and area which are both real numbers. hence we define the withinTol sub module for this purpose and then call it when required.

The calcArea implementation is pretty straight forward. Its signature matches the abstract definition in Circle and so all we needed to do is to said the area calculation.

The areaLessThan, greater than and equals methods need a bit more examination. Note firstly that, because we defined them as abstract methods in the super class, its IMPORT is defined as a super class object (i.e. Shape) even though we know that, whenever the method is invoked it will be handed a sub class object. The functionality required of the IMPORT objects is a call to the calcArea accessor. We know that ALL the sub classes HAVE to implement the calcArea method (because it was one of the abstract methods defined in the abstract super class) so we know that this will always work.

For completeness here is the Rectangle class. You will note that it has very similar functionality. One thing that needs further explanation is the representation for a rectangle. We will represent the rectangle by the four corned points but, in order to ensure we always have a rectangle we need to ensure the four points are consistently arranged. We will use the pattern below:



IMPORT will always be pt1 followed by width and height, so that the other points are determined by adding width and/or height to the pt1 coordinate (e.g. p2 = pt1 + height).

```
CLASS Rectangle INHERITS FROM Shape
Class Fields, pt1, pt2, pt3, pt4 (all TwoDPoint)

Default constructor IMPORT None
construct pt1 using 0, 0
construct pt2 using 0, 1
construct pt3 using 1, 1
construct pt4 using 1, 0

Alternate Constructor IMPORT inPt1 (TwoDPoint),
                             height, length

IF length and height > 0 THEN
   construct pt1 using inPt1
   construct pt2 using inPt1.getX, inPt1.getY + height
   construct pt3 using inPt1.getX + length,
                       inPt1.getY + height
   construct pt4 using inPt1.getX + length, inPt1.getY
ELSE
   construct pt1 using 0, 0
   construct pt2 using 0, 1
   construct pt3 using 1, 1
   construct pt4 using 1, 0
ENDIF

Copy Constructor IMPORT inRect (Rectangle)

pt1 = inRect.getPt1
pt2 = inRect.getPt2
pt3 = inRect.getPt3
pt4 = inRect.getPt4

Mutator setPoints IIMPORT inPt1 (TwoDPoint),
                             height, length

IF length and height > 0 THEN
   construct pt1 using inPt1
   construct pt2 using inPt1.getX, inPt1.getY + height
   construct pt3 using inPt1.getX + length,
                       inPt1.getY + height
   construct pt4 using inPt1.getX + length, inPt1.getY
ENDIF

Accessor getPt1 IMPORT None EXPORT pt1
Accessor getPt2 IMPORT None EXPORT pt2
Accessor getPt3 IMPORT None EXPORT pt3
Accessor getPt4 IMPORT None EXPORT pt4

Accessor getLength IMPORT None EXPORT length

length = pt4.getX - pt1.getX

Accessor getHeight IMPORT None EXPORT height

length = pt2.getY - pt1.getY
```

```
Accessor toString IMPORT None  EXPORT stateStr
stateStr = "Bottom Left hand Corner is ", pt1.toString,
           " length is ", getLength,
           " width is ", getWidth

Accessor equals IMPORT inRect (TwoDPoint) EXPORT same
IF pt1.equals<--inRect.getPt1 AND
   pt2.equals<--inRect.getPt2 AND
   pt3.equals<--inPt.getPt2 AND
   pt4.equals<--inPt.getPt4 THEN
   same = true
ELSE
   same = false
ENDIF

PRIVATE withinTolerance IMPORT num1, num2 EXPORT inTol
IF | num1 - num2 | < 0.001 THEN
   inTol = true
ELSE
   inTol = false
ENDIF

Accessor calcArea  IMPORT None  EXPORT area

area = getLength x getWidth

Accessor areaLessThan IMPORT inShape(Shape) EXPORT isLess
IF calcArea < inShape.calcArea THEN
   isLess = true
ELSE
   isLess = false
ENDIF

Accessor areaGreaterThan IMPORT inShape (Shape)
                         EXPORT isGreater
IF calcArea > inShape.calcArea THEN
   isGreater = true
ELSE
   isGreater = false
ENDIF

Accessor areaEquals IMPORT inShape (Shape) EXPORT isSame
myArea = calcArea
otherArea = inShape.calcArea
IF withinTol<--myarea, otherArea THEN
   isSame = true
ELSE
   isSame = false
ENDIF
```

From an algorithmic perspective, we are ignoring the fact that the coordinates for our rectangle could occur in any quadrant and assuming that they are always in the positive quadrant (where x and y are both >= 0). I am doing that for the sake of simplicity. For practice, you might consider what modifications

to the algorithms for the methods in Rectangle are required to cater for this.

Secondly notice, that we have a mutator and an alternate constructor which does not simply IMPORT values which are a precise reflection of the class fields. In this case we have to ensure that the points are ordered in a precise way and that they do form a rectangle and the simplest way to do that is to have the IMPORT be one point, which locates the rectangle in 2D space, and the width and height, which can be used to generate the values for the other four corner points. Note also that we have accessors for the EXPORT of length and height, in addition to accessors for each corner point. In other words this example class should be making you realise that, while there is a pattern to the collection of constructors, accessors and mutators found in a class, it isn't so well defined that they all look exactly the same. The differences arise as the level of complexity increases.

Notice also, that the algorithms for the abstract methods are identical to those is the Circle class with the exception of the calcArea implementation. This might make you think that we could have fully implemented them in the super class and not specified them as abstract methods. However the other three implementations all make use of the calcArea method, which is different. Also the calcArea method makes use of sub class state information which is not known in the super class. Hence the implementations must stay down in each sub class.

Finally lets look at some ramifications of the abstract method declarations. Consider the Main algorithm below:

```
MAIN
   INPUT shapeType
   IF shapeType = "c" THEN
      shape = createCircle
   ELSE
      shape = createRectangle
   ENDIF
   OUTPUT "Shape area is ", shape.calcArea
```

The Shape variable in Main would be declared as an object of the abstract class Shape but would always be constructed as an object of either Circle or Rectangle. The createCircle and createRectangle sub modules would input the required information from the user, call the appropriate constructor and finally EXPORT the newly constructed object back to MAIN. Notice that in MAIN we have assumed sub class functionality (i.e. the calcArea method). Normally we are not allowed to do this but because the sub class functionality we are assuming is specified as an abstract method in the super class, we can get away with it. The reason is because it is specified as an abstract method, we know that the sub class MUST fully implement it and so it will be available in all the sub classes.

The Protected Declaration.

In the Circle and Rectangle example, both sub classes have a method called withinTolerance specified. The algorithm is identical in both cases. This would lead us to move this algorithm up to the super class as a means of removing the repetition of algorithm. However if we do this, we then have to declare the method as being public. This is not good because, while we do want both sub classes to know about it and be able to invoke it, we don't want the rest of the software system to know about it because it represents low level functionality and should not really be part of the public class interface. OO provides us with a compromise for this type of situation. Instead of declaring the method public we can declare the method protected. A protected method is visible to all sub classes (direct or indirect) but invisible to any other part of the software. So now we have three categories of visibility of methods and data:

1. Private: visible only within the class.
2. Public: visible everywhere.
3. Protected: Visible within the class and from any sub class but not visible anywhere else.

Note also that the need for this only arises after the first draft of the classes has been produced. It is very important that, when developing classes, the Software Engineer is willing to go back and revise the contents of each class and, in some cases, what the classes themselves should be. For completeness the revised Shape, Circle and Rectangle classes are shown below. The modifications are in bold:

```
ABSTRACT CLASS Shape:
Class Fields: None

default constructor  IMPORT None
super

ABSTRACT Accessor calcArea  IMPORT None  EXPORT area

ABSTRACT areaLessThan    IMPORT inShape (Shape)
                         EXPORT isLess

ABSTRACT areaGreaterThan IMPORT inShape (Shape)
                         EXPORT isGreater

ABSTRACT areaEquals      IMPORT inShape (Shape)
                         EXPORT isSame

PROTECTED withinTolerance IMPORT num1, num2 EXPORT inTol
IF | num1 - num2 | < 0.001 THEN
   inTol = true
ELSE
   inTol = false
ENDIF
```

```
CLASS Circle INHERITS FROM Shape
Class Fields, radius, centre (TwoDPoint)
Class Constant PI = 3.14159

Default constructor IMPORT None
radius = 1.0
construct centre using default constructor

Alternate Constructor IMPORT inRadius, inCtre (TwoDPoint)

IF inRadius > 0.0 THEN
   radius = inRadius
   construct centre using inCtre
ELSE
   radius = 1.0
   construct centre using default constructor
ENDIF

Copy Constructor IMPORT inCircle (Circle)

radius = inCircle.getRadius
centre = inCircle.getCentre

Mutator setRadius IMPORT inRadius EXPORT None
IF inRadius > 0.0 THEN
   radius = inRadius
ENDIF

Mutator setCentre IMPORT inCtre (TwoDPoint) EXPORT None
construct centre using inCtre

Mutator setCircle IMPORT inRadius, inCtre (TwoDPoint)
                  EXPORT None
IF inRadius > 0.0 THEN
   radius = inRadius
   construct centre using inCtre
ENDIF

Accessor getRadius IMPORT None EXPORT radius

Accessor getCentre IMPORT None EXPORT centre

Accessor toString IMPORT None  EXPORT stateStr
stateStr = "Center is ", centre.toString,
           "  radius is ", radius

Accessor equals IMPORT inCircle (Circle) EXPORT same
IF centre.equals<--inCircle.getCentre AND
   withinTolerance<-- radius, inCircle.getRadius THEN
   same = true
ELSE
   same = false
ENDIF
```

```
Accessor calcArea  IMPORT None  EXPORT area

area = PI x radius x radius

Accessor areaLessThan IMPORT inShape(Shape) EXPORT isLess
IF calcArea < inShape.calcArea THEN
   isLess = true
ELSE
   isLess = false
ENDIF

Accessor areaGreaterThan IMPORT inShape (Shape)
                         EXPORT isGreater
IF calcArea > inShape.calcArea THEN
   isGreater = true
ELSE
   isGreater = false
ENDIF

Accessor areaEquals IMPORT inShape (Shape) EXPORT isSame
myArea = calcArea
otherArea = inShape.calcArea
IF super.withinTol<--myarea, otherArea THEN
   isSame = true
ELSE
   isSame = false
ENDIF



CLASS Rectangle INHERITS FROM Shape
Class Fields, pt1, pt2, pt3, pt4 (all TwoDPoint)

Default constructor IMPORT None
construct pt1 using 0, 0
construct pt2 using 0, 1
construct pt3 using 1, 1
construct pt4 using 1, 0

Alternate Constructor IMPORT inPt1 (TwoDPoint),
                              height, length

IF length and height > 0 THEN
   construct pt1 using inPt1
   construct pt2 using inPt1.getX, inPt1.getY + height
   construct pt3 using inPt1.getX + length,
                       inPt1.getY + height
   construct pt4 using inPt1.getX + length, inPt1.getY
ELSE
   construct pt1 using 0, 0
   construct pt2 using 0, 1
   construct pt3 using 1, 1
   construct pt4 using 1, 0
ENDIF
```

```
Copy Constructor IMPORT inRect (Rectangle)

pt1 = inRect.getPt1
pt2 = inRect.getPt2
pt3 = inRect.getPt3
pt4 = inRect.getPt4

Mutator setPoints IIMPORT inPt1 (TwoDPoint),
                            height, length

IF length and height > 0 THEN
   construct pt1 using inPt1
   construct pt2 using inPt1.getX, inPt1.getY + height
   construct pt3 using inPt1.getX + length,
                       inPt1.getY + height
   construct pt4 using inPt1.getX + length, inPt1.getY
ENDIF

Accessor getPt1 IMPORT None EXPORT pt1
Accessor getPt2 IMPORT None EXPORT pt2
Accessor getPt3 IMPORT None EXPORT pt3
Accessor getPt4 IMPORT None EXPORT pt4

Accessor getLength IMPORT None EXPORT length

length = pt4.getX - pt1.getX

Accessor getHeight IMPORT None EXPORT height

length = pt2.getY - pt1.getY

Accessor toString IMPORT None  EXPORT stateStr
stateStr = "Bottom Left hand Corner is ", pt1.toString,
           " length is ", getLength,
           " width is ", getWidth

Accessor equals IMPORT inRect (TwoDPoint) EXPORT same
IF pt1.equals<--inRect.getPt1 AND
   pt2.equals<--inRect.getPt2 AND
   pt3.equals<--inPt.getPt2 AND
   pt4.equals<--inPt.getPt4 THEN
   same = true
ELSE
   same = false
ENDIF

Accessor calcArea  IMPORT None  EXPORT area

area = getLength x getWidth
```

```
Accessor areaLessThan IMPORT inShape(Shape) EXPORT isLess
IF calcArea < inShape.calcArea THEN
   isLess = true
ELSE
   isLess = false
ENDIF

Accessor areaGreaterThan IMPORT inShape (Shape)
                         EXPORT isGreater
IF calcArea > inShape.calcArea THEN
   isGreater = true
ELSE
   isGreater = false
ENDIF

Accessor areaEquals IMPORT inShape (Shape) EXPORT isSame
myArea = calcArea
otherArea = inShape.calcArea
IF super.withinTol<--myarea, otherArea THEN
   isSame = true
ELSE
   isSame = false
ENDIF
```

There is more to abstract classes than what we have discussed but the remainder is probably best left to a second year Object Oriented Software Engineering subject.

Conclusion

In this chapter we have covered a lot of ground, in terms of how inheritance works and how it is used. In this book we have covered all of the major OO concepts except one. OO gives us the ability to design and write code which can be used on different types of objects. This means we don't need a specific version for each object type. This is called generic code and is covered in the next chapter.

Chapter 9

"In my experience, if you cannot say what you mean, you can never mean what you say. The details are everything."
Durano, Season Four, Babylon 5.

Generic Code.

Introduction

One of the most powerful OO tools is the capability to design and implement code that can work on different objects. This code is called generic code because it can be generically applied to a set of objects generated from different classes. The key to this process lies in the fact that an object variable which is declared to be an object of a super class can be instantiated as an object of any of the sub classes of the super class. For example, the Java programming language has a class called Vector. The Vector class is a collection class in that it administers a collection of objects. It doesn't need any of the object's functionality because it is just looking after a list of them. In the implementation of the Vector class, the object variables are declared to be objects of the base class. Given that every other class, directly or indirectly inherits from the base class, this means you can hand a Vector object any other object and it will be happy to look after it for you!

Generic Code and Abstract Methods

If you recall from the previous chapter, the one problem that arises, is that while it is possible to instantiate a super class object variable with a sub class object, the sub class functionality is not accessible. In the case of the Vector class that isn't an issue because the Vector class just keeps the objects in a list. It doesn't need to access any of the functionality so the issue doesn't arise. What if it did? How are we going to get around this problem?

The answer lies in the use of abstract methods. Remember that generic code will be doing generic things with the objects concerned. Given that is the case, we could simply ensure that all the required functionality was in the super class. That way there is no problem because the sub class functionality is never needed.

What happens though if the algorithms required to achieve the required generic functionality was different for each sub class? We have to place the methods required in the sub classes because each one will have a different algorithm. Think about the shape example from the previous chapter, the four

methods had different implementations in each of the sub classes. The answer is to define abstract methods in the super class that cover the required sub class functionality. That way the functionality will still be accessible.

Confused yet? Okay, time for some examples before suicidal tendencies set in! Actually we have already seen an example of generic code in the previous chapter. Remember the Abstract Shape class:

```
ABSTRACT CLASS Shape:
Class Fields: None

default constructor  IMPORT None
super

ABSTRACT Accessor calcArea  IMPORT None  EXPORT area

ABSTRACT areaLessThan    IMPORT inShape (Shape)
                         EXPORT isLess

ABSTRACT areaGreaterThan IMPORT inShape (Shape)
                         EXPORT isGreater

ABSTRACT areaEquals      IMPORT inShape (Shape)
                         EXPORT isSame
```

The inclusion of the four abstract methods mean that the main algorithm below doesn't need to know which sub class object is involved because all of the functionality that it needs from the Shape object is encompassed in the four abstract methods. The main algorithm below (taken from the previous chapter) is an example of generic code. It will function the same regardless of what sub class the shape variable is constructed from.

```
MAIN
   INPUT shapeType
   IF shapeType = "c" THEN
      shape = createCircle
   ELSE
      shape = createRectangle
   ENDIF
   OUTPUT "Shape area is ", shape.calcArea
```

The use of generic code may not seem that powerful until you start thinking about the sorts of tasks that need to be done in the real world. Imagine a human resources software system. At different times, employees would need to be treated generically and at others, specifically in terms of the category of employment. Imagine producing a list of employees sorted by the amount of leave owed to them. The code that did this would need to treat the employees generically. On the other hand, when dealing with employees individually, the software would have to recognise the specifics of the position.

Therefore when designing generic code, the idea is to roughly determine the steps in the generic algorithm. After doing so, the next step is to determine the functionality required of each of the sub class objects involved. If the functionality required is the same for all sub classes then implement it in the super class. if the implementation needs to be different then define a set of abstract methods to cover the required functionality and them provide the implementation in the sub classes.

Object Interrogation and Object Conversion

What happens though when we have been dealing with an object generically and at some point we need to make use of its sub class functionality? The problem is we need a mechanism for firstly, identifying which sub class the object has been created from (remember if the code is generic we won't have that information) and secondly, we need a mechanism to instruct the compiler to convert the object to the required sub class object so that from that point onwards the algorithm can access the sub class functionality.

The mechanism for achieving these two steps differs from language to language. Java is provides one of the most straight forward mechanisms for this so I will use Java as an implementation example. In Java we can interrogate an object using the *instanceof* operator:

```
if ( shape instanceof Circle )
```

If shape is a Circle object the above Boolean expression will evaluate to true, otherwise it will evaluate to false. For the purposes of our Pseudo Code, we will use a similar approach:

```
IF shape ISA Circle THEN
```

Similarly each programming language has a different mechanism for object conversion. Java uses type casting. In other words Java simply treats object conversion the same as primitive data type conversion (at least from a syntax perspective):

```
circle = (Circle)shape;
```

Typecasting is specific to languages whose syntax is derived from C. I look forward to the happy day when the stain of C is forever removed from human civilisation and so do not wish to use type casting in the Pseudo Code. The following, while a bit clumsy, will suffice:

```
circle = Convert shape to a Circle object;
```

The circle variable must have been declared as a Circle object not a Shape object. After conversion, the Circle methods can be access via the circle variable. Confused yet? Not surprising really. This is a lot to take in and only really makes any sense at all if you are completely happy with the concepts covered in the previous chapter. As always, an example might help.

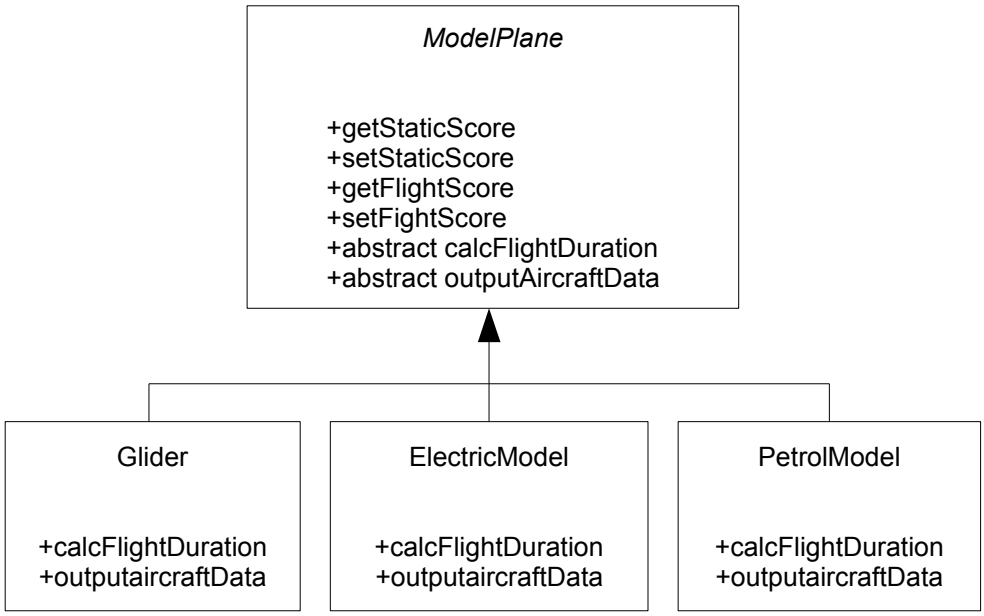
Suppose we were writing a software application to be used by a model plane club when running competitions. To start with we will ignore the object interrogation/conversion issues and develop the classes required as well as the generic algorithm. There are three types of model planes to consider:

- Unpowered gliders.
- Electric powered models.
- Petrol powered models.

To successfully run the competitions and judge the models, the following information needs to be stored regardless of the model type:

- Maximum flight duration (in minutes).
- The static score. A point value awarded by the competition judges on the appearance of the model.
- The flight score. A point value awarded by the competition judges based on the model's competition flight.

Also a method which outputs the details of the model. The static and dynamic scores are user input and so these methods can be placed in the super class because they are the same for all model types. The calculation of flight duration and the model details to be output are different for each type of model so they must be abstract methods in the super class and full implementations must be provided in the sub classes. Hence we end up with the following classes:



I have purposely left out any class details not concerned with the generic code issues so we can focus on what we need to without distraction. What we see in the super class, is full implementations of the accessors and mutators required for maintaining the static and flying scores. We also see the two abstract methods which ensure that these methods must be implemented in all the sub classes. This means the top four methods are available directly

from the super class and the bottom two must be implemented in all of the sub classes and so can be assumed to be available no matter what sub class is involved.

We can now develop an algorithm which deals with the scoring and judging of models generically. No allowance needs to be made on what type of model aircraft is being dealt with.

Our algorithm needs to input the model details, one at a time, all the while keeping track of the highest scoring model. Finally the winner needs to be output. A first run out our main algorithm results in:

```
MAIN
   DO
     INPUT nextModel
     IF nextModel scores better than winner THEN
         winner = nextModel
      ENDIF
      INPUT goAgain
   WHILE goAgain = yes
   OUTPUT winner
```

We need to do a lot more refinement but we have the basic idea. The first point to recognise is that we cannot simply input an object from the user so we need a sub module which will input the information required and then construct and export the desired object.. In main nextModel and winner would be declared as objects of the Abstract class ModelPlane but would always in reality be initialised with a sub class object.

Also we need to determine if one object is greater than another. This will be done by adding functionality to our classes. The abstract class requires another abstract method called calcTotalScore. It has to be abstract because the calculation is different for each model type. Then this method must be fully implemented in each sub class. Assuming that we make these changes to the classes, our main algorithm then looks like:

```
MAIN
   winner = constructLoser
   DO
     nextModel = inputNextModel
     IF nextModel.calcTotalScore > winner.calcTotalScore
                                                    THEN
         winner = nextModel
      ENDIF
      INPUT goAgain
   WHILE goAgain = 'Y' OR 'y'
   OUTPUT winner.outputModelData
```

Note the constructLoser submodule constructs a Glider object with a flight duration of zero minutes and static and flying scores of zero. That way any of the models input will score higher than it and so winner will always be updated to the first model. After that it will only be updated when the next model has a

higher total score.

Have you spotted the problem with our code yet? The problem lies in the step which updates the winner object. We need to call a copy constructor. The problem is the copy constructor has to be from one of the sub classes. At this point in our algorithm we don't know which sub class nextModel is an object of. Hence we don't know which copy constructor to call (i.e. Glider, PetrolPowered or ElectricPowered). As always we side line the task to a sub module. The sub module's job will be to use object interrogation to identify which sub class nextModel is an object of and to initialise winner with the correct sub class object. Firstly lets add the sub module call to our main algorithm:

```
MAIN
   winner = constructLoser
   DO
     nextModel = inputNextModel
     IF nextModel.calcTotalScore > winner.calcTotalScore
                                                    THEN
         winner = updateWinner<--nextModel
      ENDIF
      INPUT goAgain
   WHILE goAgain = 'Y' OR 'y'
   OUTPUT winner.outputModelData
```

Now we can deal with the object interrogation and conversion in its own sub module. Note that using a sub module preserves the generic nature of our main algorithm.

This leads us to develop the following algorithm for our new sub module:

```
Sub Module updateWinner
IMPORT nextModel (ModelPlane)
EXPORT winner (ModelPlane)

IF nextModel ISA Glider THEN
   glider = convert nextModel to a Glider
   construct winner using glider
ELSE IF nextModel ISA ElectricPowered THEN
   electric = convert nextModel to ElectricPowered
   construct winner using electric
ELSE
   ASSERTION if nextModel isn't a glider or an electric
             powered model then it must be a petrol
             powered model.
   petrol = convert nextModel to PetrolPowered
   construct winner using petrol
ENDIF
```

Note that because both object interrogation and object conversion are handled differently from one programming language to another, our use of pseudo code is starting to reveal cracks in its armour. Just keep in mind that nothing in this world is perfect (certainly not Pseudo Code or OO) but the lack

of perfection in a tool does not mean that you should discard the tool and seek perfection.

Conclusion

In this chapter we have introduced the concepts of generic code and the related issues of object interrogation and object conversion. Suffice to say this book has just scratched the surface. However, as first year students, this is as far as we need to go. If you have understood the concepts discussed in this chapter then you understand object oriented algorithm design at a level which is more than sufficient for the first year of a computing degree. Well done!

Chapter 10

"It is known that there are an infinite number of worlds, simply because there is an infinite amount of space for them to be in. However, not every one of them is inhabited. Therefore, there must be a finite number of inhabited worlds. Any finite number divided by infinity is as near to nothing as makes no odds, so the average population of all the planets in the Universe can be said to be zero. From this it follows that the population of the whole Universe is also zero, and that any people you may meet from time to time are merely the products of a deranged imagination." Douglas Adams, The Restaurant At the End of the Universe, 1980.

Arrays.

Introduction

In our algorithms so far we have only dealt with small amounts of data. Obviously in the real world algorithms have to deal with large collections of data. That leads to the fact that Computer Scientists have gone to great lengths to provide us with many, many different ways in which to arrange information. Typically these are taught in a subject by themselves (e.g. Data Structures 102) and are not really part of what i am trying to cover in this book. However the simplest and most straight forward data structure is called an array and that is something i want to talk about.

What is an Array?

An array is simply a linear list of data. Each position in the list is indicated by an integer which represents where it comes in the list. The first item would be in position 1, the second in position 2 and so on. All the items in the list must be the same data type. I cannot have an array which is a mixture of real numbers and integers for example. They would have to be all integers or all real numbers. Arrays of objects blur this completely but, at least for the moment, let us just consider arrays of primitive data types. Each item in an array has two pieces of information attached to it:

- The actual value. This is called an array *element*.
- The position of the element in the array. This is called an array *index*. It is also referred to as an array *subscript*. I will use these two terms interchangeably so you can get used to them both.

The exact syntax used varies from programming language to programming

language but the general idea is to:

- Give the array a name (e.g. myNumbers).
- Specify which array element you are referring to by proving an index inside brackets (e.g. myNumbers(2), myNumbers[4].

Most modern programming languages allow the actual size of an array (i.e. how many data items I can store in it) to be determined when the program executes. However the sizing of an array can only occurrence and from that point on its size is fixed. Older programming languages require you to specify the size of the array when declaring the array variable.

That means that to play with an array we need to:

- Specify its size, either statically in a variable declaration or as a step in an algorithm.
- Initialise all its elements.
- Cycle through the array doing whatever we need to do with it.

The exact manner in which arrays are treated varies a lot from one programming language to the next. We will discuss the attributes of arrays from a completely generic perspective but that means we need to define how we are going to refer to things in the pseudo code for this chapter. You don't need to mimic the exact representations we will use, you just need to have something equivalent and be very consistent.

In our arrays:

- We will allow them to be sized (also referred to as dimensioned) once only.
- We will use () to refer to each array element via its array name and index.
- We will use a period followed by the word *length* to refer to the size of the array.

Confused yet? I am not surprised! Let's see if we can clarify things with an example. Suppose we were writing an algorithm which:

- Inputs 10 integers.
- Calculates their average and then
- Outputs the number of values input that are below the calculated average.

Notice that, as algorithms go, this one is about as useful as a fridge in Antarctica, but the point is we simple what a simple algorithm to illustrate how arrays are used.

For simplicity we will put our entire algorithm in main so we can focus on arrays but we will revisit that shortly and use sub modules.

```
MAIN
   size numbers to have 10 elements.
   FOR next = 1 TO numbers.length CHANGEBY 1
      input numbers(next)
   ENDFOR

   sum = 0.0
   FOR next = 1 TO numbers.length CHANGEBY 1
      sum = sum + numbers(next)
   ENDFOR
   average = sum/10.0

   belowAve = 0
   FOR next = 1 TO numbers.length CHANGEBY 1
      IF numbers(next) < average THEN
         belowAve = belowAve + 1
      ENDIF
   ENDFOR
   OUTPUT "Number of input values below average are:",
          belowAve
```

Notice the first thing we have to do is size (or dimension) the array, in this case to have 10 elements. The next thing we do is to use a FOR loop to index from the first element to the last. Each time through the loop we are initialising one array element. Notice that the only time we refer to the array without specifying an array subscript is when we are sizing it. The only other time is when it is specified as IMPORT or EXPORT. The length of the array is used in each FOR loop to ensure that the algorithm never attempts to refer to an array element that does not exist. This is very important because many programming languages will not warn you if you run off the end of an array! Bringing sub modules into the mix doesn't really cause any problems in pseudo code but there is one issue that we need to keep in mind when using real programming languages. We will discuss that a bit further on so for the moment don't worry.

Arrays Sub Module IMPORT and EXPORT

The only extra thing is the specification of arrays as IMPORT and EXPORT. Lets redo the previous algorithm using sub modules:

```
MAIN
   size numbers to have 10 elements.
   inputNumbers<--numbers-->numbers

   average = calcAverage<--numbers

   belowAve = calcBelowAverage<--numbers, average
   OUTPUT "Number of input values below average are:",
          belowAve
```

```
Sub Module inputNumbers
IMPORT numbers
EXPORT numbers
   FOR next = 1 TO numbers.length CHANGEBY 1
      input numbers(next)
   ENDFOR

Sub Module calcAverage
IMPORT numbers
EXPORT average
   sum = 0.0
   FOR next = 1 TO numbers.length CHANGEBY 1
      sum = sum + numbers(next)
   ENDFOR
   average = sum/10.0

Sub Module calcBelowAverage
IMPORT numbers, average
EXPORT belowAve
   belowAve = 0
   FOR next = 1 TO numbers.length CHANGEBY 1
      IF numbers(next) < average THEN
         belowAve = belowAve + 1
      ENDIF
   ENDFOR
```

Given your experience with sub modules, the above example should not be too difficult to grasp with one exception. Notice that the inputNumbers sub module lists the numbers array as both IMPORT and EXPORT. The array is being filled with numbers input from the user. This means it should be categorised as EXPORT but why is it also classified as IMPORT? The reason is because it was sized in the main sub module so even though its elements were uninitialised in the main sub module, it size was initialised. In other words we created the array in MAIN by sizing it, handed the empty array to the inputNumbers sub module as IMPORT, the sub module filled the array with input from the user and then EXPORTed it back to MAIN. If we dimensioned the array in the inputNumbers sub module then it would only be categorised as EXPORT: When implementing arrays in programming language this can be a very significant issue which impacts how the array is IMPORTed or EXPORTed to/from the sub module.

```
MAIN
   inputNumbers-->numbers

   average = calcAverage<--numbers

   belowAve = calcBelowAverage<--numbers, average
   OUTPUT "Number of input values below average are:",
           belowAve
```

```
Sub Module inputNumbers
IMPORT None
EXPORT numbers
   size numbers to have 10 elements.
   FOR next = 1 TO numbers.length CHANGEBY 1
      input numbers(next)
   ENDFOR
```

Hence the concept to take away from this is that when initialising an array there are two steps to it:

- Initialise the size of the array by dimensioning it.
- Initialising all of the array elements.

Are all Arrays Indexed from 1?

In the previous example the first element of the array had an index of 1. While this seems the obvious thing to do, many languages don't! Some languages allow the coder to specify the index range. The C programming language started the trouble by setting a rather unfortunate precedent of the first element of an array having an index of zero. They were not mad, they did this for a reason which was to make the calculations of the memory address of an array element slightly simpler and hence faster. That was very important in 1970 because computers were much slower than today. However in a world where graphics cards do real time 3D modelling and rendering the same reasoning no longer applies (in those days they also said that it would never be possible to create a movie that was entirely computer graphics! I wonder how they felt about that when they saw Toy Story). Because C managed to become so prevalent, many newer languages have adopted the same idea which is unfortunate. Indexing an array from zero makes no sense in terms of algorithm understandability. Many mathematical algorithms assume indexing from 1 and this can cause many problems with translating them into languages that index arrays from zero. One way around this is to size your arrays one element more than you need and simply don't use the first element. While this wastes a bit of memory and requires you to pay extra attention when sizing arrays, it does make the rest of your algorithm much simpler.

Array Based Algorithms

Most algorithms treat arrays in one of two ways. Either they use FOR loops to cycle through the array elements or they calculate an index value and use that to refer to a particular array element. To demonstrate both categories we will design an algorithm which inputs some integers, uses a simple sorting algorithm to sort them and then uses a search algorithm to search the array for a particular value. The sorting algorithm uses loops to process the array and the search algorithm repeatedly calculates the centre position of the array. Of course we start in main by ignoring all that detail and specifying sub modules for each task:

```
MAIN
   numbers = inputNumbers
   sortNumbers<--numbers-->numbers
   searchNumbers<--numbers

Sub Module inputPositiveNumber
IMPORT None
EXPORT number
INPUT number
WHILE number <= 0 DO
   OUTPUT "Number must be positive"
   INPUT number
ENDWHILE

Sub Module inputNumbers
IMPORT None
EXPORT numbers

size = inputPositiveNumber
size numbers to have size elements
FOR next = 1 TO numbers.length CHANGEBY 1
   numbers(next) = inputPositiveNumber
ENDFOR
```

The inputNumbers sub module is straight forward. Like the last algorithm we dimension the array in the input sub module so the array is specified as EXPORT only. Note this is totally subjective and it would not be wrong to dimension the array in MAIN. The sorting algorithm we will use is the simplest one in the history of human civilisation. It is called the bubble sort. It works by repeatedly cycling through the array and swapping each array element with its neighbour if its neighbour is larger. Each time the array is cycled through, the next largest array element will have *bubbled* to its sorted position (hence the name). This process is repeated until a pass through the array is made where no array elements were swapped. If no swaps were made then the array must be sorted and the algorithm can stop as its job is done.

```
Sub Module sortNumbers
IMPORT numbers
EXPORT numbers

DO
   swapped = false
   FOR next = 1 TO numbers.length-1 CHANGEBY 1
      IF numbers(next) > numbers(next+1) THEN
         swap<--numbers, next, next + 1
         swapped = true
      ENDIF
   ENDFOR
WHILE swapped = true
```

This in turn has resulted in the need for a swap sub module:

```
Sub Module swap
IMPORT numbers, pos1, pos2
EXPORT numbers

temp = numbers(pos1)
numbers(pos1) = numbers(pos2)
numbers(pos2) = temp
```

Notice the FOR loop which is visiting all the array elements, one at a time. This is in contrast to the search algorithm. The search algorithm is called the Binary Search because it finds the middle array element and compares it to the value be searched for. If the value is equal then the search halts. If the value is smaller then the left side of the array is searched otherwise the right side is searched. Hence in this algorithm we won't see a FOR loop cycling through the array.

```
Sub Module searchNumbers
IMPORT numbers
EXPORT None

searchValue = inputPositiveNumber
found = binarySearch<--numbers, searchValue
IF found THEN
   OUTPUT "Search value is present in array"
ELSE
   OUTPUT "Search value is not present in array"
ENDIF

Sub Module binarySearch
IMPORT numbers, searchValue
EXPORT found
leftPos = 1
RightPos = numbers.length
found = false
WHILE leftPos <= RightPos AND NOT found DO
   midPos = (leftPos + rightPos) / 2
   IF numbers( midPos) = searchValue THEN
      found = true
   ELSE IF numbers( midPos) < searchValue THEN
      leftPos = midPos + 1
   ELSE
      rightPos = midPos - 1
   ENDIF
ENDWHILE
```

Don't get too hung up on the searching algorithm (but if you think about it, its pretty cool) just note that we have a loop and each time through the loop we are calculating different array index values so the binary search algorithm is a good example of the second category. Is it possible to have an algorithm which does both? Of course! The main thing in both cases is to ensure that you never attempt to index a position in an array which does not exist. As stated previously, many programming languages do not trap running off the end of an array as an error. This means your program will be accessing

memory that it should not and the general result is unpredictable behaviour. A classic sign is when a variable's value seems to mysteriously change even though you never modified it. This can happen when your algorithm runs of the end of an array and puts a value in memory that isn't part of the array. If this memory is being used by another variable then that variable's value has been corrupted. When dealing with a software application that uses arrays and behaves strangely, its the first thing to check for. Many of the security exploits that breach the security embedded in certain operating systems often uses array over-runs as a starting point for their mischief.

Multi-Dimensional Arrays

All our discussions of arrays have dealt with what we call one dimensional arrays. In other words a linear list of data items. However, often we are trying to represent information that requires more than one dimension. For example if we had a table of numbers which are arranged in rows, we would use a two dimensional array. The difference is that the array now has two index values, one which specifies which row and the other which specifies which element in the specified row (i.e which column). The idea is exactly the same except our algorithms have to deal with two indexes instead of one. For example, suppose in our first array example we used a two dimensional array instead. The resulting algorithm would be identical except that we would need more loops and array indexes:

```
MAIN
   inputNumbers-->numbers
   average = calcAverage<--numbers
   belowAve = calcBelowAverage<--numbers, average
   OUTPUT "Number of input values below average are:",
           belowAve

Sub Module inputNumbers
IMPORT None
EXPORT numbers
   size numbers to have 10 rows and 10 columns
   FOR row = 1 TO numbers.numRows CHANGEBY 1
      FOR col = 1 TO numbers.numCols CHANGEBY 1
         input numbers(row, col)
      ENDFOR
   ENDFOR

Sub Module calcAverage
IMPORT numbers
EXPORT average
   sum = 0.0
   FOR row = 1 TO numbers.numRows CHANGEBY 1
      FOR col = 1 TO numbers.numCols CHANGEBY 1
         sum = sum + numbers(row, col)
      ENDFOR
   ENDFOR
   average = sum/(numbers,numRows x numbers.numCols)
```

```
Sub Module calcBelowAverage
IMPORT numbers, average
EXPORT belowAve
   belowAve = 0
   FOR row = 1 TO numbers.numRows CHANGEBY 1
      FOR col = 1 TO numbers.numCols CHANGEBY 1
         IF numbers(row, col) < average THEN
            belowAve = belowAve + 1
         ENDIF
      ENDFOR
   ENDFOR
```

I intentionally used the same algorithm as before so you could observe the differences between playing with one and two dimensional arrays. Notice that instead of numbers.length we have numbers.numRows and numbers.numCols because now our array is a collection of numbers arranged in rows and columns. Also, each time we want to cycle through the array we need two nested FOR instead of one. Notice also that the general convention is to specify the row index followed by the column index.
As I have stated previously, the exact mechanisms for sizing and accessing the array elements and the array dimensions varies from one programming language to the next but the principle is the same.

We can actually expand the idea to three dimensions and the result is we would then have a third array subscript. We envisage our three dimensional array as a series of two dimensional arrays stacked against each other. Hence we now have slices, rows and columns:

```
MAIN
   inputNumbers-->numbers
   average = calcAverage<--numbers
   belowAve = calcBelowAverage<--numbers, average
   OUTPUT "Number of input values below average are:",
           belowAve

Sub Module inputNumbers
IMPORT None
EXPORT numbers
   size numbers to have 10 slices, 10 rows and 10 columns
   FOR slice = 1 to numbers.numSlices
      FOR row = 1 TO numbers.numRows CHANGEBY 1
         FOR col = 1 TO numbers.numCols CHANGEBY 1
            input numbers(slice, row, col)
         ENDFOR
      ENDFOR
   ENDFOR
```

```
Sub Module calcAverage
IMPORT numbers
EXPORT average
   sum = 0.0
   FOR slice = 1 to numbers.numSlices
      FOR row = 1 TO numbers.numRows CHANGEBY 1
         FOR col = 1 TO numbers.numCols CHANGEBY 1
            sum = sum + numbers(slice, row, col)
         ENDFOR
      ENDFOR
   ENDFOR
   average = sum/(numbers,numRows x numbers.numCols)

Sub Module calcBelowAverage
IMPORT numbers, average
EXPORT belowAve
   belowAve = 0
   FOR slice = 1 to numbers.numSlices
      FOR row = 1 TO numbers.numRows CHANGEBY 1
         FOR col = 1 TO numbers.numCols CHANGEBY 1
            IF numbers( slice, row, col) < average THEN
               belowAve = belowAve + 1
            ENDIF
         ENDFOR
      ENDFOR
   ENDFOR
```

In theory we can go beyond three dimensional arrays. In reality that tends to lead to complexity and confusion and finally insanity. If you think about it one, two and three dimensions maps nicely into the real world but beyond that things become a tad abstract and hence we start getting confused. I am not saying its not possible just that generally, its not a good idea. I knew someone who used a six dimensional array to solve a problem once but I think he was more excited by the challenge of actually using a six dimensional array than by any thoughts of whether it was an appropriate choice. I also remember it drove him to the borders of insanity to get his software working correctly and when he went back to modify his code a few years later, he ended up throwing it away and starting again. Needless to say his new version did not involve a six dimensional array!

You will also notice that I have cheated a bit here. The above two algorithms fit the first category where we cycle through all the elements. What about the second category where we calculate the array indexes? This second category does exist but usually isn't extendable across different dimensions of arrays. For example the binary search assumes the numbers are ordered in a one dimensional list and so isn't possible in two dimensions. In other words the second category of algorithm occurs in the context of what the array represents.

Arrays of Objects

Notice that in all our array algorithms the only difference between manipulating a variable that is or isn't an element of an array is how we reference it. For example we can add, subtract, multiply or divide two integer values regardless of whether or not they are stored in an array. The only difference is that when it is in an array element we have to use index(es) to specify which element it is. The same is true for objects. Inheritance raises an issue here because we can specify an array to be an array of superclass objects and the instantiate with a mixture of different sub class objects but, just as when we do this for single object variables, the same rules apply in that we would only have access to the super class functionality.

This is best illustrated via some examples. Lets start with a simple example and then we will move onto the inheritance related issues. In our first example we will have a simple container class called Circle which has location (i.e. x and y values) and radius as class fields. I am not going to provide the pseudo code for the class (hopefully that is pretty easy for you to come up with by now). We will simply assume the usual collection of constructors, accessors and mutators. The algorithm will create an array of Circle objects, calculate the average radius of all the circles in the array and then output the number of circles whose radius is below the average. If this sounds a bit like the first example we used at the beginning of this chapter, you are not wrong. I want to use the same kind of algorithm so you can contrast the differences and similarities between arrays of primitives and arrays of objects.

```
MAIN
   inputCircles-->circles
   averageRad = calcAverageRadius<--circles
   outputBelowAverage<--circles, average

Sub Module inputCircles
IMPORT None
EXPORT circles
   size circles to have 20 elements
   FOR next = 1 to circles.length CHANGEBY 1
      circles(next) = inputCircle
   ENDFOR

Sub Module inputCircle
IMPORT none
EXPORT circle

   INPUT x, y
   DO
      INPUT radius
   WHILE radius <= 0.0
   construct circle using x, y and radius
```

```
Sub Module calcAverageRadius
IMPORT circles
EXPORT average

sum = 0.0
   FOR next = 1 to circles.length CHANGEBY 1
      sum = sum + circles(next).getRadius
   ENDFOR
   radius = sum / circles.length

Sub Module  outputBelowAverage
IMPORT circles, average
EXPORT None

   numBelow = 0
   FOR next = 1 to circles.length CHANGEBY 1
      IF circles(next).getRadius < average THEN
         numBelow = numBelow + 1
      ENDIF
   ENDFOR
   OUTPUT "Number of circles with below ave radius is ",
          numBelow
```

The above examples show that combining arrays and objects doesn't really change anything. Arrays work the same as with primitive variables and objects are the same as before.

The only real complexity is when inheritance gets added to the mix. It is still true that there is nothing new involved its just that it all gets a little complex. We need to break things down and deal with the object issues and the array issues separately. An example is the best way to illustrate what I am talking about. Lets assume we have a family of classes related by inheritance. There is an abstract class called Shape which contains an abstract method called calcArea() which calculates the area of a 2D shape. We have two sub classes. One is Rectangle and the other is Circle. Again I won't both showing the pseudo code for these classes because that is covered in chapter eight. We start with an array called shapes which is an array of Shape objects. However we will construct each element as either a Circle or a Rectangle sub class object. That means whenever we require sub class functionality we are going to have to interrogate each array element to find out if it is a Circle or a Square object. Lets start with the main algorithm and the sub modules required to create the array.

```
MAIN
   inputShapes-->shapes
   totalArea = calcTotalArea<--shapes
   numCircles = countCircles<--shapes
   OUTPUT "Total are of all the shapes is ", totalArea
   OUTPUT "Number of Circles is ", numCircles
   OUTPUT "Number of rectangles is ",
                              shapes.length - numCircles
```

calcTotalArea will call sub class implementations of the super class abstract method called calcArea so no object interrogation will be required. countCircles will need to interrogate each array element and find out if it is or isn't a Circle object. Note that,as before, our algorithm is about as useful in the real world as a poke in the eye with a sharp stick however it does cover all the concepts involved in dealing with arrays of objects.

```
Sub Module inputShapes
IMPORT None
EXPORT shapes

   size shapes to have 30 elements
   FOR next = 1 to shapes.length CHANGEBY 1
      option = inputOption
      IF option = 'c' THEN
         shapes(next) = inputCircle
      ELSE
         shapes( next) = inputRectangle
      ENDIF
   ENDFOR

Sub Module inputOption
IMPORT None
EXPORT option

   DO
      OUTPUT "Do you want a circle or a rectangle"
      OUTPUT"Enter c for circle or r for rectangle"
      INPUT option
   WHILE option is not 'c' OR 'r'

Sub Module inputCircle
IMPORT none
EXPORT circle

   INPUT x, y
   radius = inputDimension
   construct circle using x, y and radius

Sub Module inputRectangle
IMPORT none
EXPORT rectangle

   INPUT x, y
   length = inputDimension
   width = inputDimension
   construct rectangle using x, y length and width

Sub Module inputDimension
IMPORT none
EXPORT dimension

DO
   INPUT dimension
WHILE dimension <= 0.0
```

After the inputShapes sub module is finished we have an array of objects where each element is either a Circle or a Rectangle object. The calcTotalArea sub module only needs to access the calcArea method in each object. Because this was specified as an abstract method in the Shape super class calls to the method via sub class objects are allowed because the abstract method specification guarantees it will be there.

```
Sub Module calcTotalArea
IMPORT shapes
EXPORT totalArea

   totalArea = 0.0
   FOR next = 1 TO shapes.length CHANGEBY 1
      totalArea = totalArea + shapes(next).calcArea
   ENDFOR
```

Note the object method call of shapes(next).calcArea. This is combining the concepts of arrays and objects together. To access the desired element of the array we have shapes(next). Once we have that specified we wish to indicate which method is invoked and we do that in the same way as we have always done (with a period followed by the method name). The countCircles sub module will need to interrogate each element to see if it is a Circle object or not so we will see the use of ISA. Again this is no different to what we saw in chapter eight other than the array element specification.

```
Sub Module countCircles
IMPORT shapes
EXPORT numCircles

   numCircles = 0
   FOR next = 1 TO shapes.length CHANGEBY 1
      IF shapes(next) ISA Circle THEN
         numCircles = numCircles + 1
      ENDIF
   ENDFOR
```

The previous example shows us that, although things can get a little messy when we bring inheritance into the mix, arrays of objects is simply combining the concepts of arrays with the concepts related to objects. There is no inconsistency involved. Multi dimensional arrays of objects follows the same idea. We still treat an object the same whether or not it is an element of an array and regardless of whether or not the array is one, two, three or more dimensions.

Conclusion

In this chapter we have introduced the concepts related to arrays. There are a lot of other subject areas to familiarise yourself with (e.g. file access, data structures, standard algorithms for searching and sorting). However they are best left for another time.

Conclusion

"We are dreamers, shapers, singers, and makers."
Elric, from The Geometry of Shadows, Babyon 5.

This book has discussed the basic concepts and design techniques involved in designing simple object oriented software. Everything discussed can be implemented in any OO programming language. It is vital to understand how all this works from a generic perspective because specific programming languages come and go. There are literally thousands of programming languages out there but what we have discussed in this book can be applied to all of them. Students should realise that when they sit in their first job interview and the interview panel asks them if they are experienced in a particular programming language, then the correct response is to say "No, but if I cannot pick it up in a week then sack me". I know some IT people that make up the name of a fictional programming language. If the interviewee responds as above then they will be in the running for the job. If they meekly ask if they could go on a training course (or lie and say they know the non-existent language!) then they are rejected. You need to understand that Software Engineering is now and always has been about rapid and dynamic change. Today's hot programming languages are tomorrow's dead technology. Employers expect their software engineers to pick up new languages as a matter of course.

I have taught algorithm design to first year university students for over 20 years and in that time there have been many changes. However all of the concepts and techniques presented in chapter 1 through 5 have not changed. The concepts discussed in the remaining chapters have undergone change over the years since OO was first introduced. However the major issues have also largely remained the same. During all of this time the programming languages employed have changed drastically, especially in the last 10 years. I sincerely hope that this book has helped you understand what you need in order to cope with both OO algorithm design and changes in programming languages.

I wish you all the best in your studies. In addition to reading this book, by now you should have also read the works of Douglas Adams. If so you will understand where my parting message comes from:

So long and thanks for all the fish.