

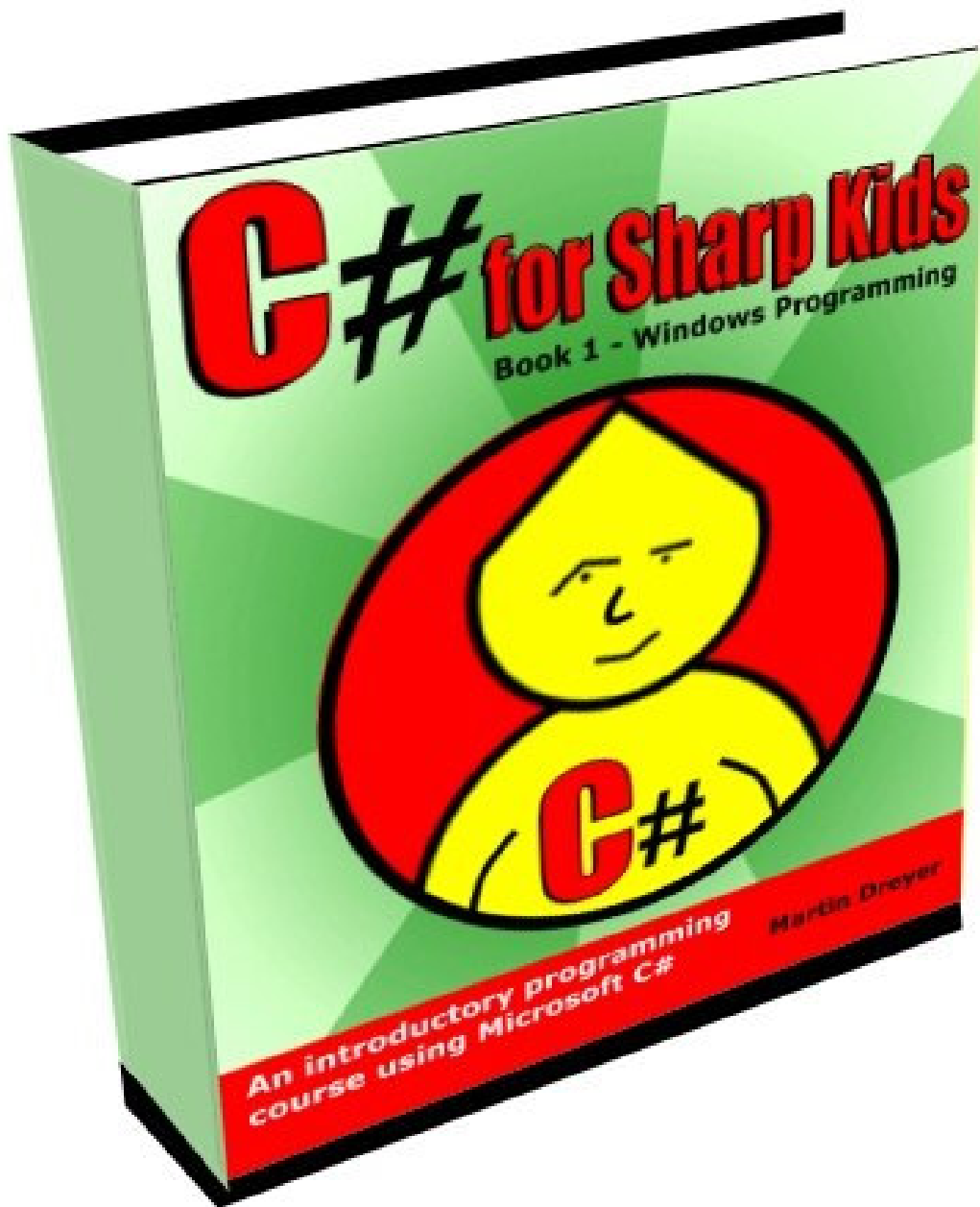


Table of Contents

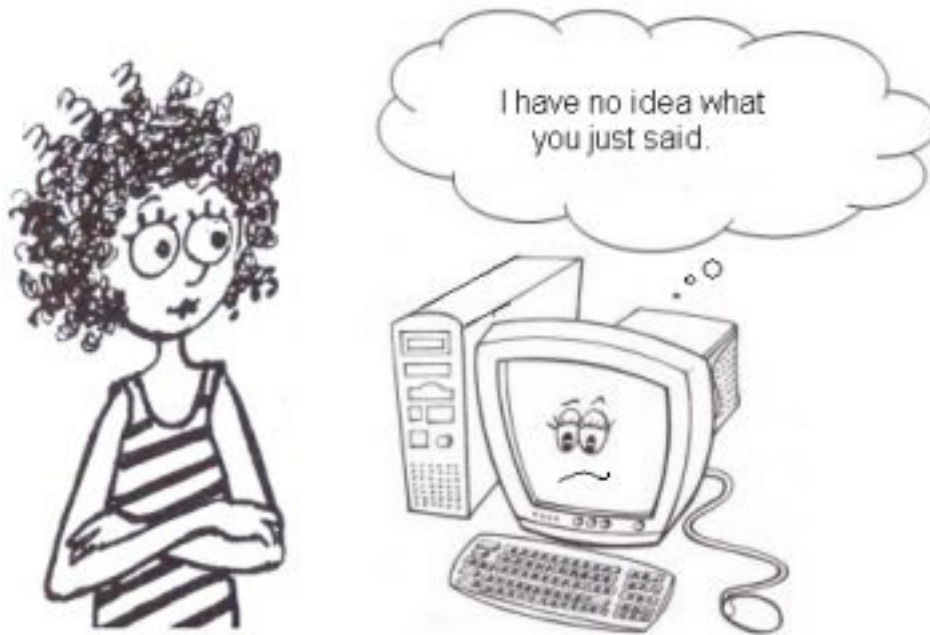
Part II – Learning to Communicate	3
A Shared View of The World: The Alien, the Elephant and the Coconut Tree	4
Computers Are Just Like Aliens	9
The Object-Oriented Approach Summarized	10
Classes	11
Objects	13
Fields and Their Types	17
Types	18
How to <i>Use</i> Fields	23
Methods	28
More About Methods	33
Parameters	33
Methods that Send Values Back	36
Public, Private and Protected Methods	39
Constructor Methods	41
Events	43
The Button Click Event	45
Mouse Events	47
Name Spaces	49
Inheritance	53

Part II – Learning to Communicate

The basic problem we face when we write computer programs is the fact that we and the computer were not raised by the same parents. Perhaps if we were, we would have gained a similar understanding of the world around us and also been taught the same language.

That would have made it really simple to speak to each other with understanding. We could simply write down "computer, please draw a picture of a cat" and that would be that.

"So teach computers to understand us" you say. But computers, being constructed very differently to the human brain, have a really foreign way of seeing things - and teaching them to think like we think is not easy.



The solution is to find some middle ground:

- a common "view of the world" and
- a common "language"

that both humans and computers can comprehend.

This part of the book is all about learning to communicate with the computer. You have to twist your mind a little to see things from the computer's point of view. This is a give-and-take exercise - the computer will be

taught a whole bunch of things (like some English words) from your world, but you'll need to learn some things from its world too. Try to climb inside the head of the computer and imagine how you would like this human to explain what it wants.

A Shared View of The World: The Alien, the Elephant and the Coconut Tree

Computers have no way of knowing what is in our heads, what we want them to do or how to do it. We may dream of a day when we'll glance in their direction, raise an eyebrow, and they'll understand, but back here in the real world we still have to spell out very clearly what we mean and even how the computer should go about helping us. In short, there is a major communication gap between humans and machines. If we want them to help us, we need to bridge this gap.



Now **forget about computers for a while** and imagine that you, instead, desperately needed to talk to a particular alien creature. Here's how the story goes...

- As your luck would have it, something went badly wrong in the universe and, goodness knows how, you and a strange alien being have ended up stranded on a tiny, deserted island in the middle of the ocean. After giving up with "Hi, my name is Sally, what's yours?" you eventually decide you'll just call it "Algy".



- As hunger sets in, you observe some coconuts in a tall tree and realize you'll need Algy's help to get them - and that's not going to happen unless you two learn to understand each other.

You start by drawing a coconut tree in the sand and, pointing at it, you very clearly speak the word "co-co-nut". You are astounded as Algy immediately responds "co-co-nut"! You proceed to draw more pictures and, every time, Algy responds perfectly. You have discovered that this creature has a real knack for languages. Before nightfall, Algy has mastered the English language.



- Thrilled at your progress, you now look Algy in the eye. "Algy, please hoist me up on your shoulders so I can pick the coconuts and eat them".

He looks at you blankly. He looks at the picture of the coconut tree blankly. Back to you. Back to the picture. No go - he's not getting it. Slapping yourself on the head, you realize your foolish mistake and point wildly at the *real* tree. But Algy still doesn't get it. In fact, it turns out, he's only good with pictures. Aaaargh! It slowly dawns on you that although he is able to use your language, he actually has *ZERO understanding* of the real things that you're talking about.



- Somehow, you need to translate between his "view of the world" and yours. No problem, you think - I taught him a language - I can teach him about these objects in the real world. Three days later the coconuts are still hanging. The lessons are not going well, and you're losing your grip as consciousness slips away from you.

You're not sure whether you're dreaming, hallucinating or having a vision, but a figure dressed in white appears in a bright and misty light. "Here is a great way to view the world around you" a voice booms. "A view that both you and Zniff12 could understand". (You guess this must be Algy's real name.) You lose all track of time as images magically penetrate your mind, teaching you about this way to describe the world. Eventually, with a peaceful look on your face, you emerge from this timeless state.

- Snapping out of it, you grasp Algy by the shoulders, "I've got it buddy, listen real carefully to what I'm going to tell you." Algy frowns as you proceed.



- "The world around us has different **classes** of things - plants, animals, hungry humans, aliens, etc."
- "Right now, I'm only interested in talking to you about the actual **objects** on this island. In particular, in case you hadn't noticed, I'm very interested in this coconut tree (which by the way is in the **class** called "plants"). I'm also interested in that elephant over there - gee, it's funny I didn't see him until just now." (You point to each of these as you speak).
- "The elephant is in the **class** called "animals" and has some **properties** that are important to us, Algy. Firstly all animals can *move*. Secondly, this elephant animal is really *heavy* - I reckon about 2 tons - that's going to be useful. Thirdly, he's bad tempered - that's going to be useful too.

- "Okay Algy, stay with me now ... In the world around us, **events** are occurring all the time. I become hungry - that's an event. The coconuts ripen - that's an event. Sometimes we can trigger an event ourselves, in order to make some **action** happen. I have a great idea - let's trigger an '*elephant disturbance*' event that gets that elephant to help us with the action '*get the coconuts down*'. "
- "Let's split the work. I'll take care of triggering the **event** (I'll disturb the elephant so he looks at you), and you get him to get the coconuts down, Okay?" *At this, Algy looks sheepishly at you and admits* "Sally, I hear what you're asking, but I just don't know *how* to do it". You respond with a pat on his back, "Algy, not to worry - I'll never ask you to do anything without first explaining the **method** of doing it". Here are the steps to be carried out when I disturb the elephant and say to you "get the coconuts down" :
 - Run out from behind this rock and stand alongside the tree. The elephant will charge at you immediately.
 - Precisely when he reaches the line I've drawn in the sand, run for your life (go sideways - forwards or backwards would be a bad idea).
 - He'll hit the tree and will be dazed for a few moments, but the force will knock the coconuts down, and we can collect them later.
 - Finally, come back to where you started and report on how we did.



- Algy pauses for a long time, apparently digesting your lesson, before unexpectedly whipping out what is clearly an advanced communication device. He speaks into it for a few moments in a modem-like voice and then slips it back into its pouch. "What was that all about, Algy?" you ask. Algy responds "Allow me to translate Sally ...
 - There is a **class** of things out there called X-Zniffers.
 - I am one **object** of that class, but there are others.
 - All things in that class have the **properties** of
 - name (mine, for example, is Zniff12)
 - earShape (which always seems to have the value "tube")
 - likeToWatchTv (which is true in most cases but false in others)

The truth is, my landing here was no accident. You're on a really popular TV show called 'Survivor'. But I just called up my producer to tell him I don't get paid enough for this. He's going to run the **method** SendRescueCraft immediately."

Okay, by now you're onto the fact that I've thrown you a rather bogus story. The point is that you and Algy have developed a common way of describing the world - you both use the same terms and both have the same understanding of what they mean. This, along with the fact that you share a common language, allows the two of you to communicate.

Computers Are Just Like Aliens

Imagine now that instead of trying to explain things to the alien you need to explain them to a computer. You'll have a similar experience to the one above. And you'll be able to solve the problem in a similar way - by agreeing on some common way of viewing the world that both human and machine can understand.

First of all, let's make it clear that the approach using "classes, objects, properties, events and methods" described in the story above is not some perfect, shining truth. It's just one way that we could choose to explain the world around us to a foreign creature. The important thing is that both of you agree to describe the world in the *same* manner; But that way of viewing the world, which by the way is called the "object-oriented" approach, has become very popular as a way of describing things to computers, and in fact C# insists that you use the object-oriented approach.

So learning C# involves learning how to describe things using classes, objects, events, methods, etc.

To get the computer to help you do useful things:

- **You** (the human) learn the C# language and how to describe things in an object-oriented way.
- You install the C# software, and some other related software - now **the computer** too understands both the C# language and how to work with things that are described in an object-oriented way.

And there we have it. You and the computer share a common view of the world as well as a common language and will soon be communicating like old buddies.

The Object-Oriented Approach Summarized

Let's explain clearly what we mean by object-orientation, through a programmer's eyes.

- The world around us can be divided up into different **classes** of things such as “cats” or “houses” or “trees”. These classes can be thought of as categories or labels for grouping similar things. We need a way to describe these classes to the computer so it can solve problems about them.
- In each **class** of things are some specific **objects** that we are interested in. For example, within the class “cat” is my cat Fluffy who is an object of the cat class. Similarly, the pink house on the corner of my street is an object of the “house” class and the oak tree outside my bedroom window is an object of the “tree” class. When there is a specific object we want the computer to work with, we must write some description down for the computer that represents that object.
- Objects have a variety of **properties**. Properties are the things that describe the objects more clearly. If you remember your English classes, properties are like adjectives in the English language. For example, my cat Fluffy may be described with properties such as *height*, *weight*, *color*, etc.
- **Events** are things that happen to an object. In the real world, events occur around us all the time. For example, when I pet my cat Fluffy, that’s an event that happens to Fluffy. Similarly, the wind blowing through the branches of the oak tree outside my window is an event that happens to the tree. Events are usually actions that affect the specific object, but which the object has no control over. In the computer's world, the events are usually things like "a button was clicked" or "the mouse was moved".
- **Actions** are things that an object does. For example, my cat Fluffy licks her fur – that’s an action. In the computer’s world, sometimes when certain interesting events occur, we want to fire off certain actions, like "when this button is clicked, draw me a picture of a cat".
- Computers, believe it or not, are really dumb. A computer has no idea how to draw a cat. So we have to write down, in a way the computer can understand, the **method** for drawing a cat. This is a set of step-by-step instructions that tells the computer how to do it, like a recipe.

In the following chapters, we take each of these aspects of object-oriented programming and explain them in more depth. We also show precisely how to write them down in the C# language so that the computer can understand what we're describing.

Classes

We can choose to divide the world around us into various groups of things – let's call them classes.

As one obvious example, consider the classes at a school; children of a particular age may be grouped together and put in one class, or perhaps children of a particular ability may be grouped in a class. This allows them all to be treated in a similar fashion. They will all learn the same set of things and will all be tested in the same way.

That doesn't mean everyone in a class is the same – it just means there are certain things they have in common.



As another example, we could choose to group all living things into two classes, named “plants” and “animals”.



Now getting back to expressing ourselves in such a way that the *computer* understands what we're trying to say ...

In C#, we describe a class to the computer like this:

```
class Animal
{
}

```

From just this little bit of code it now knows:

- we want to talk about a **class** of things
- we will be calling that class **Animal**
- everything we tell it about this class will be written between the curly braces { ... }.

While calling your class "Animal" will certainly make sense in a program that displays pictures of animals, or stores interesting facts about animals, it is also common to have class names such as "MyPhotoProgram" or "MyGradeCalculator". So you must realize that classes don't necessarily work with things that exist in the *real* world - you name your class in a way that describes what it's working with.

Building Block : Classes

All C# programs are built inside **classes**. The basic structure of a class is written down for the computer as follows :

```
class Animal
{
}
```

The word “class” is known to the computer and must be written in small letters, but the name that you give to your class may be a word of your own choice and may be written using a mixture of small and capital letters. It may not contain things like spaces.

Although we should write our code on separate lines to make it clear and easy to read, the computer will be just as happy if we write the above piece of code as follows :

```
class Animal { }
```

More about what goes *inside* the class description (between the curly braces) a little later. But first, in the next lesson, we’ll discuss objects.

Objects

In each class there are some actual objects. In the class of things called “Animal”, for example, there are some specific creatures:

- Your aunt’s cat “Fiddlesticks”
- A cow you saw in a field once, whose name you don’t know
- A gorilla named Jeremy
- etc.



In the previous lesson, you learned how to begin telling the computer about the `Animal` class in general, but at some point you will probably want to tell it something about one or more specific objects. For example, “there is a particular creature in the animal class which I call `FiddleSticks`”.

We can express this to the computer as follows:

```
Animal fiddleSticks;
```

This is so short that you have to actually stop and think or you’ll miss the point entirely. The computer understands this to mean “This human wants to talk about something in the `Animal` class and wants to call it `fiddleSticks`". I have no idea what `fiddleSticks` means, but if that’s what the human wants to call it, that’s fine by me.”

We need, however, one more statement to follow that one:

```
fiddleSticks = new Animal();
```

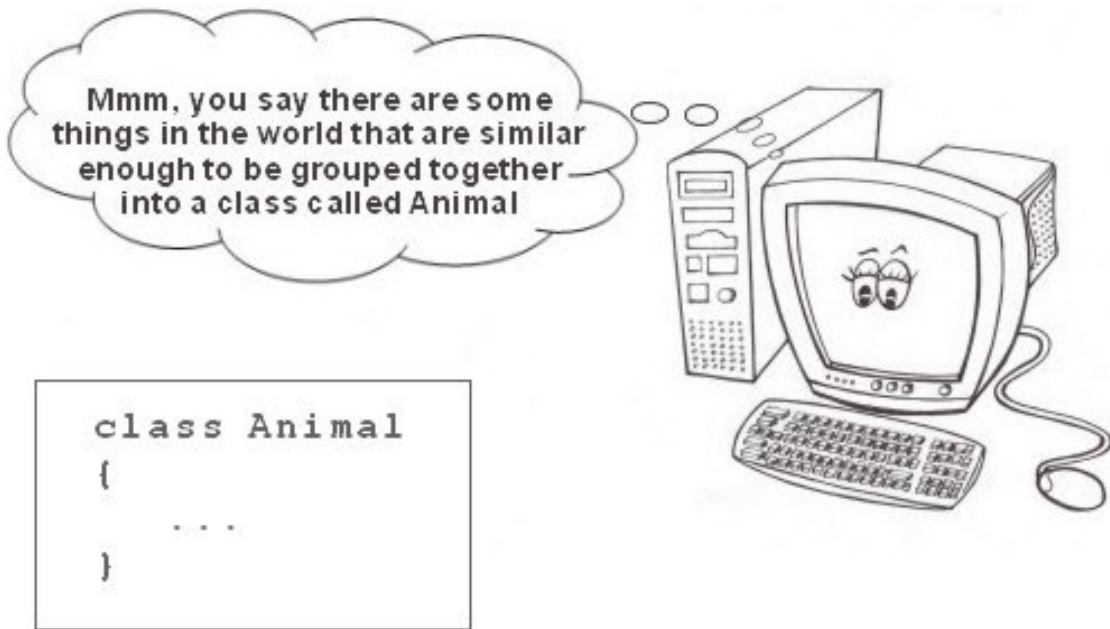
It may seem strange that we need to do both these things. But from the computer's point of view it makes sense. Put yourself "in the mind of the machine" and think of it like this:

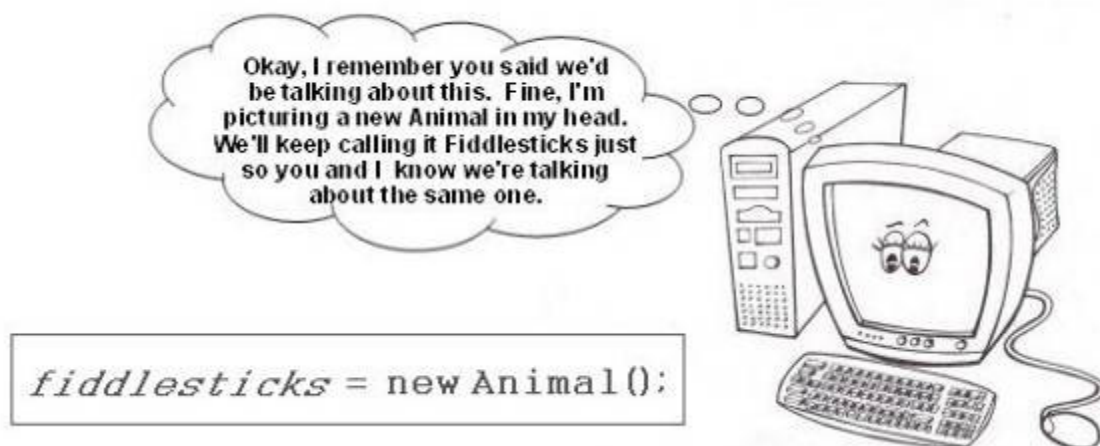
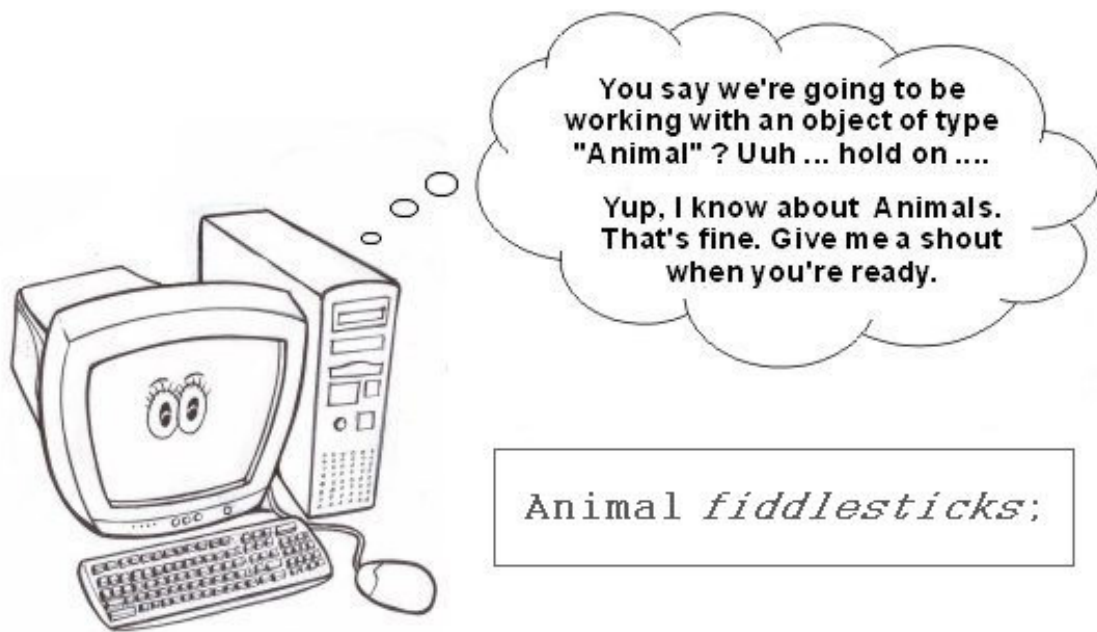
- The first statement warns the computer that, sometime soon, we're going to ask it to work with a specific individual in the class "`Animal`". The computer now has a chance to check whether it knows about the "`Animal`" class. It runs through your whole program and sees whether you have somewhere described a class called "`Animal`". If not, it can warn you "hey, I'm not going to be able to do that". If it *does* find what it needs, it just smiles quietly and carries on, keeping in mind the fact that you'll be

asking it to do something later, with the Animal named fiddlesticks.

- Then, at some later point, we add the second statement, which says to the computer "Hey, remember that fiddleSticks object, I need you to work with it now. Make some space in your head to think about this Animal." The computer imagines a new "Animal" object in its head and sees that you're talking about the same one (fiddlesticks) that you mentioned earlier. Having already read up on the "Animal" class, it already knows some basic stuff that must be true about fiddleSticks and it holds this information ready in its head, hoping you'll either tell it some more or ask it some questions about fiddlesticks.

Let's put all 3 things together now and picture how the computer "thinks" about them





You should take note that in the first block, we're describing the class of things in general, while in blocks 2 and 3 we've singled out a particular object within the Animal class.

Building Block : Objects

Objects are specific individuals of some class.

The class just describes what “all of these things are like” in general. It is the basic plan for creating an object.

```
class Animal
{
}
```

If we are planning to work with a particular object, we must tell the computer (up front) what class it belongs to - so that it can check first whether it has knowledge of that class. We do this by specifying the class that it belongs to and then giving the object a name of our choice.

Class name	Object name
↓	↓
<code>Animal</code>	<code>percyThePorcupine;</code>

This is called "declaring the object".

And when we are ready to actually work with the object, we must ask the computer to "create an instance" of the class. Now it uses its knowledge of the class to create a 'real' object that it can do things with.

```
percyThePorcupine = new Animal();
```

This is called "instantiating the object".

In this example, "percyThePorcupine" is now an instance of "Animal". Another way to say this is percyThePorcupine is an "object" of type "Animal". Either way, we know clearly that we're talking about a specific Animal now.

Now we know how to talk about a specific object in a class, but because the class we've been defining is rather simple (okay, I admit, it's entirely empty), there hasn't been much of interest to work with. So in the next lesson, we will add some stuff to our class.

Fields and Their Types

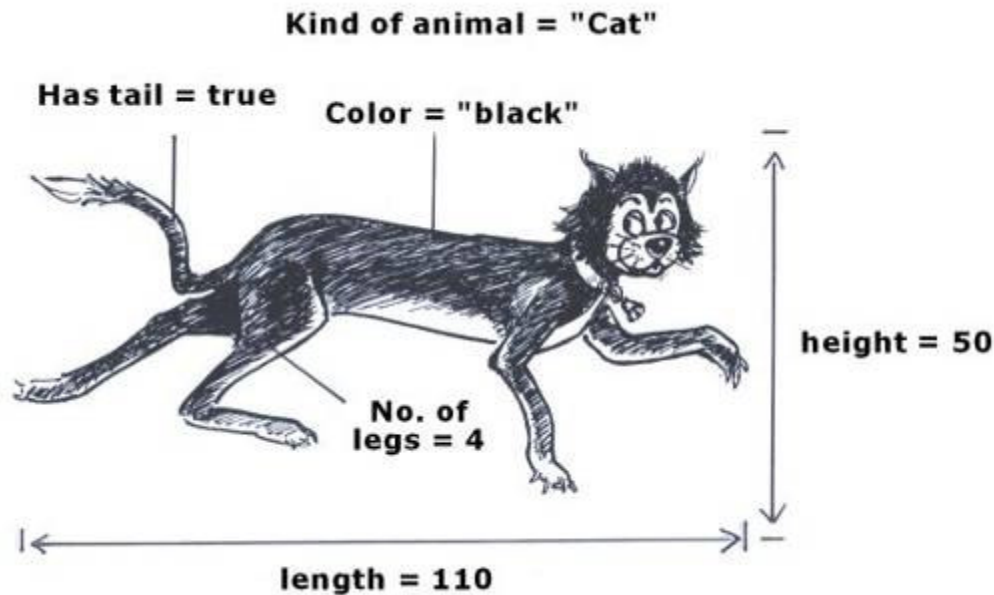
If I asked you to list some properties of animals, you might come up with things like this:

- Kind of animal
- Height

- Length
- Number of legs
- Color
- Has a tail
- Is a mammal

And when you talk about a specific one of these animals, you can then study it and give **values** to each of its properties

- Kind of animal = "Cat"
- Height = 50 cm (he's a big cat!)
- Length = 80 cm
- Number of legs = 4
- Color = "Black"
- Has tail = true
- Is mammal = true



In C#, these properties are usually called **fields**. (The word "properties" is kept for a slightly different purpose, which we don't really need to go into here).

Types

Now here comes something that we don't have to worry about much in the real world but which matters very much to computers: Although all the properties above may look like the same kind of thing, there are actually different types of fields. Let's look at some examples:

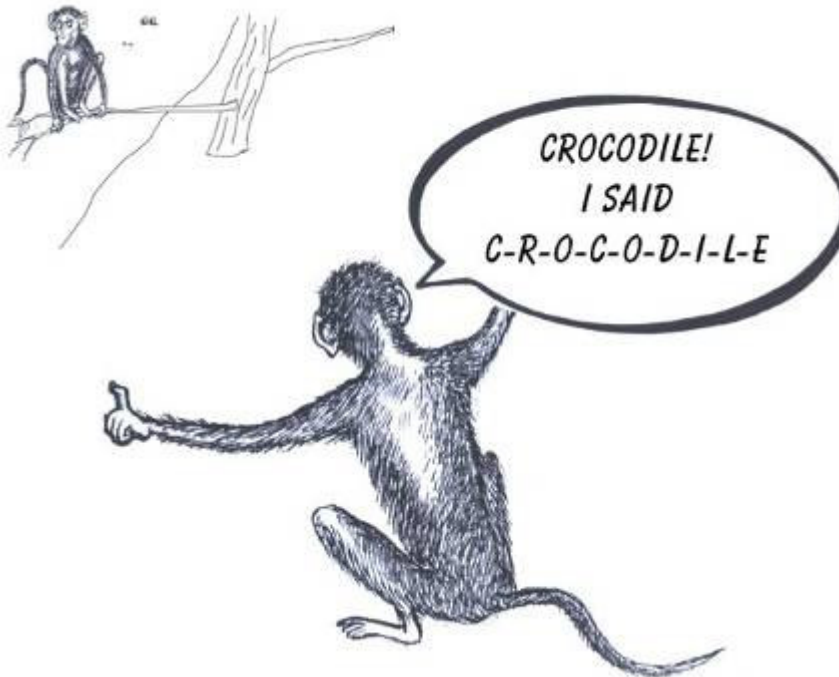
- The values above for *height*, *width* and *number of legs* are **numbers** – usually things we might want the computer to do some mathematics with. There are many situations in which this is the case.



For this situation, you could decide to use three fields named something like: "numberOfBananas", "numberOfPeople" and "numberEachOneGets". Since you need to calculate something by mathematical division, you must make these number types.

Note : In C#, field names may not contain spaces – stick to simple characters like letters and numbers..

- The values for *kind of animal* and *color* are made up of **letters that are strung together** - in this case the letters making up the words "cat" and "black".



It's important to take note, though, that strings can contain number digits too. You might, for example, store the following in a string field: "That's the 17th crocodile I've seen today!", which has alphabetic letters, spaces, number digits and punctuation marks. But you can't perform any mathematical operations on the numbers in a string!

- A field like *has tail* will always be given a value of either **true** or **false**, as will any "yes or no" type fields.



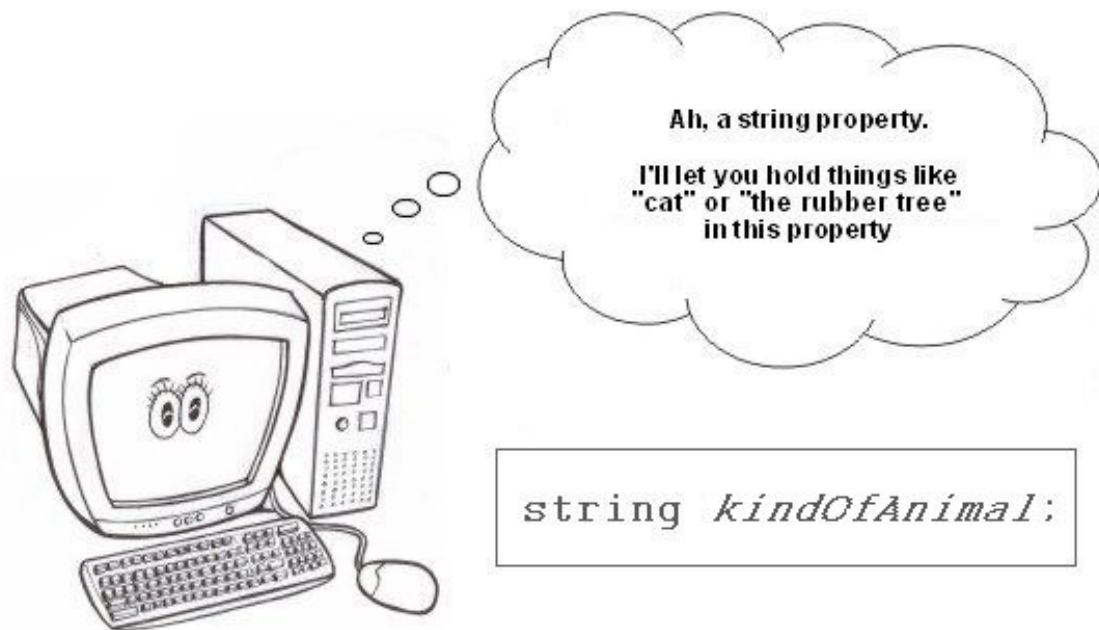
Computers work with each of these different “types” in special ways, so when we define fields we’re expected to tell them what field type we want them to use. In one program we may tell it to treat *number of legs* as a string of letters, because we’re not interested in doing any mathematics with that property, while on another occasion we may make it a number because we need the computer to work out how many legs there are if we have 500 of these animal objects.

The String Type

The way to tell the computer you want it to treat some property as a string of letters is simple:

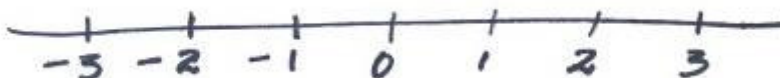
```
string kindOfAnimal;
```

We're not yet putting any value into this field - just telling the computer what kind of data we're going to work with and giving the field a name. Put yourself in the computer's head and imagine how you'd respond if a human wrote that down for you.



Number Types

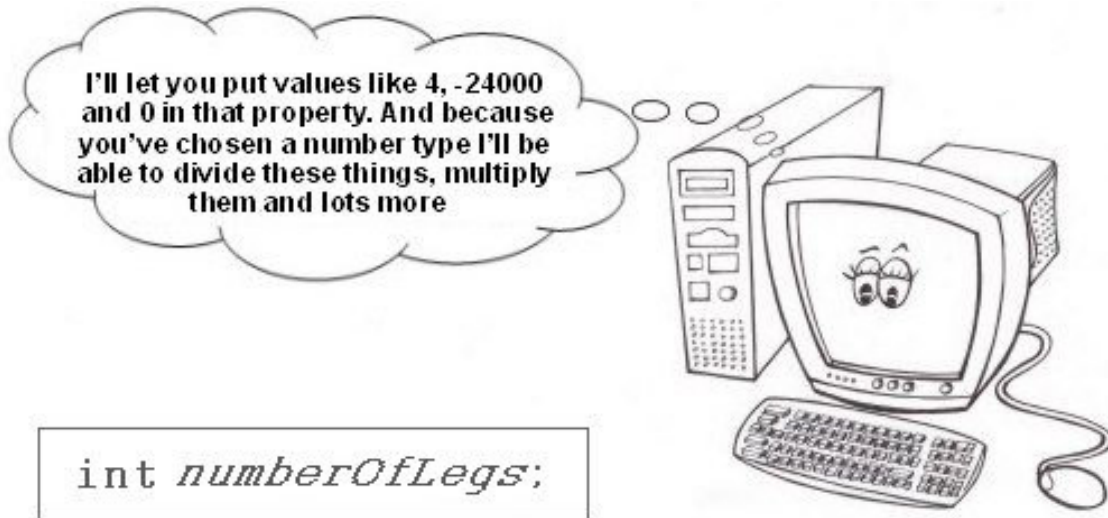
To tell it to rather treat a field as a number, you could specify any one of several number types that C# knows about. One of the most common number types is the *integer*. If you recall your Math lessons you should know that an integer is really just a whole number, and it can be negative, positive or zero.



An easy way to express to the computer the fact that you want a field to be of the integer type is as follows:

```
int numberOfLegs;
```

And this is what our digital friend will think...



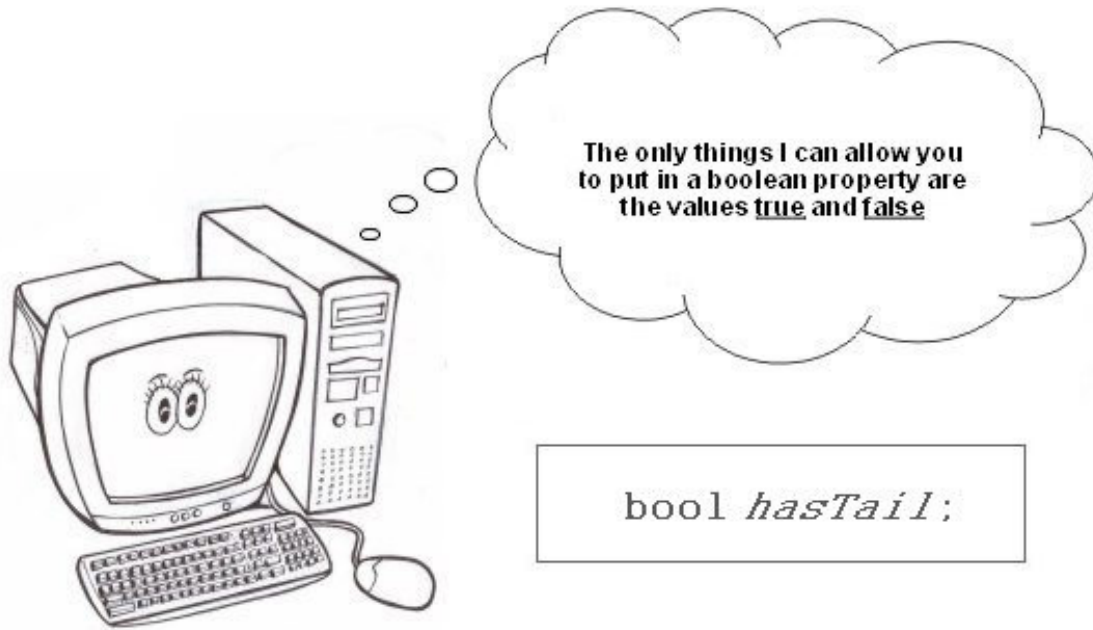
The Boolean Type

If you know that something is always going to be “true” or “false” and never have any other value, it’s useful to use C#’s true/false type. This has a rather strange name, “Boolean”, named after a mathematician called George Bool, but it really is the simplest of all.

You could tell the computer you want it to work with a field called “hasTail” as a “true/false” type like this:

```
bool hasTail;
```

And the machine will be thinking...



How to *Use* Fields

So fields are the pieces of information that allow us to describe things in a class better. And fields are always of some particular type.

Adding Fields to the Class

Previously, all we told the computer about “Animals” in general was that they were called “Animal”. We included some curly braces and promised to later put more information about “Animals” between them.

```
class Animal
{
    ...
}
```

Well, now we’re ready to do that. Let’s add some fields to our class definition:

```
class Animal
{
    string kindOfAnimal;
    string name;
    int numberOfLegs;
    int height;
    int length;
    string color;
    bool hasTail;
    bool isMammal;
    bool spellingCorrect;
}
```

(to keep your code neat and readable, it's usual to **indent** the fields as shown above. To do this, press the TAB key on the keyboard. You can also select a few rows and indent them all with one press of the TAB key.)

Using Fields with Specific Objects

Okay, so we've explained to the computer that ALL Animals have these fields. Now, let's ask it to create an instance representing a specific one of these animals (as we did in the previous chapter) and then let's tell the computer some things about that specific animal - by giving values for each field.

```
Animal fiddlesticks;  
fiddlesticks = new Animal();  
  
fiddlesticks.kindOfAnimal = "Cat";  
fiddlesticks.name = "Fiddlesticks The Cat";  
fiddlesticks.numberOfLegs = 4;  
fiddlesticks.height = 50;  
fiddlesticks.length = 80;  
fiddlesticks.color = "Black";  
fiddlesticks.hasTail = true;  
fiddlesticks.isMammal = true;
```

Notice that, for *string* fields, we always put the value inside *quotes* (""), but *not* for the other types. And we put a semicolon ";" at the end of each statement.

It's worth reminding you at this point, that the object name we chose, "fiddlesticks", is of no sentimental importance to the computer. It will work just as happily with this object if we replace all the "fiddlesticks" names with the name "X", or any other name of our choosing:

```
Animal X;  
X = new Animal();  
  
X.kindOfAnimal = "Cat";  
X.height = 50;  
X.length = 80;  
X.color = "Black";  
X.hasTail = true;  
X.isMammal = true;
```

Building Block : Fields

A class normally has one or more **fields**, which allow us to describe the detail of things of that class.

```
class SchoolKid  
{  
    string firstName;  
    string lastname;  
    int age;  
    bool isWideAwake;  
}
```

Although the fields can be written against the margin, it is good practice to indent them one tab space toward the right by pressing the TAB key on the keyboard. This makes the code easier to read.

These fields are always of a particular type. There are many different possible types, but here are some common ones :

Type	What it means	Examples of values
string	Mainly holds things like words and sentences, which are made up by “stringing” letters of the alphabet, and some other symbols, together	“Sam” “Eight-legged Octopus” “Girls are too clever to fall out of their prams”
int	Short for “integer”, which is a kind of whole number.	328 28000 -520
bool	Short for “boolean” which really comes down to meaning “true or false”	true false

When you define a field in a program, you must enter a *type* and a *name* for the field :

```
string lastname;
```

The *type* must be entered exactly as the computer knows it – the basic types above (like int) are all written in small letters - some other, fancier types will have some capital letters though.

The field’s *name* is a word of your own choosing like “isWideAwake” and can be written with both small and capital letters. Things such as spaces are not allowed.

When you work with a specific object, you normally assign values to the fields, which help to describe the object. If, for example, we have an object named *schoolKid*, we could give values to its fields as follows :

```
schoolKid.firstName = "Jiminy";  
schoolKid.age = 13;
```

We separate the object name and field name with a period “.”.

So `schoolKid.firstName = "Jiminy"` means “put the value 'Jiminy' into the *schoolKid* object's *firstName* field”.

Private, Protected and Public Fields

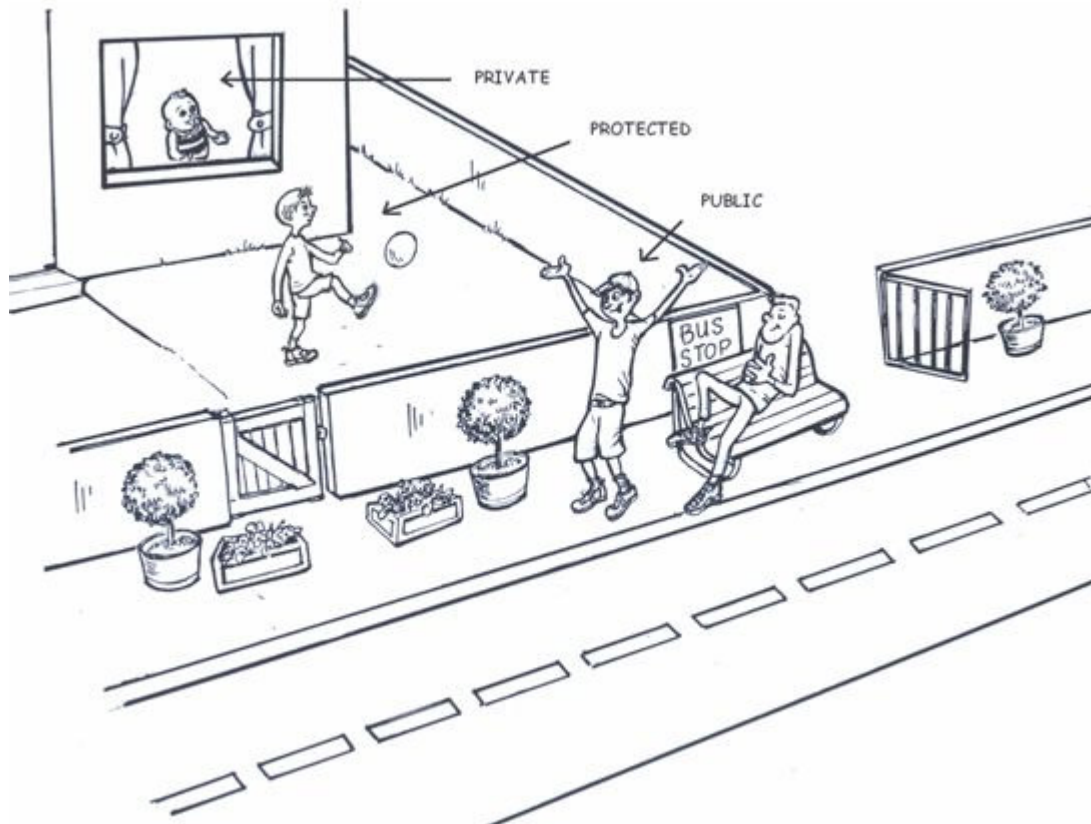
This topic won't be too important while you're first learning C#, but you may see words like "private", "protected" and "public" at some later point and wonder what they mean. So let's talk about them.

```
class Animal
{
    string kindOfAnimal;
    string name;
    int numberOfLegs;
    int height;
    int length;
    string color;
    bool hasTail;
    bool isMammal;
    bool spellingCorrect;
}

class MyTestClass
{
}
```

It is possible to have things from one class, such as *MyTestClass* above, **talk about things in another class** like *Animal*.

When fields are declared as above, **we can decide whether we want to allow other classes to access them or not**. Sometimes it's quite important to make the fields "private" because, for example, another class might put some nasty values into the fields and break the program. Setting a field somewhere in the range from private through to public is about how safe we need our class to be.



Notice the different protection levels of the kids in the family portrayed above. Sally, the baby, is being kept *private* – she is not being allowed out of the house and enjoys the highest level of safety. Only those who are most trusted can interact with her. Billy has the freedom of the front and back yards too but is still somewhat *protected*. Tom, the eldest, is *public* and so anybody else out there can interact with him in whatever way they choose.

So we can add a word at the beginning of the field declarations, as follows:

```
class Animal
{
    public string kindOfAnimal;
    public string name;
    public int numberOfLegs;
    public int height;
    public int length;
    public string color;
    bool hasTail;
    protected bool isMammal;
    private bool spellingCorrect;
}
```

You might wonder what happens if we leave these words out. The answer is that the computer silently treats it as if you'd written "private" in front of it since "private" is the default setting. So the *hasTail* field above will automatically become private.

What do these words mean?

- **Private** means "only things in this class can access this field"
- **Public** means "any class can access this field"
- **Protected** means "only closely-related classes can access this field"

The following two classes illustrate how **public** and **private** work in practice. We're not going to discuss protected further here, but you can read up in the Visual C# Express help if you're curious about it,

We first define, as before, some private and public fields in the "*Animal*" class and then try to access them from the "Zoo" class.

```
class Animal
{
    public string kindOfAnimal;
    public string name;
    public int numberOfLegs;
    public int height;
    public int length;
    public string color;
    bool hasTail;
    protected bool isMammal;
    private bool spellingCorrect;
}

class Zoo
{
    Animal a = new Animal ();

    // The following line will succeed because the "Zoo" class is allowed
    // to address public fields in the "Animal" class

    a.kindOfAnimal = "Kangaroo";

    // The following lines will both FAIL because the "Zoo" class
    // is not allowed to address private or protected fields

    a.isMammal = false;           // Try to access a protected method
    a.spellingCorrect = true;      // Try to access a private method
}
```

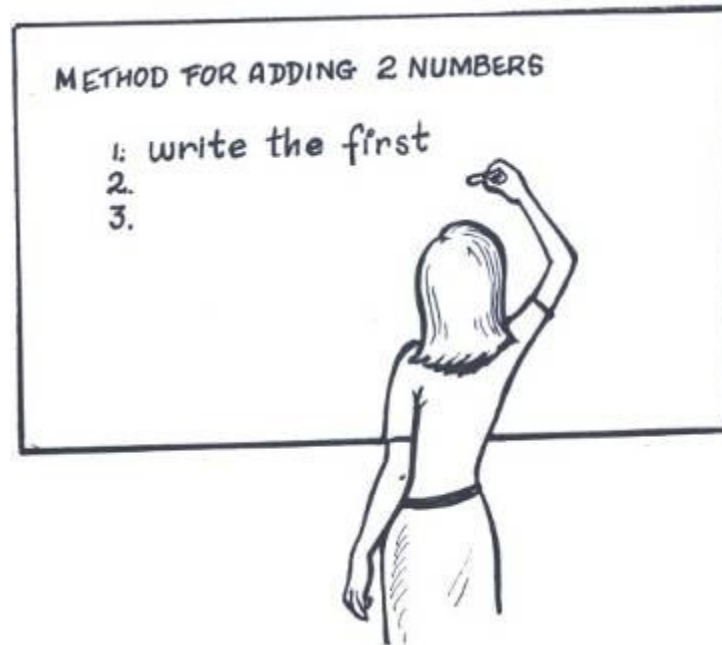
Methods

Programs would be rather useless if they could only allow us to describe objects. Clearly we want them to "do things" too:

- Write to the screen

- Calculate an answer to a problem
- Copy data from a website
- Brighten up a photograph
- ... and a thousand other things

Now think back to when your teacher taught you to add numbers. Most likely he or she first taught you **HOW** to add numbers – probably showing you some steps you should follow. In other words you were first taught the **METHOD** of solving this kind of problem. A method is a set of step-by-step instructions explaining how to do something.



Then, once you have a method for solving problems of a particular type, you can be given all sorts of different problems of that type and you will be able to solve them – by following the steps in the method.



Similarly, if we want the computer to do things, we must write code to tell it how to do them – in other words, we must describe the method it should use to achieve these things. It can then use that method each time we give it a problem of that type.

In C#, a method is a piece of code that describes to the computer the way that it should do something. When we later call that method, the computer runs through the code, doing exactly what we wrote.

We can describe a simple method to the computer like this:

```
void SayHello()  
{  
  
}
```

But a method is pointless if it does nothing – so let's add some code inside the curly braces that can do something.

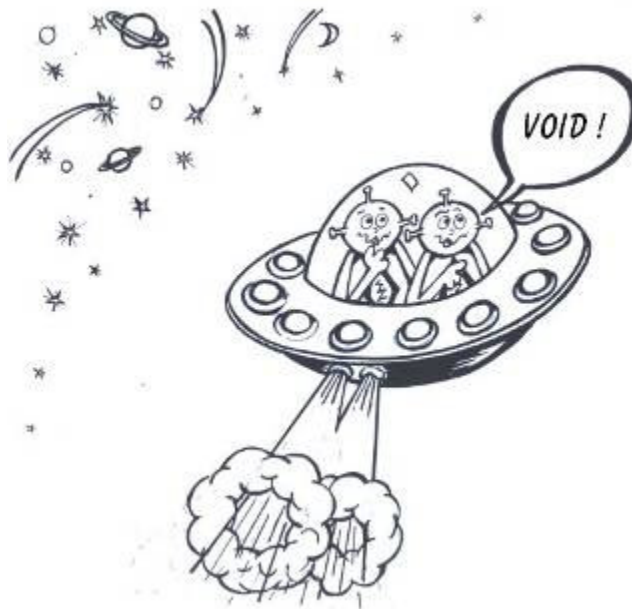
```
void SayHello()  
{  
    Console.WriteLine("Hello");  
}
```

(For the sake of neatness, you should indent code in a method one tab space to the right by pressing the TAB key)

If we were to call this method now, it would write the word “Hello” to the screen. Not a very difficult problem – so the method is quite simple.

What does VOID mean?

You may feel confused by this strange word “void” in the example above, and wonder why we write it there at all. So let’s look at this word briefly. Just think of your favorite space movie – picture a vast, empty space somewhere out there. Void means “nothing”.



In a method, putting the word “void” before the method name means that once the method is finished running, it will send back a “nothing” value – in other words, nothing! It doesn’t mean the method does nothing, but rather that once it’s finished doing its thing it sends nothing back. This suits us fine in our SayHello method above because, after it’s written the word “hello”, it doesn’t need to send anything back. That’s all we needed it to do.

This will make more sense when we discuss some cases where we might want to send something back from a method, which we’ll do a little later on in the journey.

Calling a Method

By writing down the code above, we have taught the computer HOW to write hello to the screen, but we have not yet told it to actually do this.

It’s like your teacher showing you how to add numbers, and getting you to write the method down, but not actually asking you to do any particular sum.

To get the computer to “do it”, we must call the method. This is sometimes done by simply writing the name of the method, along with the parenthesis, like this:

```
SayHello();
```

When the computer hits this line, it does not know by itself how to handle such a command – so it goes looking frantically for a method that you’ve written called “SayHello”. If it finds one, it quickly runs through each line in that method, doing whatever you wrote there.

Then it comes back, dripping with pride, “Look what I did”, and takes all the credit.

Building Block : Declaring and Calling a Method

A class most often has one or more methods. These do something – they perform some action. They’re called methods because it is here that we describe to the computer the method of doing something: the step-by-step instructions explaining how to do something.

```
class Person
{
    // Fields

    public string firstName;
    public string lastName;

    // A method

    public void ShowFullName()
    {
        Console.WriteLine("Name is " + this.firstName + " " +
this.lastName);
    }

    ...
}
```

A plus sign “+” is a way to add strings onto the end of each other, and recall that we also need quotes “” around text characters but not variables. The two forward slashes “//” are a way to tell the computer to ignore everything on this line. We use these slashes to insert human-readable comments and notes into our code. The word “**this**” refers to “**this object I’m talking about**”. So “this.firstName” refers to the firstName field of the current object (which is Jiminy in the example below). A method is set into action by writing the method name, including the parenthesis, somewhere in your program.

```
Person Jiminy;
Jiminy = new Person();
Jiminy.firstName = "Jiminy";
Jiminy.lastName= "Cricket";
```

```
Jiminy.ShowFullName();
```

When the ShowFullName line is encountered, the computer goes back and finds the method of the specified name and runs through the code written in it, step by step.

More About Methods

In the previous chapter we introduced methods, but there is a lot more to them than what we covered in the last chapter. This chapter covers some of the more advanced aspects of methods. At first, you'll be able to get by with just the basics, but at some stage you'll need to understand things like sending values into a method and getting values out of a method.

Parameters

Let's imagine you get the urge to have the computer write the following to the screen:

```
Hello Jo  
Hello Sam  
Hello You
```

Here's one way to do it ... we could write a separate method for every situation:

```
void WriteHelloJo()  
{  
    Console.WriteLine("Hello Jo");  
}  
  
void WriteHelloSam()  
{  
    Console.WriteLine("Hello Sam");  
}  
  
void WriteHelloYou()  
{  
    Console.WriteLine("Hello You");  
}
```

and then call the methods like this :

```
WriteHelloJo();  
WriteHelloSam();  
WriteHelloYou();
```

But that seems like rather a waste of energy since the three methods are all almost the same. What if we could use one WriteHello method and, each time we call it, just tell it the part that needs to be different.

This can be done by writing the method as follows :

```
void WriteHello( string someName )  
{  
    Console.WriteLine( "Hello " + someName );  
}
```

and then calling it like this :

```
WriteHello( "Jo" );  
WriteHello( "Sam" );  
WriteHello( "You" );
```

As you can see this saves both space and effort. You should always try to write as little code as possible. Generally speaking, the shorter the program is, the smarter is the programmer.



When we write the method in this smart new way,

```
void WriteHello( string someName )  
{  
    Console.WriteLine( "Hello " + someName );  
}
```

we're really saying "Whenever I call this method, I'll pass it a string of letters holding some name. Whatever that *thing* is that I pass you, I want you to write after the word "Hello".

The thing in the parenthesis (`string someName`) is called a **parameter**. It allows you to pass a value into the method when you call it.

After all, when your teacher taught you how to add numbers, she didn't teach you every single possibility - she taught you the method and then threw a whole bunch of different problems at you: "Add 2 and 5. Now add 7 and 3." It's as if she gave you a method called `AddNumbers` and passed you different parameters each time. No matter what parameters she sent you, you were able to work out the answer because you knew the method.

The computer could care less what name you give to a parameter – but is very fussy about your using the same name throughout your method. This, for example will work correctly:

```
void WriteHello( string x )
{
    Console.WriteLine( "Hello " + x );
}
```

But this will not :

```
void WriteHello( string someName )
{
    Console.WriteLine( "Hello " + someBodysName );
}
```

Can you see the mistake? "someName" and "someBodysName" are different - this will confuse our binary friend and cause it to throw a tantrum.

It is also perfectly legal to have *more than one parameter* in a method, you just need to separate them with commas:

```
void WriteHello( string firstName, string lastName )
{
    Console.WriteLine( "Hello " + firstName + " " + lastName );
}
```

But then you must pass in the right number of values when you call the method

```
WriteHello( "Jiminy", "Cricket" );
```

This will, of course, write out to the screen "Hello Jiminy Cricket".

Sending Wrong Parameter Types

Suppose that, back when you first learned how to add numbers, your teacher suddenly threw this problem at you: "Add the numbers 5 and *flower*".

How would you have responded? Probably something like this: "Flower is not a number! You can't add those." Quite right.

In the same way, the computer will complain bitterly if you send it a value of the *wrong type*. This is a common programming error. So if things aren't working as you'd expected, go back and check that you're sending values whose types match those defined in the method.

Building Block : Parameters

If we want to pass specific values into a method, we define parameters in the method :

```
class Person
{
    // Fields

    string firstName;
    string lastName;

    // Method

    public void LuckyNumber( int numberOfTeeth, int age )
    {
        Console.WriteLine( "Lucky number is " + numberOfTeeth * age );
    }

    ...
}
```

And then, whenever we call the method, we make sure we pass the right kind of values into those parameters. In the example below we pass in two integer numbers because the parameters for the method “LuckyNumber” were defined as integers.

```
Person jiminy;
jiminy = new Person();
jiminy.LuckyNumber( 24, 14 );
```

Methods that Send Values Back

The methods we’ve shown thus far have written something to the screen and then simply given control back to the line that called them – “I’ve done my bit. Back to you Jack”. Sometimes, however, we want the method to send some useful value back to the line of code that called it.

Here’s an example. We’ll write a method that looks up how many legs a specified animal has and then sends that number back to whatever called the method.

Here’s how we could write the method – remember that this is the part where we teach the computer HOW to do something - we’re writing the method down. I’ll first write what I want the computer to do in English, and then I’ll write it in C#:

- If the animal that we’re talking about is named “elephant”, then the Number of Legs = 4.
- Otherwise, if the animal that we’re talking about is named “turkey”, then the Number of Legs = 2.
- Otherwise, if the animal that we’re talking about is named “oyster”, then the Number of Legs = 1.
- Otherwise, for animals named anything else, then the Number of Legs = 0.

```

int NumberOfLegs( string animalName )
{
    if ( animalName == "elephant" ) // If the name of the animal equals "elephant"
    {
        // Send back the value 4
        return 4;
    }
    else if ( animalName == "turkey" )    // Otherwise, if it equals "turkey"
    {
        // Send back the value 2
        return 2;
    }
    else if ( animalName == "oyster" )    // Otherwise, if it equals "oyster"
    {
        // Send back the value 1
        return 1;
    }
    else    // Otherwise (under all other conditions)
    {
        // Send back the value 0
        return 0;
    }
}

```

And then we could *call* the method. Let's call it twice:

```

1.    int i; //Create a variable "i" of integer type to hold the Number Of Legs value
2.    i = NumberOfLegs("turkey");    //Now i = 2 from the NumberOfLegs method above
3.    Console.WriteLine("The turkey has " + i + " legs");
4.    i = NumberOfLegs("monkey");    //Now i = 0 from the NumberOfLegs method above
5.    Console.WriteLine("The ape has " + i + " legs");

```

Line 3 will write out "The turkey has 2 legs" and line 5 will write "The ape has 0 legs". So the method passes back a value, and that returned value can be plugged directly into the line of code that called the method.

Why did we define the method like this?

```

int NumberOfLegs( string animalName )
{
    ...
}

```

instead of like this :

```

void NumberOfLegs( string animalName )
{
    ...
}

```

or like this :

```

string NumberOfLegs( string animalName )
{
    ...
}

```

It is, of course, because we wanted the method to send back a whole **number** in this case; not a “nothing” value (void), not a string of letters but a whole number. And one data type for working with whole numbers is the “integer”, or “int” for short.

When we write a method, we always specify what kind of data that method will return. If we don’t need it to return anything, we specify that it will return “void” - nothing.

```

void JustWriteSomething( string something )
{
    Console.WriteLine( something );
}

```

Lastly, you will probably have figured out that the word “**return**” is the part that actually sends back a value. As soon as the computer encounters the “return” word it jumps out of the method and sends back whatever it was asked to send back.

Building Block : Return Values

Sometimes we want to pass a value back from a method. In such a case, instead of specifying the type “void”, which means “we’ll be sending nothing back”, we specify the particular data type that the method will return.

```

class Person
{
    // Fields

    string firstName;
    string lastName;

    // Method

    int LuckyNumber( int numberOfTeeth, int age )
    {
        return ( numberOfTeeth * age );
    }
}

```

The value returned is then automatically available wherever we called the method.

We could, for example, use the returned value in either one of the following ways:

We could first store the answer in a variable and then make use of the variable value in a separate statement

```
Person Susy;  
Susy = new Person();  
int num = Susy.LuckyNumber(24, 14);  
Console.WriteLine("Susy's lucky number is " + num);
```

Or we could call the method directly in our WriteLine statement like this :

```
Console.WriteLine("Susy's lucky number is " + Susy.LuckyNumber(24, 14) );
```

Public, Private and Protected Methods

Just as with fields (see the previous chapter), methods can be called from other classes. We may want to either allow this or prevent it.

By default, all methods will be treated as "private", meaning that they can only be used inside their own class. If you wish to open this up so that other classes can call them, you can add a word such as "public" at the beginning of the declaration.

```
public void JustWriteSomething( string something )  
{  
    Console.WriteLine( something );  
}
```



Some areas in the real world are "out of bounds" unless you're specially authorized. Like in restaurants, only the chefs, waiters and waitresses can go into the kitchen which is a "private" area. Whereas, the dining room is a "public" area – everyone can walk around in there. In a similar way, some code is out of bounds to other classes.

In an earlier section, we used an example with private and public *fields*. Let's add to that example and include some private and public *methods* in the "Animal" class and then try to access them from the "Zoo" class.

```
class Animal
{
    public string kindOfAnimal;
    public string name;
    public int numberOfLegs;
    public int height;
    public int length;
    public string color;
    bool hasTail;
    protected bool isMammal;
    private bool spellingCorrect;

    public string GetFoodInfo()
    // Public method that retrieves information about what food the animal eats
    {
        // Imagine we had some code here that searched a database
        ...
    }

    private void CheckSpelling()
    // Private method to check whether the animal's type was spelled correctly
    {
        // Imagine we had some spell checking code here
        ...
    }

    protected bool IsValidAnimalType()
    // Protected method that decides whether the animal type is a valid one
    {
        // Imagine we had some code here to test valid animal types
        ...
    }
    ...
}

class Zoo
{
    Animal a = new Animal();
    a.name = "Kangaroo";

    string food;
    bool animalExists;

    // The following will succeed because the "Zoo" class is allowed
    // to address public methods in the "Animal" class

    food = a.GetFoodInfo();           // Call a public method

    // The following lines will both FAIL because the "Zoo" class
    // is not allowed to address private or protected methods
```

```

        a.CheckSpelling(); // Try to call a private method
        animalExists = a.IsValidAnimalType(); // Try to call a protected method
    }

```

Constructor Methods

You will very often see classes written to include a rather special kind of method called a "**constructor**". The *name of a constructor method is the same as the class name itself* and there is *no return type* included.

It's called a constructor method because it helps to "construct" or build the object when it is first created. It is good practice to write a constructor method in your classes.

```

class Person
{
    // Fields

    string firstName;
    string lastName;

    // Constructor Method for the Person Class

    public Person()
    {
        firstName = "Johnny";
        lastName = "Rocket";
    }
}

```

A constructor method is treated specially - it gets **automatically** run whenever an instance of the class is created.

Reminder :

By “an instance of the class”, we mean a *specific* object of the class. For example, in the earlier section on “objects” we highlighted “Jeremy the Gorilla” as a specific object, or instance, of the class named Animal.

So if we executed the following code...

```

Person p = new Person();
Console.WriteLine( p.lastName );

```

we would find it would write out the word "Rocket". By creating a "Person" object we automatically and invisibly ran the "Person" constructor method, which sets the lastName to "Rocket" in this case.

Here's a similar example from the real world. When a “new instance of a kid” is born at a hospital, somebody automatically marches into the room and takes care of a few details to make sure this new instance is properly registered and legally ready to participate in the world – before it leaves the hospital.



This action is similar to a constructor method being run on a class – before the new class instance is allowed to do anything, the constructor method is run. In that constructor method, you include anything you feel needs to be done automatically before the object is to be considered “ready for the world”.

Constructors with Parameters

You may write a constructor method to include parameters. Here's an example of a class with two different constructor methods:

```
class Person
{
    // Fields

    string firstName;
    string lastName;

    // First constructor method

    public Person()
    {
        firstName = "Johnny";
        lastName = "Rocket";
    }
}
```

```
// Second constructor method

public Person( string f, string l )
{
    this.firstName = f;
    this.lastName = l;
}
}
```

And now there are now *two different ways we could construct the object*:

Either like this:

```
Person p = new Person();
```

And the p.lastName field will automatically get the value "Rocket".

Or like this:

```
Person p = new Person( "Jiminy", "Cricket" );
```

And the p.lastName field will get the value "Cricket".

Recall that the word "**this**" refers to "**this object we're creating**". So it is effectively saying "set this new object's firstname and lastname fields to whatever values get passed to the constructor method."

Events

In the real world, events occur all the time. There are some that we *cannot control*, such as the sunrise and sunset (I'd like to see you try and make the sun rise and set by yourself), and there are other events that we *can* choose to kick off, such as music coming out of a stereo's speakers.



Events in the computer's world are more like these:

- A button is clicked
- A timer runs out of time
- The mouse is moved
- A key on the keyboard is pressed

Obviously, if a button is clicked, we want the computer to do something - otherwise why bother having the button there in the first place. But the computer expects you to tell it that this click is important to you - and it also expects you to specify what you want it to do if the button does get clicked.

Let's take the button clicking example, since it's probably the most common event, and follow through how to deal with it. Let's assume you have, in your program, a button object named *mrButton*, with the words "Click me" written on it.

In fact, **how about trying this yourself** as you follow the discussion? To get started:

- Open Visual C# Express by clicking START -> All Programs -> Microsoft Visual C# 2005 Express Edition
- Start a new Windows application project: Select File -> New Project and then choose the project type "Windows application".
- Visual C# Express will put down a few files with some "skeleton code".
- In the "Solution Explorer" window on the right side (where all the files are listed), delete the file named Form1.cs.
- Double-click the file named Program.cs and delete all the skeleton code that was automatically inserted.
- To create a program with an instance of a button, type the following code into Program.cs exactly as it is here (except that you may ignore the fact that some words are written in *italics or bold*)

```
using System;
using System.Windows.Forms;

class MyButtonClass : Form
{
    private Button mrButton;

    // Constructor method

    public MyButtonClass()
    {
        mrButton = new Button();
        mrButton.Text = "Click me";
        this.Controls.Add(mrButton);
    }

    // Main method

    static void Main()
```

```
{
    Application.Run( new MyButtonClass() );
}
```

- Run the program by pressing the F5 key on the keyboard (or by clicking the green Run button). If you encounter any error messages, check carefully for typing errors. If the program runs correctly, you should see a form displayed with a button named “Click me”. At this stage, the button will not do anything if you click it. Quite a let-down, I know, but we’re getting there.

The Button Click Event

Specifying What To Do When an Event Occurs

Once we know what it is we want to do when the button is clicked, we must specify the **method** by which it can be done. These types of methods that tell a computer what to do when an event happens are called event handler methods because they “handle” an event. Here's an example – the bold code below - all it actually does is change the writing on the button - so it's a short method.

```
using System;
using System.Windows.Forms;

class MyButtonClass : Form
{
    private Button mrButton;

    // Constructor method

    public MyButtonClass()
    {
        mrButton = new Button();
        mrButton.Text = "Click me";
        this.Controls.Add(mrButton);
    }

    // Main method

    static void Main()
    {
        Application.Run( new MyButtonClass() );
    }

    // Event handler method

    void MyButtonClickEventHandler( object sender, EventArgs e )
    {
        mrButton.Text = "You clicked me!";
    }
}
```

Do you still have your program running?

- Stop it (by clicking the X in the top right of the window that the form is displayed in)
- Add the bold code above to your program and then press F5 to run the updated program.
- Try clicking the “Click me” button now. Arrgh! Still does nothing eh?

If you've read the chapter on methods, you'll recognise the basic structure of the method above. The word "void" means that the method will return "nothing" once it's finished. The word "MyButtonClickEventHandler" is the name that we chose to give to this method.

But then it starts to get a bit weird. You probably realize that there are two parameters inside the parenthesis, but they're odd : (object *sender*, EventArgs *e*). I'm sorry to be the one to break this bad news to you, but with event-handling methods, we *cannot make up our own parameter types*. When this method gets called, the system itself *automatically passes certain parameters* to it - there's not a damned thing we can do about it.

So we just have to play along and make sure we always write our event handler with the parameter types that are expected. In the case of a "click" event, and many other events, the parameters passed are of type "object" and "EventArgs". In the example above we chose parameter names of "sender" and "e", but we could have chosen any *names* - it's the parameter *types* that the computer is fussy about. The following, for example would work precisely the same as the example above. You can prove this to yourself, if you wish, by changing the parameter names in your program to “x” and “y”.

```
void MyButtonClickEventHandler( object x, EventArgs y )
{
    mrButton.Text = "You clicked me!";
}
```

The first parameter usually holds some information about the thing that caused the event. The second parameter relates to information about the event itself.

It's also worth knowing that, although the system always passes some values in these two parameters, you will often have no need to use them - they are sent to the event handler method "just in case".

Hooking the Event Handler Method Up to the Event

Wondering why the button's not doing anything when you click it? The method above gets called only if we *link the button's "click" event to it*. Somewhere in the program we have to say "when this button is clicked, go to that event handler".

This is obvious really - if you had several buttons and several event handlers, how would the computer know which method to run for each button click?

The code for linking an object's event to an event handler method is a little strange too. Again, we show it in bold below.

```

using System;
using System.Windows.Forms;

class MyButtonClass : Form
{
    private Button mrButton;

    // Constructor method

    public MyButtonClass()
    {
        mrButton = new Button();
        mrButton.Text = "Click me";
        mrButton.Click += new System.EventHandler(MyButtonClickEventHandler);
        this.Controls.Add(mrButton);
    }

    // Main method

    static void Main()
    {
        Application.Run( new MyButtonClass() );
    }

    // Event handler method

    void MyButtonClickEventHandler( object sender, EventArgs e )
    {
        mrButton.Text = "You clicked me!";
    }
}

```

You could translate this from the computer's world to our own like this:

To the button's "click" event, add a new event handler method - the one I wrote down called *MyButtonClickEventHandler*.

So when you click the button, the line of code above ensures that the program gets directed to your event handler method. When that method runs, the writing on the button will change to "You clicked me!"

To experience this, stop your program, add the bold code above (make sure it's in the right place) and then press F5 to run it again. Click the button and the text on it will change. See Program 5 in the examples folder of this book to see a working sample of the button click event.

Mouse Events

This is the basic structure of an event handler method for handling mouse events. In this case, it is probable that you *will* want to use the information passed in the *MouseEventArgs* parameter, to find out, for example, which one of the mouse buttons is being held down.

```
public void TheMouseIsDown( object sender, MouseEventArgs e )
{
    if ( e.Button == MouseButtons.Left )
    {
        this.Text = "Left mouse button just went down";
    }
}
```

And this is how you would hook up an event to it. This example is saying "if the mouse button goes down while this program is running, then go to the method called *TheMouseIsDown*", which is a method type that knows how to handle mouse events :

```
this.MouseDown += new MouseEventHandler( TheMouseIsDown );
```



You could get a little fancier. When the following handler is triggered, the code we've written within it will make the whole window wider or narrower, depending on which button is clicked.

```
public void TheMouseWasClicked(object sender, MouseEventArgs e)
{
    // If left button is clicked
    if (e.Button == MouseButtons.Left)
    {
        // Make the current window wider
        this.Width = this.Width + 100;
    }
    else if (e.Button == MouseButtons.Right)
    {
        // Make the current window narrower
        this.Width = this.Width - 100;
    }
}
```

And this mouse event handler to detect any movement of the mouse and draw a circle wherever it currently is:

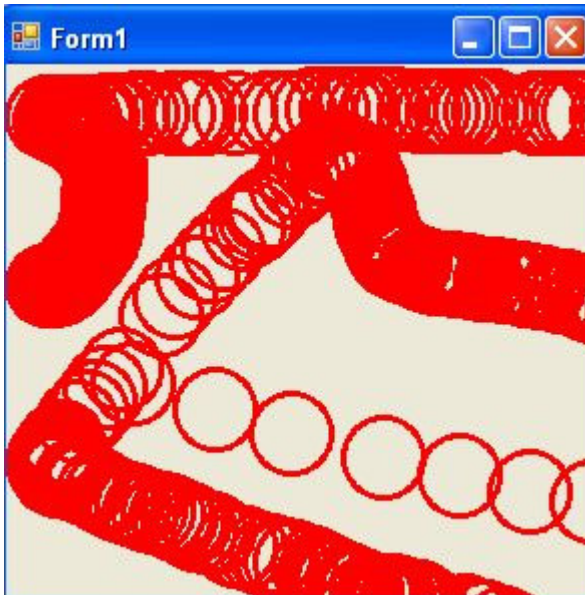
```
public void TheMouseMoved(object sender, MouseEventArgs e)
{
    // Prepare drawing surface
    System.Drawing.Graphics g = this.CreateGraphics();

    // Use a red pen
    System.Drawing.Pen redPen = new System.Drawing.Pen(Color.Red, 3);

    // Draw a circle (an ellipse with equal width & height)
    // at the current X and Y co-ordinates of the mouse
    g.DrawEllipse(redPen, e.X, e.Y, 40, 40);
}
```

```
// Clean up  
g.Dispose();  
}
```

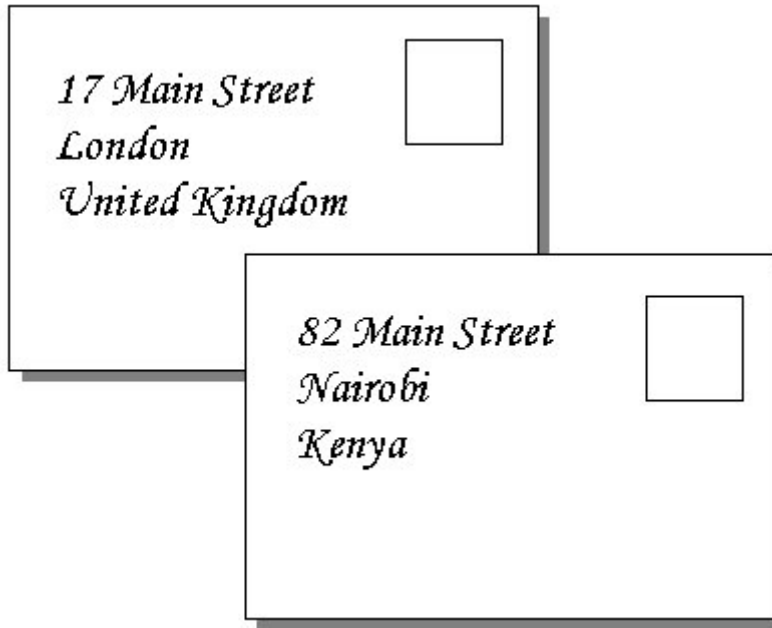
This screen-grab shows how that looks as the mouse is moved around:



Name Spaces

Name Spaces and the Postal Service

There must be hundreds, if not thousands, of streets in the world called "Main Street". How does a letter sent to someone in "Main Street" find it's way to the correct place? The answer of course is that there is more to the address. We generally add at least a city name and a country name.



Obviously if we write:

- 17 Main Street
- 82 Main Street

then we're not showing how the streets differ and the postman would be horribly disturbed.

Instead, we could write them:

- 17 Main Street. London
- 82 Main Street. Nairobi.

That's better, but what if there happen to be two cities called London! Then we'd better go a bit further and add the country name too.

- 17 Main Street. London. United Kingdom.
- 82 Main Street. Nairobi. Kenya.

It would work just as well to write them the other way too:

- United Kingdom. London. Main Street. 17.
- Kenya. Nairobi. Main Street. 82

Now the letter should get to it's destination okay. In the 2nd case, for example, fly it to Kenya. From the airport, send it to the city of Nairobi. Once there, get a postman to walk down Main Street and find no. 82.

The format [Kenya.Nairobi.MainStreet] can be considered a kind of "name space" for posting letters.

Name Spaces and Programming Code

So what's this got to do with programming?

Suppose that Microsoft wrote a class called "Point", to be used for drawing a shape at a particular *point* but you also wrote a class called "Point" - let's say it loads a picture of a person *pointing*.

Clearly these two classes do entirely different things but it makes sense to call them both "Point". How will programs know which one to use?

It would certainly make sense to make their names differ somehow. The .NET way of doing this is to use different name spaces. For example:

- Microsoft.Drawing.*Point*
- Susy.PictureStuff.*Point*

The class names are the same ("Point"), but the "name space" before the class name makes it quite clear which one is which. Now, if I wanted to use Susy's Point class, I could create a Point object using the constructor method like this:

```
Susy.PictureStuff.Point SusyPoint = new Susy.PictureStuff.Point();
```

And to use Microsoft's Point class, I would write :

```
Microsoft.Drawing.Point MicrosoftPoint = new Microsoft.Drawing.Point();
```

Although both of my objects are of class type "Point", they will behave quite differently.

How to Give Your Own Class a Namespace

Putting your own class into a name space is dead easy. Instead of simply writing your class like this :

```
class Animal
{
    ...
}
```

wrap it inside something like this :

```
namespace Susy
{
    class Animal
    {
        ...
    }
}
```

or like this if you wish :

```
namespace Susy.FunStuff
{
    class Animal
    {
        ...
    }
}
```

You can make up any name space that you feel makes sense. Simply separate your words with a period ".".

In the last example above, the class *Animal* is now in the name space *Susy.FunStuff*.

You will also find that all Microsoft's code libraries, as well as those from other organizations, are placed in name spaces. So to use their code, it's necessary to understand how to work with name spaces.

Building Block : Name Spaces

To distinguish classes from each other and to group similar classes together logically, it is good practice to always put a class inside a sensible **name space**.

```
namespace Charles.SchoolUtilities
{
    class Animal
    {
    }
}
```

An *instance or object* of the above class could be created using the class constructor method as follows :

```
Charles.SchoolUtilities.Animal cat = new Charles.SchoolUtilities.Animal();
```

Or if we were going to use many objects in our program from the same namespace, and we didn't want to have to write out "Charles.SchoolUtilities" each time, we could use the "using" keyword at the top of our program to tell the computer that we will be using this namespace in our program from now on. For example :

```
// tell the computer that we'll be using classes from this namespace
using Charles.SchoolUtilities;

// create an Animal object from the Charles.SchoolUtilities name space
Animal cat = new Animal();
```

Inheritance

As this book is about getting started programming, we don't want to confuse you with too much complicated stuff, so we won't say too much about inheritance.

However, since you may sometimes see classes declared like this:

```
class MyFancyClass : Form
{
    ...
}
```

instead of just like this

```
class MyFancyClass
{
    ...
}
```

we should explain what the part from the semi-colon ":" onwards is about.

People Inheritance

It is common for people to inherit characteristics from their parents. You may have your Mother's hair color or your Father's nose.



This doesn't mean you are identical in all respects to your Mother or Father, but there are certain characteristics that come "built in" when you're born. But you also have various unique characteristics and abilities all your own beyond those of your parents.

Code Inheritance

When we write programming code, it could be extremely useful to have access to something like that - to inherit a whole bunch of abilities from an existing class, either your own or someone else's.

Here is an example. We have defined two classes “Animal” and “Bird”, but the “Bird” class inherits from the Animal class.

```
class Animal
{
    public string kindOfAnimal;
    public string name;
    public int numberOfLegs;
    public int height;
    public int length;
    public string color;
    bool hasTail;
    protected bool isMammal;
    private bool spellingCorrect;
    ...
}

class Bird : Animal    // “Bird” class inherits from “Animal” class
{
    public string featherColor;
    ...
}
```

In the real world, a *bird is a kind of animal*, but it has some characteristics that don’t apply to all animals. So it makes sense for a *Bird* class to have all the characteristics of an *Animal* as well as some extra ones. In this case, we’ve identified one special field for birds only – featherColor.

So when we write

```
class Bird : Animal
{
    ...
}
```

we’re really saying “I’m defining a new class called ‘Bird’ but it must automatically inherit everything from the “Animal” class too. We often say that the Bird class has been **derived from** the Animal class.

When we create an instance of a “Bird”, we can immediately address the fields of both an Animal and a Bird (as long as they’re not private):

```
Bird b = new Bird();
b.kindOfAnimal = "Tawny Eagle";
b.isMammal = false;
b.featherColor = "Tawny";
```

For clarity, we left out the *methods* in the Animal and Bird classes above, but the same applies to methods. A derived class can call any methods in the parent class, as long as they were not declared private.

Another Example - Inheriting some Windowing Abilities for your Program

Suppose that you want to write a program that runs within a typical "Window". It must be resizable, maximizeable, minimizeable, draggable, etc. It makes sense to let your own class "inherit" abilities from a class that can already work with this kind of interface. The class `System.Windows.Forms.Form` is the usual choice.

So when we write

```
class MyFancyClass : Form
{
    ...
}
```

we're really saying "I'm writing my own class but it must automatically inherit all the power of the 'Form' class too".

The discussion above is made more practical in Part III - the chapter on `System.Windows.Forms`.

When to Use Inheritance

You don't have to inherit from a class in order to use it! Inheritance is best used in cases where what you're trying to achieve can mostly be done by an existing class and you just want to extend it or customize it.

Building Block : Inheritance

A class can be made to inherit the characteristics and abilities of another class.

In the following example, the class "Guitarist" inherits three fields from the class "Musician" and adds two fields of its own.

The colon ":" is the part that tells the computer to make the new class (Guitarist) inherit from the class written to the right of the colon.

```
public class Musician
{
    public string name;
    public int ageInYears;
    public bool canSing;
}

public class Guitarist : Musician
{
    public string guitarType;
    public string guitarBrand;
}
```

```
Guitarist g = new Guitarist();
g.name = "Jimbo van Helsdingen";
g.ageInYears = 28;
g.canSing = false;
g.guitarType = "Acoustic";
g.guitarBrand = "Gibson";
```

About the Book

Microsoft's newest programming language, C#, (pronounced "c-sharp") is both powerful and easy to use. It presents a great opportunity for the new generation of developers to start out with a language that is highly respected in the modern workplace.

This text introduces object-oriented programming to the young developer (core target age is 12-16) in a lightweight fashion, allowing them to get started with real programs in a Windows environment.

About the Author



Martin Dreyer is an ex-high school teacher who now heads a team of software developers in South Africa.

His formal qualifications are a Higher Diploma in Education : Physical Science and a Bachelor of Science Degree : Computer Science and Information Systems.