9. Repair, Reuse, Recycle (Object-oriented Programming)

I have done one braver thing
Than all the worthies did,
And yet a braver thence doth spring
Which is to keep that hid.
John Donne, Songs and Sonnets ‘The Undertaking’

What we have looked at so far has been procedural programming. This is just one way of writing programs or **programming paradigm**. A paradigm is a way we see the world. For example, people with different political views – a Right winger versus a left winger, for example – have different political paradigms. They see events in different ways. A right-winger might see the British Welfare State as being about paying money to scroungers who have not bothered to look after themselves, and as just encouraging more such behaviour. A left-winger on the other hand would see it as a humane thing ensuring basic human rights. A left-winger would see an increase in taxes to improve services as a good thing. A right-winger would see the same thing as a bad thing. The two people concerned have a different view of the world. They have different paradigms.
A programming paradigm is a different way of seeing the task of writing a program. There are a variety of different programming paradigms including procedural programming, object-oriented programming, functional programming and logic programming. The procedural paradigm, can roughly be thought of as “a program is a recipe book”. This idea was the way early programs were written in part because it is tied closely with the definition of what an algorithm is, but there are other ways of thinking of the task of programming – the task of writing instructions. Object oriented-programming is now the most popular approach. A procedural programmer would see the task as determining which procedures need to be written – about the state of the system and how to change it. They are thinking in terms of actions that the program performs: each procedure is a recipe for how you do some action. Given this input, how do I turn it into that output? An object oriented programmer, would instead think about what objects are to be manipulated to carry out the task, what their relationship would be. They still have to think about operations but in the context of what operations could be performed on each kind of object that has been identified. The objects come first.

To more clearly see the difference, imagine you are a chef running a small but select restaurant. Because you are expensive, you only offer a set menu, with a small number of options each evening (but boy is it good). How would you think about the task of preparing that meal and plan for it: it would be a series of dishes, one for each course. One obvious way is to write out a recipe for each dish. The menu acts as a contents page into this list, but also indicates the order that the dishes are needed – it is a sequential set of instructions. First prepare the Brishotta as it is the starter, then get to work on the Fiorentina Pizza while the guests eat the first course. You are thinking procedurally. You might split up the procedures and produce an action list that determines the order that everything should be done in. I must put the potatoes on first, then while they are boiling, I must prepare the starter. You have lost the procedural paradigm as you are no longer thinking in terms of separate recipes for each course, but one big recipe for the meal. However, you are still thinking in terms of actions and the order they are achieved. Perhaps you might have a series of helpers

– they could be each given a course to prepare. You have moved to a parallel paradigm as now several things can happen at once – though still one procedurally or action based.

Now think again. Your team of helpers could be trained with specialist skills. One could be a vegetable expert perhaps. He would know how to chop a carrot with a big sharp cleaver extremely quickly and without pieces of finger ending up mixed in. He can core a cauliflower by a flick of the wrist and a tap on the table... Another is a meat expert with similarly appropriate skills. Someone else is a bread, pastry and dough expert. Another is a pizza expert. Given the prepared ingredients, she can make a pizza pass as a piece of modern art and was once shortlisted for the Turner prize. Now think again about the activities in such a kitchen. Instead of focussing on recipes you can think about the objects that are being manipulated: spinach, dough and pizzas. He can twirl dough like Tom Cruise can mix a cocktail. Customers pay just to watch him. The waiter, passes an order through the hatch – perhaps there are a row of hooks that each order is spiked on so that they form a queue to ensure they are processed in order. Fiorentina Pizza is needed. The order is taken up by the Pizza specialist. She realises she needs some Spinach, an egg, dough etc. She writes out orders on a new set of slips and passes them to the appropriate people – putting them on their hook list, to process in their turn. The spinach is been washed and chopped by the vegetable expert, the dough made by the bread expert, etc. Once the prepared ingredients are passed back from the separate people to the pizza expert, she puts them together into a pizza to die for. Eventually it is ready and she passes it to the waiter who passed the order on. In the meantime, an order comes in for Crispy Duck. The meat expert similarly sets in motion a chain of sub-orders. At various points the various experts need new vegetables or whatever to work on. Whenever a request for chopped cabbage comes in, the first job is to get a new cabbage, for example. We are now thinking in a more object-oriented way. We no longer have a single set of instructions for a meal. Instead the instructions, the operations are distributed – they reside with the objects they apply to. The knowledge of how to make dough by kneading and twirling the dough is with the dough expert. Operations are associated with the objects they apply to. Whenever an object is needed – you just go to the expert of that object.

The second view of how the kitchen works, gives a completely different picture, even though ultimately the same operations are performed on the same things. It is a different paradigm. What is the difference? Well, one part of the difference is between verbs and nouns. In the first version, the chef concentrates primarily on *verbs* – action words – the things that need to be done. He thinks in terms of "I must chop this, then mix that, boil,...roast, slice, wash...". The focus is on the actions and the order that they must be done to obtain the final result. Remember that was the way we talked of an algorithm: writing down a series of actions and the order they must be performed in. People are allocated to those actions and when done they come back to be given a new action to do. First they might chop a cabbage, then marinade a chicken, then wash some spinach...

In the second description of the kitchen, the focus is on the *nouns* – things – like cabbages, chicken and spinach rather than the verbs – chop, marinade and wash. Now when people are allocated, it is not to tasks, but to things or categories of things: "You are the cabbage expert. Learn all the skills needed to do anything I might ask with cabbages". You on the other hand are to be the chicken expert and you are to be the cake expert". Now the people – the work units of the kitchen – jump into action whenever one of the things that they specialise in are needed, rather than just being assigned tasks. The focus of the organisation is on nouns rather than verbs. It is closer to an object-oriented description rather than a procedural description.

A procedural programmer thus thinks of the task in terms of the actions that must be taken – the verbs. An object-oriented programmer thinks in terms of the things that actions will be performed on – the nouns. Their thinking is oriented towards things: they are oriented towards objects rather than actions. This difference is then reflected in the language used to write the program. In a way there are similariites in human languages. When babies learn to speak, their first words could be nouns or verbs. In English the first words are likely to be nouns. One of my daughter's earliest words were for things: "teddy" and "car", for example. In some other countries the first words are likely to be verbs, because verbs are predominant in the language and so when the parents are pointing out things they are likely to be emphasising the verbs rather than the nouns. Similarly some programming languages make it easier to focus on verbs – action-oriented languages (i.e., procedural ones), while others make it easier to focus on the nouns – object-oriented languages. We will look at the features that make a language object-oriented in the subsequent sections.

The instruction booklet for my hi-fi is written within an object focus. Rather than being organised around tasks – with sections based on tasks I might wish to perform. The sections are organised around the objects in the hi-fi: a section on the CD player, a section on the radio, a section on the tape deck and so on. Actions are tied to those objects. The manual writer thought of objects first then actions. It is an object-based description. My clock radio on the other hand is organised by tasks. There is one task for each task I might perform. "set the time", "Set the alarm", "Switch on the radio", "Change radio channels". There is no clear focus on the separate objects, the clock, the alarm and the radio in the way the manual is organised. It is a task-based description.

We also see this distinction between organising around nouns (categories of things) and organising around verbs in the way shops are organised. In a traditional high street the shops specialise in one category of objects: the butchers, the bakers, the fruit and vegetable shop and so on. Supermarkets tend to do the same thing too – there is usually a bread counter, a cheese counter, a meat counter and so on. It does not have to be like that though. Many DIY stores organise themselves differently – around tasks – there is a tiling section (tiles, grout, tile-cutters), a painting section (paint, paint brushes), a wallpapering section (wallpaper, paste, papering tables), a gardening section (plants, spades, compost). A supermarket could organise itself in a similar way, with a dinner section, a breakfast section and so on. Or it could have a baking section, a boiling section. It works for DIY because people buying DIY materials tend to be task-oriented. They are thinking of wallpapering the living room on Sunday. When shopping for food, people tend to be much more object-oriented. They are buying for more than just one meal (more than just one task), so in the butchers they might buy, lamb and chicken. In the bakers, sliced bread for breakfast toast and doughnuts for tea. Also the supermarket organised round meals or actions would have to have particular objects replicated in several sections. Was that toast for breakfast of for tea? Were those potatoes for boiling or for baking? This gives a hint as to why object-oriented thinking is so much more powerful in programming. It helps make programs more reusable. When writing an object-oriented program, you are thinking in terms of writing instructions that will be reused for a whole range of different tasks. A task-oriented program is written for a single task, so is likely to need more work to make it fit a different task.

There are many concepts tied up with object-oriented programming called things like classes, objects, encapsulation, inheritance, message-passing and so on. We will look at each in turn. Many of the basic concepts appear in other paradigms. It is only when they are all combined that we truly get the object-oriented paradigm. Hidden within the above description of the kitchen are many of these concepts. We will gradually separate them and explore what these concepts are individually.

Objects
One of the aims of object-oriented programming is to think about programs more like the real-world – just like this book does! The world is made of objects of various kinds, and since most programs are concerned with modelling something in the real world it makes sense to think of the virtual world a program creates to also be made up of **objects**. Objects could represent cabbages, people, houses, aeroplanes ... anything that can be thought of as a person in its own right. Objects come equipped with **attributes** or **state** – something that makes this one different from the next – the way you tell them apart. For example, cabbages have weights, people have eye colour, ages and heights, aeroplanes have flight numbers and destinations, and so on. Attributes are nouns: weight, destination, etc. If two objects have the same state then they will look identical, just like two identical twins appear to the same person. They may act the same and look the same but they are really two completely different people. For real objects the state is just the way it is – the weight of a cabbage is intrinsic within it. For virtual objects the values have to be stored somewhere. As we discussed previously, in computers, things are stored in variables. Think about a car. One of its properties is its registration number. It is stored on a registration plate – in a known place for all cars. If you want to know the registration number of a car then you look at the registration plate. It is acting like a variable for one of the pieces of state of the car. With objects, it is as though all the different parts of the state were all stored in a separate plate that could be checked when the information was needed. For example, it is as though a person carried an identity card with them, and that identity card became the essence of them as far as the state was concerned. If your ID card says you have blue eyes then your eyes are blue! If the police want to check if you are the person they are looking for they just ask to see your ID card and compare it with the profile of the person they are after.

Objects come equipped with more than just **attributes**, however. They also have **behaviours**: the operations that can be performed on an object or more in line with object-think: actions that objects can perform. Each object comes with its own set of behaviours specific to that kind of object. For example, a cabbage can be *chopped* and *boiled*, dough can be *kneaded* and *twirled* methods. Paint can be *mixed*. Thus whereas attributes are described using nouns, behaviours are described using verbs: doing words. The behaviours of an object are implemented as **methods. Methods** are just functions personalised for a particular object. For each kind of object in a program a programmer writes methods to describe its behaviour and gives instance variables to describe its attributes. The instance variables can be checked to see what the values of the attributes are: is the hair of this person black? The methods can be followed to take the actions of the object. In doing so the attributes might change. For example, if a person whose hair colour is brown (hair colour attribute: brown) *dyes* their hair blonde (following the instructions (the method) on the bottle of dye), then when the method has terminated, that person's hair attribute will now have changed to value blonde.

In the Restaurant at the End of the Universe (Adams, 1980) genetically modified animals are bred to delight in being your meal. They take great personal pride in being perfectly fattened and might suggest their leg as being particularly tasty. We can take this idea a little further. Imagine a genetically engineered future in which vegetables and animals are bred to have such intelligence (if you can call it that) and are able to cook themselves. Each chicken has access to a knife and can slice itself to perfection at your request. It can roast itself to death or alternatively fry itself to perfection. Now, when a request comes in messages go to the cabbages, the ducks, etc and they get to work preparing themselves as requested by the pizza, who on creation promptly bakes itself in a hot oven as requested. This is roughly how an object-oriented programmer sees a program. Each different kind of object is supplied with instructions about all the operations that can be performed on it. If such an operation is required, a request goes to the object itself to do it.

Children's films, television programs and books often personify objects in this way. For example, in the Disney version of *Beauty and the Beast*, most of the objects in the castle such as the cups and candles can move around and talk. They not only have attributes they also have behaviours. They get things done by communicating with each other and the object concerned just gets on and does whatever is required. Both the attributes and the behaviours are intimately toed together. Similarly in *Bob the Builder*, all the diggers and cranes can do things for themselves when requested. To be a truly object-oriented television programme however, the bricks, gates walls and so on should also have minds of their own.

Futurologists and science fiction writers will have us believe that all the objects of the future will be responsible for their own actions in this way. In *The Hitchhikers Guide to the Galaxy*, the lifts have control of their behaviours (and personalities to go with them). In the TV programme *Red Dwarf* the vending machines do. In the film *Dark Star* even the missiles have control of their own behaviour – they have the method: *explode* which is fine if they only do so when told to but leads to one depressed missile becoming suicidal. In futuristic houses of the moment, there are fridges that order milk when it runs out – so changing the attribute of amount of milk in the fridge; the garage doors *open* when the car pulls into the drive, and so on. Why not take it further and have the mugs take control of making the coffee.

Even inanimate objects in the real world often have behaviour – a sponge soaks up water on its own, even though it is inanimate, so you do not need to move to a fantasy world to get the idea of objects having both attributes and behaviours. Most commonly inanimate objects only do things when they get some kind of signal or message telling them to do so. For example, my hi-fi switches itself on when I use the remote control to send it a message to do so. The light bulb switches itself on when I throw a switch at the other side of the room to tell it to do so. When I pull the chain, or press the handle on the toilet, the toilet flushes. That is how objects communicate: by **message passing**.

When I am driving along the road and the brake lights of the car in front (an object) go on, it is sending my car (another object) a message to take an action: apply your brakes. The message is the light going on, but because it is a specific light, it sends a message about a specific action I should perform. In taking that action my car will similarly send a message to the car behind it.

The lights on cars are not strictly directed at a specific single other object – and if no one is behind then the message may not go anywhere. The objects in programs send messages to specific other objects. Thus, when writing instructions, two things at least are needed to communicate to an object that it should perform some behaviour. You must indicate the object that must take the action and the behaviour out of the many associated with it that it must take. Many dogs can perform tricks. They are "objects" (appologies to dog lovers) each being able to exhibit a range of doggy behaviour. When we want them to do tricks we give instructions.

"Fido Sit",

"Bonzo Beg",

or even

"Caesar Kill",

all consist of the name of the object followed by the behaviour to be performed (here separated by a space). Commands to objects in a programming language are similar though the symbol used as the separator might be different (for example using a dot instead of a space).

The attributes of objects can be other objects. You could consider a person as having an attribute of blue eyes and another of short sight. Alternatively, you can think of a person of having an attribute that is another object: the eye. Eyes then have attributes of colour and short-sightedness. Because people have eyes and eyes have colour, then by default, people have an eye colour. This can be a good way of organising a description of an object. If you have defined what you mean by eye objects for people, then you can reuse that description for Kangaroo eyes should you later be describing them instead because a kangaroo object also has eye objects as attributes.

Classes

As we discussed earlier, we naturally group and classify objects together into things of the same kind. For example, in the above, I have spoken about cabbages. There are many cabbages in the world, but the **class** of cabbage all share the same properties and the same operations. That is why when I said "cabbage" you pictured the right thing – not any specific cabbage, but all cabbages. A class is thus just a category that groups a variety of objects together. In this sense it is just a type as we discussed previously, but one whose values come equipped with operations too. They are a complex form of **user-defined type** as new categories can be created if needed for a new set of instructions.

As it stands the bricks in Bob the Builder are more like primitive types than classes. A value of that type – a brick for example – is acted upon by other objects but cannot do anything for itself – it has state (its size and colour, for example) but no operations – it is passive. The cement mixer in Bob the Builder on the other hand is not a passive thing that operations happen to but one that comes with the skills to accomplish various tasks (and attitude!) Given the ingredients it can apply operations to itself.

A **class definition** thus is a description of what it means to be an object of a particular class: that is of a particular category. It defines what state objects of this category have and what operations can be performed on it – and how the objects of the category perform those operations. For example, some of my recipe books have sections that are organised along these lines. The Pocket Encyclopedia of Vegetarian Cookery for example (Dorling Kindersley, 1991), has sections on "Nuts", "Beans", "Fresh Fruit" and so on. Each is in effect a class definition. The "Nuts" section not only defines what it means to be a Nut ("[it has a] kernel containing the whole future plant in embryo") but also lists the operations explaining how each is done. "To blanch nuts...To roast nuts, ...To grind nuts..." If you need to do something with nuts, the nut section tells you how. That is basically what a class definition for nuts would do: want to train to be the Kitchen Nut expert? Then make sure you know and can do everything in the nut section.

Constructors
One of the things a class definition should tell you is how to get hold of a new object of that class. The Encyclopedia of Vegetarian Cooking does this too. The nut section contains a passage about buying nuts. You need a new nut, then follow the instructions in that passage and you shall then successfully be holding a nut. The Grains section contains a passage about buying grain. Such instructions are known as **constructors**. Given some information about its state – the values of the attributes it should start with, eg of the weight of cabbage needed, by following the cabbage **constructor** you will have an appropriate cabbage. A constructor is thus a special form of method associated with a class, rather than with an individual object. It creates an object of that class. In our restaurant example above the vegetable specialist will also specialise in buying vegetables. He will know the market stall that sells the freshest vegetables at the lowest price, for example. In doing so he is exhibiting the vegetable constructor behaviour.

Information Hiding / Abstraction
An important subject in computing and especially with respect to that of data structures is **abstraction**. Abstraction is the process of hiding or ignoring detail so as to make something clearer. It is used throughout everyday life. One of the most successful abstractions is that of the map. We use maps all the time because they are such a good abstraction. They are an abstraction of the things you would find on the ground – not every last detail is represented – a map is not a virtual reality simulation of the area it represents. Instead the map-maker picks out the details that you need to know to do the task the map is designed for. Different maps are for different things so use slightly different abstraction i.e. they hide or accentuate different features. A road atlas is designed for drivers. They do not usually worry about hills so most hills are not visible on a road atlas. A hill-walker's map is for walkers, however, and they do care about hills so contours are used to plot the hills to a very fine level. Walkers are not interested in exit numbers of major roads or whether a road is a one-way street, so those details are hidden on a walker's map.

Games and puzzles are often intended to be abstractions of real life. Chess, for example, models some of the features but not all of ancient battles. The battle involves destroying the units of the opponents and is won when the leader is captured, for example. Other aspects of real warfare such as the horrors of death are lost. Monopoly is intended as an abstraction of the property market. The buying and selling of property is modelled, but other aspects such as getting surveys done and paying solicitors are not.

The important skill of note-taking is an abstracting task – indeed a summary of a document is often called an “abstract”. What is important when taking notes in a lecture is that you get down the important points that the lecturer makes, whilst omitting the less important. If done well your notes should contain sufficient information that you can reconstruct the rest yourself. Similarly when summarising a passage read in a book for an essay, your aim is to extract the important facts that are relevant to the question you are answering, omitting less relevant points even if they are interesting. Novels and films often contain gushing abstracts written by reviewers. They abstract all the details of the plot, leaving only enough detail to convince you to read the book or see the film. Sometimes they go too far and lose so much detail that you are left without a clue what the book is about – just that it is “a novel of rich diversity that triumphantly integrates rationality and imagination”. Does that tell you details that help you decide whether to buy the book? Is it a spy novel? A romance? Abstraction is only of use if appropriate details are hidden. Sometimes it can have the opposite effect. For example, if a review contained the quote “the most interesting book I’ve read” you might decide to buy it. If you knew that the full sentence was “the most interesting book I’ve read was not this one” you might think again. An abstraction should hide details but leave behind the essentials so that the message is not corrupted.

There are several different kinds of abstraction, grouped around the kind of information that is being hidden. We will look at each in turn.

Hiding the Value
One form of abstraction is to hide the value or **abstract away from the value** of something replacing it with a label or a name. You know the attribute exists, just not what its value actually is. This is done in mathematics all the time. For example, the name pi refers to a number with approximate value 3.14. When we learn equations about circles we just learn them all using the name rather than the number. We similarly use labels to refer to particular roles. For example, if you wished to apply for a place for your child at Primary School, you would ask to speak to “The School Secretary” and you would be directed to the correct person. If by the time you were applying for your second child the person had left and been replaced by someone else, the label “School Secretary” would still get you to the right person. Similarly if you have a complaint to make about a product you bought, you could write to “The Complaints Department” and it would get to the correct person, even if you did not know who the person was. This form of abstraction again is of use because it allows the details that have been hidden, to be changed without other people needing to be aware of the change.

Fitted kitchens normally have identical doors on all the cupboards and drawers. Think of each cupboard as a variable – a place where something can be stored. The doors are to hide all the things that are in them – hiding the contents or the value. Some of the doors may even have fridges, freezers or ovens behind them. You cannot tell.

In drug trials, often different people in the trial are given different doses of a drug and some are not given any dose at all – a placebo. However, all are given an identical pill so that knowledge of the dose they are taking does not affect the outcome and so that it can be determined how much better the drug is than taking nothing. The participants do not know the size, the value, of the dose they are taking.

Hiding the Implementation

Another form of abstraction that is important in computing is that of hiding the details of the *way* a thing does whatever it does. We **hide** or **abstract away from the implementation**. For example, cars are designed so that on the whole you can drive one without having a clue how an internal combustion engine works. In fact you do not even need to know a car contains an internal combustion engine at all to drive it. You just need to know the function of each of its controls but not how they do it. Modern cars often have electronic rather than mechanical controls. However, someone who learnt to drive in one car can easily switch to another because even if the insides are different, the controls are the same – all cars have a steering wheel and a brake, for example. This is an important reason for using this kind of abstraction in programming. If the abstraction is done well, you can completely change the internals of the thing, and as long as the controls are the same, the change will not be noticed by whoever or whatever is using it.

Pianos similarly hide the details of how they work. All pianos have the same controls, and if you do not lift the lid, you do not need to know how they make the music to play one. A person may normally play a normal piano that works by plucking strings. However, they can be put in front of an electric piano that works using a completely different mechanism and still play it and again, unless they have a good ear, may not be able to tell which mechanism is being used.

We have already seen this form of abstraction. The use of methods, functions and procedures allows us to use the name of the function or procedure and so hide the details of how it is done. When we write "Make the pizza dough" in a recipe we are hiding the details of the recipe for making pizza dough – it will be on another page if we really do need to look at the details. This kind of abstraction allows us to replace instructions of *how* to do something by an instruction of *what* it is we wish to be done.

Hiding the Attributes themselves

We saw above that one form of abstraction is to hide the *value* of an attribute. Related to this is the idea of hiding **attribute** itself. Not only is the value hidden but the existence of that atribute is also hidden. We are hiding or **abstracting away the details of the representation**. By wearing dark glasses, you hide various attributes of your eyes such as their colour, but also the direction they are looking (the direction they are pointing is an attribute, whereas actually looking and taking in what is in that direction is a behaviour). That is why some people wear dark glasses indoors – to hide the direction so other people can not tell where they are looking.

Good poker players hide their emotions behind a stony face – they are "poker-faced" – they are "emotionless". When they turn up the fourth ace, there is no way of knowing from the other side of the table that they have done so. It is as though they have no emotions. The attributes corresponding to their emotions are hidden to the outside world – not just which emotion but that they have any emotions at all. You could be playing a robot. That does not mean they do not have emotions. Inside the person with all the Aces is grinning. It just does not get to his or her mouth. It just means that the emotions are internal, private attributes that cannot be seen on the outside. At the other side of the table you cannot tell what emotions the person possess or even if they have any at all. Similarly when haggling in a market, a good market trader will not let the person know the minimum they would be willing to sell it for. It is a private attribute. You may suspect that there is such a price, such an attribute in the trader's head, but you cannot be sure. For all you know the trader has not thought about it at all. Business negotiations are also like this. The two sides go into a negotiation knowing lots of details of their position, but not knowing those of the other side. Good negotiators are ones who can break through the other person's abstraction

Rich people (in films at least) often have hidden safes. A room has pictures round all the walls, but one of them has a safe behind it. To a casual observer, there are no safes, no storage spaces, in the room. This is a different situation to the fitted kitchen. There you did know there were cupboards, just not what was in them. With the hidden safe you do not even know the cupboard exists. The attribute of the room, the safe, and so its contents, is not visible. Only a small select number of people know of its existence and can access it.

Hiding Behaviours

A person wearing dark glasses hides whether or not they can actually exhibit the behaviour of seeing. They hide the behaviour of seeing. Perhaps they are blind. They can then not perform the particular operation of seeing. You may be able to infer this from other behaviour, but perhaps they can achieve the same thing in a different way. In one episode of *The Avengers*, a blind man has arranged his room so he can negotiate it by touch and by knowing the positions of things, to the point that he can walk over to his firing range, pick up a pistol and score a bulls-eye. The fact that he was "seeing" using his other senses was not noticeable to anyone else. A car has visible behaviour like moving forwards but also lots of behaviour that is hidden. For example, when driving do you ever notice the behaviour of exploding that your car is repeatedly doing? On every cycle petrol is exploded in the engine but that behaviour is hidden.

Objects and Abstraction

Objects combine all these forms of abstraction. Instance variables give a name to be used in place of the value of an attribute. Similarly methods give names to be used in place of having to write out all the commands to do the action. However, the facilities for information hiding with objects are much deeper than this. It is also possible to hide the *existence* of an attribute (ie instance variable) or behaviour (ie method) to the outside world. That the behaviour or attribute even exists at all cannot be seen outside the object – just like the poker player hiding the fact that he is experiencing a range of emotions – not just what the emotions are, but that there are emotions being felt at all. When writing a program the programmer decides which attributes and behaviours need to be seen by the outside world

Polymorphism

Often the same basic algorithm works in a range of situations. By following exactly the same instructions, except for applying them to different objects, we can get the same effect on the different objects. For example, for example "slicing" is something that can be done on lots of different things. I can slice an egg, slice a tomato or slice a loaf of bread. If I was trying to teach someone how to do basic cookery, I could teach them how to slice each thing separately. One week we would go through slicing an onion, another time we would learn how to slice a boiled egg. However, the instructions I would give would be the same each time, whether it was egg or tomato being sliced:

Instructions for slicing a given thing

1. Place the thing on a chopping board.
2. Hold the thing with one hand.
3. Place a sharp knife just off one edge of the thing.
4. Make a downward, back and forth motion with the knife

In these instructions I use the word "thing" to refer to whatever it is I am trying to slice this time. What I have done is given a reusable set of instructions that can be applied to lots of objects. This is a little like the idea of method/function arguments: we write one set of instructions then apply them to different objects – we pass different values as arguments that replace the word "thing". For example if thing is a particular egg, then the instructions become as though we had written "hold the egg...". If slicing only worked for eggs we would have written:

Instructions for slicing a given *egg*

1. Place the *egg* on a chopping board.
2. Hold the *egg* with one hand.
3. Place a sharp knife just off one edge of the egg.
4. Make a downward, back and forth motion with the knife

However, it is more than just that going on in our original instructions. As we have discussed all objects have **types**: they belong to a particular class. Any particular egg in my fridge belongs to the class of eggs: it has type *egg*. If we just gave a normal method for slicing eggs, we could use it to slice anything in the class of egg: any given *egg* could be sliced that way, but that does not mean other kinds (ie classes) of things could be sliced in the same way. To be useful a method must work on any object of the same class. The egg slicing instructions should work for slicing any eggs. **Polymorphism** gives us a way of writing methods that work for different kinds of objects: not just eggs but tomatoes and bread too. The above instructions can be followed for a whole range of classes of object. Not only can the word "thing" stand for different eggs: objects of one particular kind, it can stand for different classes of objects too. A **polymorphic** function or method is one where the class of object it works on can vary.

As another example, consider the task of searching a series of things (perhaps that are in a pile) for a particular thing you have lost. We will look at searching in more detail later, but for now it serves as an example of a situation where a polymorphic set of instructions can be written. Whether the pile is a pile of DVDs, a pile of books, a pile of papers, or a pile of tins of food, the same algorithm can be followed.

```
1. Look at each thing in the pile in turn
2. If the thing currently being looked at is the thing you are looking for
     then stop – you have found the thing you were looking for
     else  move on to the next thing and repeat this step
```

Replace "thing" in these instructions by any class of objects: DVDs, books, tins, people, then it will still result in you finding the thing you looking for (as long as it was there).

There is a related but less powerful notion to polymorphism called **overloading**. With **overloading** the same name is given to instructions for (usually related) tasks on different kinds of things. This notion is different in that even though the names are the same the actual instructions are actually different: unlike in the above example, you really need to do something different for the different objects. For example, we talk about "folding up" a blanket but also "folding up" a deck chair. However we are using one phrase to mean two completely different tasks. We are **overloading** the phrase "fold up". If I gave you instructions for folding up a blanket it would not help you any if I then gave you a deckchair to fold up. Try it:

Instructions for folding a blanket.

1. Spread out the blanket on a large surface.
2. Take two adjacent corners of the blanket and move them to touch the opposite corners.

If this was just the same as polymorphism, I would be able to take the above instructions and replace the word "blanket" which indicates the class of things the instruction applies to and replace it with "deckchair" which refers to a different class of objects and get instructions that work:

Instructions for folding a deckchair.

1. Spread out the *deckchair* on a large surface.
2. Take two adjacent corners of the *deckchair* and move them to touch the opposite corners.

This makes no sense. **Overloading** is different to **polymorphism**. I cannot just replace the word "blanket" with "thing" and get a set of instructions that apply to different classes of thing. Therefore I can not write one set of instructions on how to fold things that can apply to both deckchairs and blankets. For any given set of "fold-up" instructions I would need to say what it was designed to apply to.

A truly **polymorphic** function would apply to absolutely any object: you could replcae the word "thing" with anything and the instructions would work. In practice, most sets of instructions in real life only work for related classes of objects. The slicing example only really works for food (and then only particular kinds) – it would not as it stands work for slicing planks of wood, for example. This leads to a further concept: **inheritence**.

Inheritance

When learning to be an expert in a particular area, it is usual to start by learning the general things that are relevant to the whole area. For example, if training to be a Heart Consultant in a hospital, you would first learn the same general medicine as someone who was to be a GP or a Paediatrician. Only later do you specialise. When studying to be a Computer Professional, you are likely to study many common things such as networks, programming, ethics and so on. In the second and third year you will probably specialise more and more. An advantage of structuring educational programmes in this way is that single modules can be used on a whole range of degrees. Everyone does the same first year modules. They are reusable. Designing a new degree? Then you can just use the existing modules. You do not to design a new first year – just use the existing one.

In the second description of the kitchen at the start of this chapter we suggested people were allocated to single kinds of objects: a cabbage expert or a carrot expert. More generally someone might be assigned to a whole class of things where similar skills are needed – vegetables rather than just cabbages. This is sensible since much that you learn and skills you acquire about one vegetable apply to others. For example, knowing that if you need to steam or boil vegetables it is best done in a microwave is a general fact that applies to a whole range of vegetables.

The idea of **inheritance** is similar. We define general categories of things – defining the attributes and behaviours that are common to anything in that category, then as we need more specialist objects within the category we can build on top of the basics. We create new categories that **inherit** the properties of the more general ones. The general classes can be reused in different programs. When programming we are writing instructions rather than just following some we have learnt, so the issues are about how you structure the way they are written down. The programmer is more in the role of the Academic designing the course rather than the student following it.

We previously discussed categories in the context of types. We argued that splitting things into classes is almost a natural human instinct. However we can go further, humans do not just divide the world into categories, they also divide those categories into sub-categories. The world contains animals, but animals can be mammals, birds, insects etc. Mammals can be marsupials or non-marsupials. Marsupials can be broken into categories such as Kangaroos, Kangaroos can be broken into Red Kangaroos or ... As we get to smaller and smaller categories, the attributes and behaviours become more specialised. Notice that we are talking about the categories: the classes of objects now, rather than any particular object. A particular kangaroo that we meet in the street will be an **instance** of all the classes. Skippy the Kangaroo is a Red Kangaroo. She is also a Kangaroo, and a Marsupial, and a mammal and an animal. At different times we might wish to refer to her as being in any one of those classes. For example in a discussion of rights: "Should she as an animal have the same rights of life and liberty as we assume humans have?", we are happy to talk of the whole class of animal. In a discussion of reproduction we might talk just about her being a marsupial.

Similarly shapes can be split into circles, triangles and rectangles. These categories can be broken down further. A square is a form of rectangle. It has all the attributes of a rectangle – 4 sides, 90˚ angles and so on. However it is a more specialist case of the rectangle – it is a rectangle with all the sides the same length. Any particular square is a rectangle. However there are rectangles that are not squares. Thus the type-subtype relationship is one-sided. A subclass inherits the attributes of the class, but not vice-versa.

Notice that this idea of inheritance is different to the notion of an object having other attributes that are objects. Then we were talking about for example an eye being a part of a person, but that did not mean a person was an eye, or an eye was a person. However, a Kangaroo *is* an animal. Then we were talking about objects and their parts. Now we are talking about categories and how everything in a given category might also be in some other category of object.

In computing jargon we talk of defining **sub-types** of **types**. Classes are just a form of user defined type. Inheritance is about defining types in terms of more general types: that is defining classes from less specialist classes. We define a general type first, then define a more specialist sub-type by describing more specific attributes and behaviours.

Reuse
One of the most important issues when writing programs is that of reuse of code. Writing programs is time-consuming, expensive and difficult to do without making mistakes. One of the ways to avoid this is to reduce the size and number of programs you write by writing them in a way that allows reuse. Object-oriented programming is also primarily about making it easier to reuse code and many of the features covered in this chapter are important because they facilitate reuse..

Summary
Abstraction is concerned with making something easier to understand, change or use by hiding details. There are many different forms of abstraction. Examples include hiding the value of things, replacing them with a label, and hiding the details of the way something works.