An Evaluation of .NET and Java

Paul McIntyre

Submitted in partial fulfilment of the requirements of
Napier University for the degree of
Master of Science in Information Technology (Software Engineering)

School of Computing
January 2003

Authorship declaration

I, Paul McIntyre, confirm that this dissertation and the work presented in it are my own achievement.

1. Where I have consulted the published work of others this is always clearly attributed;
2. Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;
3. I have acknowledged all main sources of help;
4. If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;
5. I have read and understand the penalties associated with plagiarism.

Signed: Date: 17th January 2002

Matriculation no: 00400017

Abstract

The aim of this document is to examine the Microsoft .NET Framework compared it with Sun's Java.

The latest version of Microsoft's programming suite, Visual Studio, incorporates a new foundation known as the .NET framework. It is designed partly as a way of removing the device dependence of the programs that use it (as with Java). The differences in specification from Java mean that systems are going to have different strengths and weaknesses. This document examines the difference between the structural features of both systems and evaluates the effect of these differences on the efficiency of the produced applications. The effect on the efficiency of a development team using the system caused by the differences in some of the important libraries is also examined. As part of this, the common language infrastructure of .NET is also examined and the (potential) use of multiple languages in .NET and Java investigated. An objective of this report is to draw conclusions as to the strengths of each platform, and these are drawn together with those gained as an example application is produced in each system.

The sample application was designed to contain as many of the important features as possible and to reflect a real-world enterprise application. This was then created in both .NET and Java to examine the usability of each. The criteria used to evaluate the development process for these applications were decided and conclusions drawn from them in addition to the overall experiences with each system. The final conclusions are then summarised.

It was found that .NET is the step forward from Java. The basic system is more comprehensive and easier to use. Generally, .Net is easier to use, largely as Java has become fragmented, with all the add-ons. The report defines a number of attributes for the frameworks, such as reusability, compatibility, portability, and so on. The results of this appraisal, is that, in most cases, .NET provides a better system than Java. Java, though, scores well in its costs and compatibility, while .NET does well in team working, documentation, speed of code, and reusability.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank Colin Hastie for his guidance.

Many thanks to Bill Buchanan and Sally Smith for stepping in at a late stage.

I would also like to thank the authors of the Java and .NET software development kit documentation which was consulted frequently during the production of this thesis. Where it was useful to explain a type by looking at its members, the tables of type members in the SDK documentation were used to create the tables, to which the explanations were then added.

1 Introduction

1.1 Introduction

The latest version of Visual Studio is built around a new programming system called .NET. This is the next phase of Microsoft's overall strategy (Weiss, 2001). It is designed partly as a replacement for the old Windows API, and partly as a competitor for Java. There are many ways that .NET and Java are similar. This is largely due to the basic design concept that they share: to replace the old style of programming API's with a completely object-oriented component-based one, as described by Mingins & Nicoloudis (2001). The .NET Framework is, in a similar way to Java, a set of programming libraries and a set of runtime virtual machines for different platforms to execute the compiled code. Both systems seek to make applications device independent through the use of device specific runtimes that interpret and execute code written in an intermediate language. Java places greater emphasis on the device independence, however .NET is still new so runtimes for other systems and devices will most likely be available soon. While there are great similarities in .NET and Java on this abstract level, each is implemented in a different manner. These differences will impact the potential uses to the developers using them. Where .NET deviates from the device independence strategy is in distributed applications. It is a stated aim of .NET to create a new generation of applications which communicate across the Internet to provide greater functionality than they would otherwise be able. Microsoft is expected to try to ensure that the servers for these applications must run on Microsoft products (although open source development teams are already working on their own .NET compilers, tools and servers). This document will attempt to ascertain whether .NET is an improvement over Java for standard use and identify any areas in which Java remains superior.

Both .NET and Java are complete development environments, they are not just for standalone applications. Both systems also allow dynamic web-sites to be created, for example. The different environments that each support are examined in Section 2. The most obvious measure of performance that can be applied to Java and .NET is the efficiency of the compiled applications. For example, the details of the garbage collection system will impact the memory usage of the final application. Section 3 discusses the features within each system that can affect the efficiency of the applications. When evaluating development systems, the efficiency of the development process itself also has to be examined. The simplest features can have a large effect on the speed of development if they are overly complicated or error prone. Section 4 looks at those features of the programming libraries that make up .NET and Java that can affect the efficiency of the development process. The programming libraries are not the only things that can affect the ease and speed of developing applications. Section 5 discusses some of these other factors, such as the support in each system for documenting types. The ability of .NET to allow components and code written in different languages to interact is discussed in Section 6.

Section 7 of this document describes the testing procedure used to evaluate the systems. This includes the description of the application that was written to demonstrate the use of each system. This section also describes the problems that arose during this process and any changes that were made due to this. Conclusions are drawn in this section based upon the ease of use of each system and whether the goals of the application could be completed. Finally, Section 8 summarises the conclusions.

2 Execution environments

2.1 Introduction

For the purposes of this section, the term “execution environment” is used to mean a distinct manner and location in which code is executed. For example, the ASP.NET page execution environment in which code is executed on a web server as part of a dynamic web page access. The first part of this section looks at the different execution environments in .NET and Java and draws comparisons between the nearest equivalents. The second part of this section looks at standard (client or standalone) applications, and in particular the networking support for them. The final part of this section looks at the support for applications based on the Internet, focusing specifically on ASP.NET.

2.2 Comparison of execution environments in each system

Table 2.1.1 defines the execution environments present in .NET and Java. In addition to those listed, both .NET and Java also target other execution environments (for example, portable computers). These are not discussed in this document as they are, as yet, rarely used.

The standard application is the most commonly used environment and is used for client and server programs as well as standalone programs. Java applications need to be loaded in to the runtime on the command line, where .NET applications are executables by default. With the greater availability of Java runtimes, Java applications are still more device independent though (although this may change). Windows Forms and the current Java equivalent, Swing, are conceptually very similar. They are both collections of user interface components that can be combined to create graphical user interfaces. Little more needs to be said as a comparison of the individual components is largely irrelevant. There is little difference between applets and applications in Java (except for security considerations), as they are essentially embedded applications. There is no equivalent of this in .NET, as web forms are better described as specialised user interface components for ASP.NET pages. Web forms are however the closest equivalent to applets in .NET. These components are different versions of the windows forms components, as they have to take into account that they are situated on the server (applets are downloaded and then executed on the client). Both ASP.NET and JSP are used to create dynamic web sites. They do vary in execution and are discussed later in this section, but they fundamentally fulfil the same purpose.

2.3 Applications

As stated in Section 2.1, there are few differences between applications in both systems. One of the major ones is the structure of the compiled files. The format of the files (e.g., the intermediate language) is discussed in

Section 6 of this document. The output files in Java (in either .class or .JAR form) are not immediately executable, and must instead be loaded into the runtime as a parameter on the command line. This gives Java programs an inherent disadvantage when distributed, in that users have to know how to run Java applications. Also, Java does not have separate output file types for applications (i.e., executables) and class libraries (i.e., DLL files), which can cause further confusion when running them.

.NET Execution environments	Equivalent Java execution environments
.NET Applications The application is the basic (traditional) type of program. For example, a client program. Applications run on a standard system which may or may not be connected via a network or the Internet to other computers (and hence other applications, including other instances of itself on remote machines).	Applications There is little difference between .NET's and Java's applications in basic terms. The main differences come in structure and the class libraries, which are discussed in the following chapters.
Windows Forms These are components used to provide user interfaces to applications.	Advanced Windowing Toolkit (AWT) / Swing AWT was the original system for GUIs in Java. Swing replaced it. Both are similar in concept to Windows Forms.
ASP.NET pages These are dynamic web pages. When they are accessed, the associated code executes to initialise the page. The code and the design of the page can be separated into two separate files.	Java Server Pages (JSP) These are a relatively new addition to the Java system. They are more complex than ASP.NET, with the code being mixed in with the design of the page. They are created using servlets.
ASP.NET Web Forms These are a set of components to put graphical user interfaces on to an ASP.NET page. In concept they are similar to Windows forms, but with some changes to account for them being executed over the Internet.	Applets These differ considerably from web forms. Where web forms use a unique set of components and exist on a web page, Applets are the same as Java applications. They do not exist on the web page, but are loaded and executed by a runtime which is called by the web browser when it reads the instructions in the page.
ASP.NET Web Services These provide data in XML format when called over the Internet.	Servlets

ADO.NET (Active Data Objects)	JDBC (Java DataBase Connectivity)
This is the system used for accessing data sources. This is discussed elsewhere as it does not itself execute code, but acts as an adapter to the data source (which does).	JDBC is similar in concept and execution to ADO.NET.

Table 2.1.1: Execution environments supported by .NET and Java.

2.3.1 Applications and networking

Both systems are designed to allow applications to utilise networks (including the Internet). Networked applications usually fit into one of three categories: client-server, peer-to-peer, and standalone. Standalone applications are the simplest in that they are the only component, and use the network as a tool (for example, to download a file). Peer-to-peer applications are where multiple instances of the application communicate with one another to perform a common task (the most notorious examples being some file sharing applications). The most common type of networked application is the client-server application, where there are multiple instances of the client part of the application that connect to each instance of the server part of the application. This means that both systems need to provide types which allow an application to: query a network, make a connection to an application across a network and accept a connection from an application. The traditional method of network communication is sockets, and these are used by both Java and .NET. Sockets themselves are not discussed in any detail here as they are similar in concept and .NET aims to hide them for the most part, instead using higher-level classes that use sockets themselves.

As the name implies, networking in .NET is one of the core features. The System.Net namespace provides the classes that are used to implement the functionality. The aim of the networking types in .NET is to abstract away the actual implementations and provide a common way of using all the protocols. This means that (for client applications) sockets are a last resource, with two layers of abstraction placed above them. For the central functions of connecting to a server and accepting connections .NET uses a request/response model. The types that are usually used in .NET to connect to a remote resource and send and receive data are `System.Net.WebRequest` and `System.Net.WebResponse`. These are described below. For more common usage, to upload and download data to and from a server as quickly and easily as possible, the `WebClient` type wraps these classes. A WebClient object can upload and download data in three forms: as a stream, as a byte array or as a file. These are described in Table 2.2.1.

To have slightly more control over requests, the wrapped types can be used directly. A client sends a WebRequest object and receives a WebResponse object from the server. The (static) WebRequest.Create method examines the URL passed as a parameter and creates an instance of the appropriate WebRequest sub-type to connect to the specified resource. Hence, if the

object is assigned to a variable of type WebRequest, the actual protocol is abstracted away. There is no need for the programmer (or the application) to know that it is an instance of the HttpWebRequest type that is used for HTTP accesses, to use the most common example. This object is then used by the application to get a response from the server. As all these connection types ultimately connect using sockets, to respond to a request the server needs to listen for socket connections. This is done with an instance of the Socket type (the same type is used for connecting and listening for connections). As with making a connection, .NET provides higher-level classes that use sockets to make this easier. The `TCPListener` class in the `System.Net.Sockets` namespace is one of these. This type listens for connection requests on the specified port. When a socket connects, the listener object can then pass it to the program. The AcceptSocket method pauses the thread until a connection attempt occurs and then returns the socket object. This can then be used to return the appropriate data.

Type of data	Methods	Description
Stream	OpenWrite OpenRead	The address of the remote resource is passed as a parameter. Each method returns a Stream object that is used to either output data to the remote resource or access the data from the resource.
Byte Array	UploadData DownloadData	DownloadData works in the same way as OpenWrite, except that it is a byte array that is returned. UploadData requires the byte array to be sent to be included as an additional parameter when calling the method, and returns a response from the resource (also a byte array).
File	UploadFile DownloadFile	The file name of the source, / destination file is passed as an additional parameter.

Table 2.2.1: Data types supported by the .NET type WebClient

Like .NET, Java puts its network classes in `java.net`. In Java, there is an abstract socket class (`SocketImpl`) that is otherwise similar to the socket type in .NET. Unlike .NET, this is split into two separate socket classes in usage. These classes are `Socket`, which is the client-side type, and `ServerSocket`, which is the server-side version. Java does not contain classes to abstract away the sockets in the same way as the request and response classes in .NET. The Socket class is given the address of the server to connect to in the constructor (in .NET it is given as a parameter when connecting) and tries to connect immediately. This results in the thread it is in being stopped until the server responds when it is initialised. This makes the .NET version of sockets less error prone. The range of higher-level classes in .NET make it easier to use than Java and it gives it an advantage for networked applications.

2.3.2 Graphical user interfaces (GUIs)

Both systems use a component-based architecture for GUIs. In Java, the Swing library took over from the original library, which means for full backward compatibility the original Java graphics library (AWT) is needed, although Swing is now taken to be the standard. The `System.Windows.Forms` namespace contains the controls in the .NET framework for applications. For customised controls in .NET there are three ways to create new ones: a composite control can be created containing two or more existing controls, an existing control can be extended to add extra functionality, and new controls can be created. When extending controls, the method that draws the control (`OnPaint`) can be overridden to customise the appearance of it. When creating new controls, the OnPaint method must be written from scratch to create the user interface for the control. This allows complete control over the new component. The creation of the user interface is done via GDI+ (Graphics Device Interface Plus), which is the name for .NET's drawing mechanism. This allows vector, bitmap and text manipulation. These graphics classes are found within the `System.Drawing` namespace. As the systems are similar (and not needed unless a custom interface is needed), there is no need to go into further depth in this document.

2.4 Web based applications

2.4.1 ASP.NET

Each ASP.NET application can contain multiple web pages and services. ASP.NET pages are, as the name implies, an extension of the Microsoft ASP system. While there are differences in implementation, the goal is the same: to create a dynamic web page that is created based upon data within the server. The other part of the .NET strategy for open Internet servers (i.e., excluding distributed applications) is web services. Web services can be accessed across the Internet by any application (both clients and ASP.NET applications) and their methods can be called to return data in XML form from the server. XML is an industry standard mark-up language, and is discussed further by Anderson (2002). For a client-server application it would therefore be possible to write the server portion of the code entirely in web services, so that there is a web service for each transaction type the client could make with the server. Web services are a key part of Microsoft's strategy for the future which basically seems to amount to a network of small, specific programs co-operating to perform larger tasks. The most notable example of this is their own Passport service, which provides user authentication and allows one log-in identity to be used at any supporting sites.

An ASP.NET web page can be created all in one file, but this approach is somewhat cumbersome. The preferred method is to split it into two files: an HTML file which defines the layout of the visual elements, and a code-behind file which is a standard class file representing the page and containing the associated code. This allows the layout and the code to be modified independently, which enables a web-site designer to work independently of the programmer and without necessarily needing to know

how to program. As with all other aspects of .NET the pages are object oriented, with the components (including the page itself) raising events and so on. The user interface of an ASP.NET application is created using Web Forms, which are described below. Web services are described elsewhere. Before they are used, the code files are compiled into a DLL file which is uploaded with the page to the server. The first time a page is accessed, a class is created to represent it by the runtime on the server which inherits the code DLL. This class is automatically compiled to a separate DLL file, which becomes the unit of execution for subsequent page accesses.

Each separate ASP.NET web page is a new type derived from the class `System.UI.Web.Page`. This includes properties such as the state properties Application and Session (see below) which the code may need to access. It raises events when the page is initialised, when the page is loaded, when it is unloaded, when an un-handled exception is thrown and so on. These can then be handled on the server. In addition to the pages and services, the application itself is an entity. As it is an object, it has events of its own which affect the entire application. The event handlers for this must be placed in the global application file. On a visit to a web site, a *round trip* is defined as: the user sending data to the server, the server processing it and then returning the page to the user. With .NET the page is initialised anew for each round trip - it is not constantly held by the server for the duration of the user session. This allows the server to handle multiple requests much more efficiently. After the initial processing stage, the page has a render stage that draws the page to be returned.

When the page (on the client) raises an event, the details are encoded in an HTTP post and sent to the server, which examines it and calls the appropriate event handler and passes the information to it. Some types of event are sent immediately to the server (the event on the client side triggers the next round trip) while other, less essential, events are cached and sent together at the start of the next round trip. These are postback and non-postback events respectively. Postback events generally occur when the user initiates an action (i.e., clicks on something) whereas events that may occur without the users knowledge are usually non-postback events. On a postback event, the pending non-postback events are handled first, followed by the postback event that initiated the round trip.

2.4.2 Web Forms

Web forms allow user interfaces to be created for web based applications on an XML/HTML page. They are based on reusable components (both UI components and server controls) in the same way as Windows forms. These pages can be viewed on any supported browser (e.g., version 3 and later browsers can be specified). HTML components and features can be mixed in with the web form components, and these are also treated as objects. A web form can be targeted to a specific platform: either a specific browser such as Internet Explorer 5 or a specific device such as one of the numerous types of mobile Internet access platforms (i.e., mobile phones). This allows pages / sites to be easily created that support all the features of the platform. Events that occur on web forms are covered in Section 3.1, and are different to normal events in that they occur on the server, but are triggered by the

client. Events that are raised on a component within a container are sent to the container. This raises an `ItemCommand` event on the container. This event contains an Item property that states which component raised the event.

2.4.3 State

There are two ways of storing the state of an ASP.NET application: Session state and Application state. Each ASP.NET application has Session and Application objects. As stated before, the page object is recreated for each round trip. While the client-side page will store its immediate state, for more complex information (and information that spans multiple pages on the site) the server needs to have a method for recording the state of that user's session. The session object is stable over all the pages in the application and is user specific. This allows the programmer to add and remove objects to/from a map inside the session state object and so allows objects to be stored to keep track of the users session. The application state object also operates over the entire application, but is "static": one Application object is shared over all users.

Application objects are instances of the framework type `System.Web.HttpApplicationState`. This type is a container for information to be stored globally in an ASP.NET application. It contains this information in the form of a hashmap. The Application object itself is a property that is created the first time the application is accessed. It is kept in memory for as long as the application runs, and so can lead to serious memory management problems for a server if misused (so it should be used for storing only small and/or frequently needed pieces of data). The data in the Application object is grouped into two collections: the `Contents` which are added by code in the application, and the `StaticObjects` collection. Objects can only be added to the latter by the Global.asax application configuration file (see above).

The Session object allows data to be shared across a user's session. As each round trip is a separate event in the server, it is essentially a different execution cycle. To keep track of what a user is doing (for example, the string inputted to a search engine) the server needs to be able to identify those page requests which come from the same user. It does this by assigning a session ID number which the browser provides to the page with each request. The browser normally keeps track of this ID using a "cookie" (a small data file on the browser's system) but if cookies are not available the ID can be appended to the web page URL. In use, the Session object contains the same collections as the Application object.

2.4.4 Configuration files

ASP.NET applications can be configured through the use of XML files. In any directory on the server, an XML file named Web.config specifies the configuration for files within that directory and its sub-directories. A Web.config file within one of the sub-directories can extend and override this configuration for its own directory and its sub-directories. The overall configuration for the server is specified in the Machine.config file. The

syntax of these files can be found in the .NET framework reference documentation, and is extensible if extra functionality is needed.

2.4.5 Web Services

.NET contains support for server side methods called Web Services. A web service type is created on the server in the same way as ASP.NET pages. Each web service method is equivalent to a static method on a type. An object of that type is created on the client and it can then call the methods exposed by the web service. Calls to the web service methods are made by sending the request in XML. The data returned from the web service is also in XML (or encoded using SOAP which uses XML as a transport medium). Web services are very simple to use, and provide a convenient way of providing functionality to an application. If web services are used to do all of the rear end work, it is far more efficient to create different front ends and applications that use them. Microsoft has a stated aim of dissolving applications into a network of web services, but this is ultimately a choice of the developers that use .NET. Java has also been extended to support web services as described in Databases Journal (2002). The Java version has different strengths which gives companies a choice over which to use, as discussed by Middlemiss (2002) and Rapaport (2002).

2.4.6 Java Server Pages (JSP)

Java's system for dynamic web pages is JSP. It has some disadvantages over ASP.NET, the main one being that the code and design are all in one file. This makes it more difficult for a designer to separately alter the appearance of the web pages. Rather than being a separate part of Java, JSP is an extension of Java Servlets. This heritage makes it slightly more complicated than ASP.NET, although this may have much to do with the superior development tools available for .NET. In general JSP is very similar to ASP.NET, probably as it was designed to compete with the original ASP.

3 Technical aspects of the runtime environments

3.1 Introduction

This section describes the more structural features of both .NET and Java. Discussed are the event, exception and threading models as well as the code security and garbage collection mechanisms.

3.2 Events

For the purposes of this section, a component is defined as being the source of an event. A container is defined as being an object that contains a component and (normally) handles any events it throws. The .NET Framework developer's guide compares the event system with the Observer design pattern (Figure 3.1.1). In this case the subject is the component that fires events, and the containers are the observers. In this case the user interface, container and anything else that can fire the event can send notification, but are not necessarily observers as an event listener has to be registered with the component for the event to be fired to it.

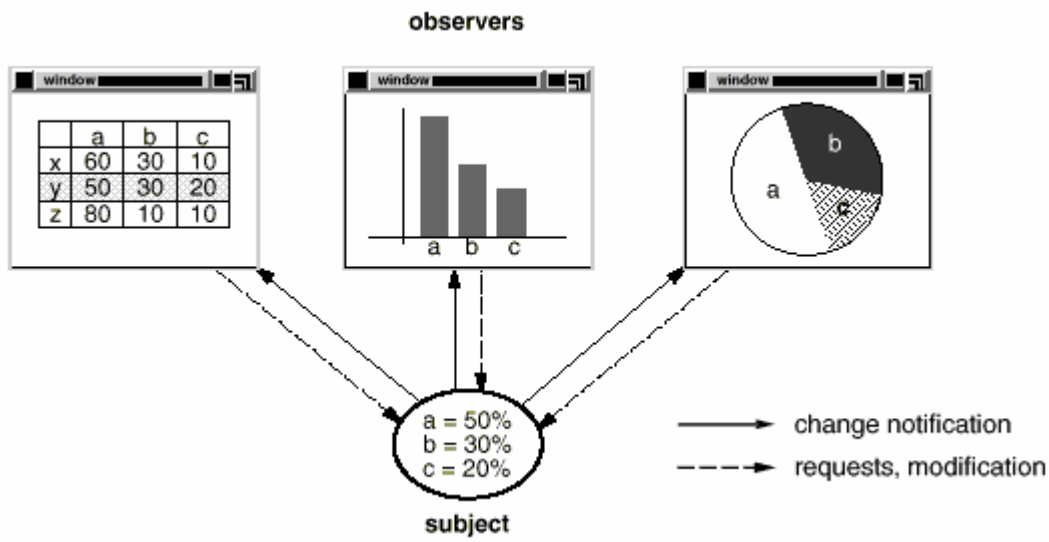


Figure 3.1.2 - The Observer design pattern. (Gamma, Helm, Johnson and Vlissides, 1995) This shows the relationship between observers and subjects in the observer design pattern. These can be related to aspects of the event model.

For a component to raise an event there needs to be: an event to be sent, a method to handle the event, and a connection between the two. As the component may be used with many different applications it cannot know which method in the application is to be used to handle it, so the method

call cannot be hard-coded into the component. For example, when a button is placed on a form it does not automatically know which method to call when it is pressed. The .NET solution to this is to use its (safer) version of function pointers: delegates. As described elsewhere in this document, a delegate contains references to one or more methods, which are then invoked when the delegate is called. If a delegate type is created for each type of event, the component will always know that the container's event handling method will be referenced by an instance of its associated delegate. All the object that contains the component has to do is register the event handler with the component by telling the component which instance of the delegate to use. There are therefore three parts to an event in an application, the event in the component, the delegate in the container that is used to register the event handler method with the component, and the method that handles the event.

Part of the nature of delegates is that the delegate and the handling method must have the same signature (i.e., return type and parameters). Events are, by nature, one way messages so the return type will always be void. The parameters are always the same as they convey the two pieces of information the handling method needs to know. The first of these is the identity of the component that sent the event. This is particularly useful when there are more than one components that fire that type of event and the same handling method is being used for all of them. It is not always sufficient just to say that an event occurred, as extra information is sometimes necessary. For example, for a KeyDown event, the handler may need to know *which* key was pressed. This additional information is passed to the handler in the second parameter of the event handler, the event arguments parameter. For each event type that contains additional information, a custom sub-type of System.EventArgs can be created. An object of this type is then created and passed to the event handler by the component whenever the event occurs. If customised information is not needed, then an instance of EventArgs itself can be used.

The naming convention in .NET is as follows: if the event is called X, the delegate type is called XEventHandler. If there is a method to handle that event for a specific instance of the component, the method is called name_XEventHandler (where "name" is the instance name). The EventArgs sub-type (if one is used) is called XEventArgs. This is summarised in Table 3.1.1. As stated before, if there are multiple components within a container that fire the same event, the same handling method can be used for all of them. In this case the "name" part of the method name should simply be something suitably descriptive, although as this is only convention any method name could be used.

The delegate here would be declared as follows:

```
public delegate void XEventHandler(object sender,
                                   XEventArgs e);
```

		Name (by convention)	Location
i)	Event	*X*	The component
ii)	Event arguments object	*X*EventArgs	Passed as a parameter
iii)	Method to fire the event	On*X*	The component
iv)	Delegate	*X*EventHandler	The container
v)	Event handling method	*name_X*EventHandler (often; can be anything)	The container (usually; can be anywhere)

Table 3.1.1: Naming conventions for the .NET event model.

The parameter sender is the reference to the component that raised the event. Within the component that fires the event, there must be the event itself. A component's events are defined by the keyword *event*. The type of the event object is the associated delegate event type. As stated previously, the invocation list of a delegate can contain multiple methods, or other delegates, as long as they all have the right signature. When the delegate is called, all of these are then called. This allows the event object in the component to reference all of the message handlers that have been set up to handle the event. To do this, the container must add its handler delegate(s) for the event to the component's event object. In the case of the above naming examples, event X would be declared in the component as follows:

```
public event XEventHandler X;
```

If the container contained a method to handle this called "test_XEventHandler", then it could be added to the list of handlers stored in the event object by creating a delegate to point to it, and then adding that handler delegate to the invocation list of the event delegate, X. The simplest way to accomplish this would be to add the following to the code that adds the component to the container:

```
test.X += new XEventHandler(test_XEventHandler);
```

The event object within the component (in this case X) then holds an invocation list of all the event handlers that have been registered with it in this way. The event keyword may not have an analogue in all .NET languages and the variable will have to be defined as an object of the type of the delegate. The methods for adding and removing handler delegates to and from that object will also need to be provided. Within a component, events are raised by the appropriate On*Event* method. This is a necessary method in a component with an event and in the case of the above example, would be called OnX. This is the method that is called when the event is raised. It is used to fire the event to the event handlers. The implementation of this

method must be provided by the author of the component, but in general looks like the following:

```
protected virtual void OnX(XEventArgs e)
{
      if(X != null)
      {
            X(this, e);
      }
}
```

This shows the most basic (usual) form of the On*Event* method, which checks that there is a registered event handler before calling them (note that *this* is the keyword referring to the current object, so is the sender object given to the handlers). The event is raised within the component by creating the appropriate EventArgs object and passing it to the On*Event* method. There are also events that do not need handlers, where the On*Event* method within the component is used to perform necessary operations. A specific example of this is the paint event in custom web forms components. The OnPaint method (which in this case includes different parameters to normal) is overridden to provide a graphical interface for the component, and the code within draws the component.

3.3 Events in ASP.NET Web forms

In web forms, the event model has to change to take into account the fact that most events will be raised on the client (the downloaded page) and handled on the server. The only events raised on the server are those where the component is responsible for the event being triggered and not the user. This separation requires the client to send a message to the server with details of the event. This means that web forms components have far fewer events to avoid crippling the Internet connection and server with requests. As an additional step towards this goal, ASP.NET allows events to be specified as postback events or non-postback events. A postback is when the page posts its data back to the server. Postback events cause this to occur immediately so the event data is received as soon as possible, where non-postback events must wait until the next postback occurs. The server automatically interprets the message containing the waiting queue of events and the server side code handles it accordingly. The event handling methods take the same form as those in standard code, although delegates are not needed as the connection is done automatically by the framework (the handling method is specified when declaring the control).

The events that are raised as well as handled on the server are handled in largely the same way as in standard code. There are several types of event that are specific to server controls however. These deal with the life-cycle of the components, and the fact that they are loaded, unloaded etc. This is necessary because of the nature and restrictions of accessing web pages. This is beyond the scope of this section but the concepts are discussed in the section on ASP.NET.

3.4 Events in Java

The event model in Java is, due to the object-oriented component design of both systems, naturally similar to .NET's. There are syntactical variations (e.g., event listeners instead of event handlers) but the principles are the same. The major difference comes with .NET's use of delegates, which are not present in Java. The event types derive from java.util.EventObject. Event types in Java are the objects that the event passes to the listeners (in other words the equivalent of .NET's EventArgs types). The method of creating an event type is also almost the same as for an EventArgs type.

The component needs to have methods for adding and removing the event listeners - as does .NET, although in languages with event support (such as C#, which is being used for the purposes of this document) the .NET framework automatically generates these. Instead of .NET's event delegate there is a list object, to which the listener references are added and removed by the listener control methods. It also has a method which (equivalently to the .NET On*Event* method) fires the event to all registered listeners for that event. This method must use a loop to send the event object to every one of the listeners in the list, however (unlike .NET where only one call needs to be made to the event delegate).

A class that handles an event from a component must implement a sub-class of the java.util.EventListener interface. This is the *adapter* class. It listens for any events that are fired by the component and calls the appropriate method to handle them. It is roughly equivalent to the delegate in .NET, in that it connects the component's event with the event handling method in the container. The container can instantiate an object of the adapter class or can contain the adapter class as an anonymous inner class, and adds the adapter as a listener in the same way as .NET adds the delegate. The separate adapter class is needed because in Java the event does not pass a reference to the component that sent it, so with no way of knowing the source of the event the container may not be able to effectively handle events of the same type from two different components. Separate adapters are therefore needed for each component in the container that generates an event of that type. They can then see to it that the appropriate method handles events from each component.

```
buttonControl1.addFocusListener(
            new java.awt.event.FocusAdapter()
            {
                  public void focusGained(FocusEvent e)
                  {
                     buttonControl1_focusGained(e);
                  }
            });

void buttonControl1_focusGained(FocusEvent e)
{
  // your code to respond to event goes here
}
```

The above example (from the Borland JBuilder JavaBeans documentation) adds an event listener using an anonymous adapter class. Compare this with the .NET example above where a new delegate pointing to the handler method is added to the component's event handler list. Although the Java version does this in a different way and seems more unwieldy, they are very similar in use. In each case a new instance is created as it is added to the event's handler list. In Java, a specific method (defined by the interface) must be present in the adapter which calls the handler method. The difference is that in .NET the event is given a reference to the method and calls it itself. The Java event model thus incurs the extra overhead from instantiating the extra class and the extra method call. There is also a scaling loss with Java when multiple components are firing the same type of event. With the source parameter in .NET telling the handler where the event came from, one delegate can be used to add a single handling method to all the components. In Java, although a single (not anonymous) adapter object could be used, the handler could not differentiate between the events. To do that multiple adapter objects are needed which further decreases the efficiency of Java's approach in these circumstances.

As can be seen in the above example, the adapter class must implement the methods in the appropriate listener interface. This allows a single listener type to listen for several different events by declaring multiple methods, in which case the associated adapter must provide implementations for all of them. This does have the advantage that when used for a logical grouping of events, all of them must be handled. Unfortunately, the unavoidable consequences of this are that it may then be necessary to handle more events than necessary (or at the least, to create and call blank methods where there are no handling methods). While this can be easily avoided when creating custom events, when using built in events in Java or other developers' components it can result in wasted method calls and the resulting performance penalty.

		Name (by convention)	Location
i)	Listener list	ListenerList	The component
ii)	Event	*X*event	Passed as a parameter
iii)	Method to fire the event	process*X*	The component
iv)	Listener	*X*listener	The container (often)
v)	Event handling method	name_*X* (name is the component object name)	The container

Table 3.1.2 - Naming conventions for the Java event model. Each item most closely corresponds to the numbered .NET equivalents in Table 3.1.1.

3.5 Exception handling

When an error occurs during the execution of a program, the program needs to be able to find out what happened and deal with it. This is done in both Java and .NET with *structured exception handling*. This means that the error is noted at its source and an exception object created immediately. This object is then passed back through the terminating functions until it reaches an exception handler that has been set up to deal with it. With .NET, while each language has its own syntax for exception handling, exceptions thrown in code using one language can naturally be caught in code using another. The most common syntax for exception handling is to have a *throw* command to throw the exception upon the occurrence of the error and a *try* / *catch* block to handle it. In a piece of code, the instructions that may throw an error are placed within the try block and the exception(s) are handled by one or more catch blocks. In .NET, exceptions can be caught even on different machines in a distributed process. The catch block can extract information about the error from the exception object to allow it to handle it more effectively.

3.6 Exception classes in .NET

An exception in .NET is represented by an object. To be CLS compliant, this object must extend the framework class `System.Exception` (or a sub-type). This object allows detailed information on the error to be passed to the error handler. There are two subtypes of the Exception class: SystemException and ApplicationException. SystemExceptions are usually thrown by the runtime and its subtypes are usually the predefined framework exception types. ApplicationExceptions are general application errors thrown by a program. Subtypes of this exception class are generally written and used by application developers. It is possible for exceptions to be chained together so that symptomatic errors can reference the exception that led to them.

3.7 Using exceptions

When an error occurs within a piece of code that is fatal for that piece of code (e.g., a file is not opened correctly in a method to read its contents), the code should immediately terminate and return to a point when the error can be handled. To do this it needs to *throw* an exception. At the point the error occurs, an instance of the appropriate exception type needs to be created. This can then be loaded with any appropriate data the handler may need. The possible data elements that could be included are represented by the parameters of System.Exception (see Table 3.2.1) and any parameters that are additional to the specific exception type being used. As is alluded to in Table 3.2.1, the constructor of the exception type is often used to pass all the appropriate data in to the exception instance to simplify the process of throwing the exception. This exception instance is then thrown using the throw keyword. For example, if there is an exception instance called testException, it can be thrown by the statement: "`throw testException;`".

Properties		
	HelpLink	This is a universal resource name (URN) or a URL pointing to a location where an explanation of the error can be found.
	InnerException	When an exception is caught and another error is thrown by the handler as a result (to be caught further up the execution tree), the initial exception may be important. The causing exception can be referenced in this property of the resulting exception so the programmer can create and later follow an exception chain to trace the causes.
	Message	This is a string containing an error message.
	Source	This is a string naming the source of the error (i.e., the object or application name).
	StackTrace	This is a string containing the stack trace when the exception was thrown.
	TargetSite	This identifies the method that threw the exception.
	Hresult	The HResult value is for compatibility with COM.
Methods		
	Constructor	The constructor is often the only part that is changed upon creating a new exception class.
	GetBaseException	If there is a chain of exceptions (see InnerException), this method returns the exception at the base of the chain (i.e., the original one).

Table 3.2.1 - Excerpted members of .NET type System.Exception. These members summarise the functionality of the exception object.

When a method is to be called which may throw an exception that the programmer wishes to handle, it should be called from within a *try* block. If an exception is thrown by code called within the try block and there is an appropriate exception handler, control will pass to the handler. .NET's exception handling model is a termination model: all instructions that would have been executed between the exception being thrown and the exception being caught are ignored. The current method is terminated and control is passed up through the hierarchy of called methods until the appropriate try block is reached (the rest of which is skipped). Control then passes to the appropriate catch method. If there is no appropriate exception handler the method containing the try/catch block will be terminated and the exception will be passed up through the call stack to the method that called it and will keep going up until it reaches another try block where it will see if there is an appropriate handler and so on. It will keep terminating the methods and moving up through the call stack until it is successfully handled. In this way, exception handling methods can be nested so errors

are passed up along the line of control until they reach a handler that knows how to deal with them.

If an exception is raised within the try block, it is handled by a *catch* block. There can be multiple catch blocks attached to a try block, each of which can be used to trap and handle different types of exception. The type of exception that can be handled by the block is specified after the catch keyword. The catch block will also catch sub-types of the specified exception so if the exact exception is not important, a more general catch block can be used. If no exception is specified by the catch block, it will catch every exception (the "shotgun approach" to exception handling). As mentioned previously, if there is no appropriate catch block, control passes up the chain of execution. The catch blocks are processed in a linear manner in the order they appear in the code, so if the specified exception does not match the thrown exception, the runtime moves on to the next catch block. The first matching catch block is executed and that is the only catch block executed, so more general catch blocks should be placed later to avoid hiding specific ones. Within a catch block, the keyword *throw* mimics the effect of there being no appropriate catch block and re-throws the exception to be caught further up the chain of execution.

As .NET uses a termination model, any cleanup code that should be carried out at the end of a method would be skipped. This would waste resources and place unnecessary load on the garbage collector. To get around this, there is an optional third part to the try/catch block: the *finally* block. The code within the finally block is **always** executed. If cleanup code is put in the finally block, it will therefore be executed whether an error occurs or not. As this block is used primarily for this purpose, it is executed **before** the catch block if it is present. This syntax is summarised in Figure 3.2.1.

3.8 Exception handling in Java

As stated in the introductory paragraph to this section, Java also uses structured exception handling. The base exception class in Java is Throwable. It has two sub-classes: Error and Exception. These deal with internal system errors and program errors respectively, and as such are comparable with the two sub-classes of the Exception type in .NET. The exception types to be used in applications derive from Exception. As with .NET, there are exception types in a hierarchy that cover the common types of exception (e.g., input and output errors). There is also a sub-type of Exception called RuntimeException, which is the base class for exception types representing code errors such as invalid array indexes and so on. These built-in types do not have the same flexibility as .NET's parameters do for passing data to the handler, with a message string and stack trace being the only real data. This message string is returned to the handler simply by the object's toString method or a getMessage method and the only other information the handler has is the exception type itself.

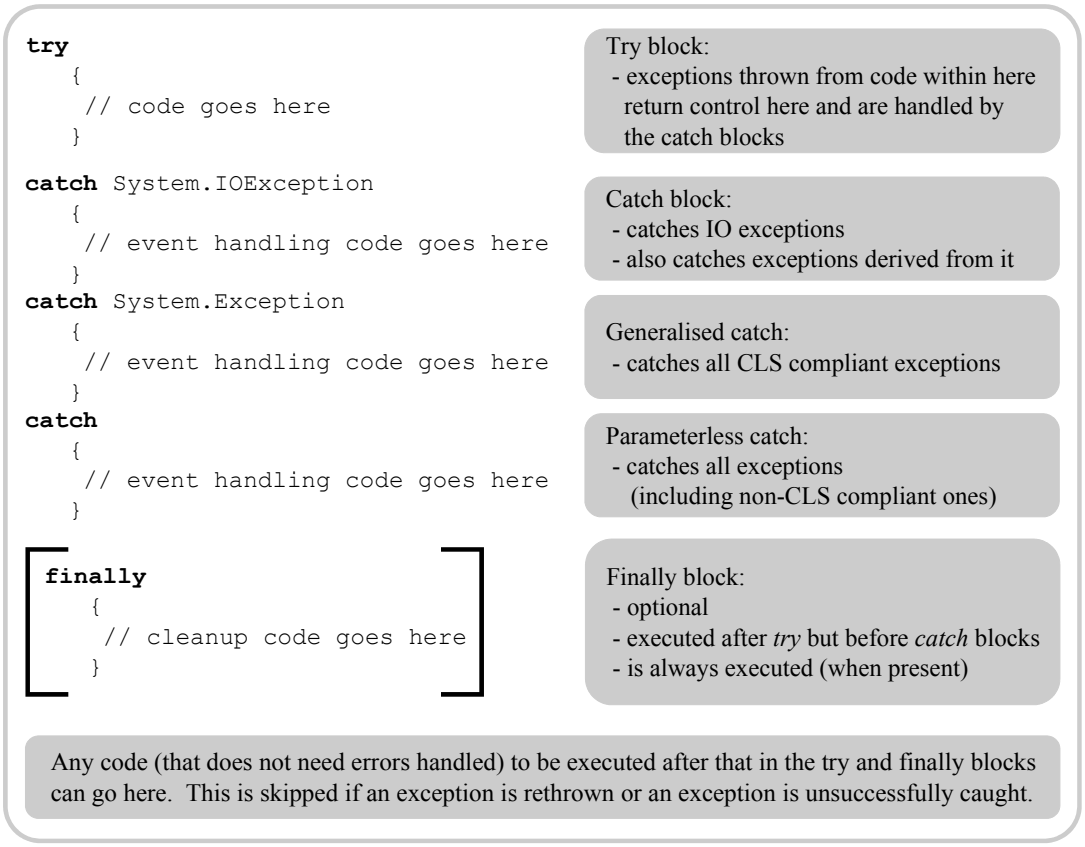


Figure 3.2.1 - Summary of .NET exception syntax. This figure illustrates the usage of exceptions in .NET

Java differs in its exception model from .NET in that Java publicises the exceptions that can be thrown by a method. This is done using the throws keyword in the method declaration, as shown below:

```
public void setName(String s) throws IOException
{
        // method body…
}
```

The method declaration can specify any number of exceptions (separated by commas). This specifies what exceptions a method is allowed to throw (in the above case, only those exceptions derived from IOException). Exceptions are thrown in exactly the same way as in .NET. Exceptions in Java are tested for using the same methodology as .NET: the try/catch/finally block. There is no parameter-less catch block in Java, but all the exceptions that can be caught derive from Exception so `catch(Exception e)` is the equivalent (as stated above, the purpose of .NET's parameter-less catch block is to make sure that exceptions from unmanaged code are caught as well). One other difference is that the finally block is executed after the catch block in Java (it is executed before in .NET) which can lead to problems when exceptions are re-thrown in the catch block.

Although Java's ability to restrict the exception types a method can throw is useful, it can be both a help and a hindrance when extending classes if an additional exception becomes necessary. While this can be avoided at the design stage, it is more problematic when using modular design techniques and creating a set of base components. The major difference between the two systems is the structure of the exception objects. Specifically, the ability to chain exceptions in .NET allows for much greater flexibility with complex applications.

3.9 Threading

Both Java and .NET are multithreaded environments (i.e., they allow several parts of applications to run simultaneously). An application that uses multiple threads has the advantage that parts of it can be running while other parts are waiting for something to happen (as with processes in a computer CPU). Threads can also have different priorities, so a low priority task can be run in the background rather than having to complete before the program can move on to the next task.

3.9.1 Threading in .NET

In .NET, the namespace System.Threading contains the classes used for multithreaded applications. Of these classes, it is System.Threading.Thread that is used to control threads. As it is the core of the threading system, its members are described in more depth in Tables 3.3.1 and 3.3.2 below. Threading is not without problems, as discussed by Reilly (2001), so these members need to be simple to use. To create a new thread, a new object of type Thread is created. To successfully initialise the thread, a delegate *must* be passed in the constructor. This delegate (of type `ThreadStart`) references the method which is to be called when the thread is started. The thread instance can then be started by calling its Start method. The Thread type contains methods for controlling the state of a thread instance. These methods (the bottom section of Table 3.3.1) allow the threads in an application to be started and stopped as and when they are needed to allow the efficiency of the application to be maximised. The thread type also contains properties that can be used to find the state of a thread instance and control things such as the priority of the thread.

There are also static members of the Thread type that are used to reference or control the threading system in general. The properties of Thread are useful to an application in that they can access the current state of the application (see the top half of Table 3.3.2). The property CurrentThread is a good example of this, as code can then alter the status (e.g., the priority) of its own thread. The Sleep method is a shortcut for calling suspend on the current thread. For example, if code needs to pause between operations (e.g., waiting for an update to propagate through before reading from a data source), Thread.Sleep() allows it to suspend itself. The other methods present in Thread allow manipulation of the data slots assigned to threads.

Properties		
	ApartmentState	This is for COM compatibility. This property (which must be a member of the ApartmentState enumeration) is normally "unknown", but can be set to indicate to unmanaged code whether the environment is single-threaded or multithreaded.
	IsAlive	This property is a Boolean value that is false if the thread has not yet been started or has permanently ceased (died), and is otherwise true (i.e., if the thread is valid).
	IsBackground	Background threads only execute while a foreground thread remains. If a thread is designated as a background thread, it will therefore abort when the last foreground thread finishes. This property is a Boolean variable that specifies the background status of the thread.
	Name	This property stores the name of the thread if it has one.
	Priority	This property is the priority of the thread. This is a member of the ThreadPriority enumeration: Highest, AboveNormal, Normal, BelowNormal or Lowest.
	ThreadState	This property has the value of one of the members of the associated enumeration ThreadState. It describes the state the thread is in.
Methods		
	Start	This method starts the thread. It calls the delegate (and therefore the method that it references) that was used to initialise the thread instance.
	Abort	This method raises an exception to stop the thread. Any finally blocks that are present within the thread are called before it stops.
	Suspend	Pauses a running thread.
	Resume	Restarts a thread at the point at which it was suspended by a call to Suspend.
	Interrupt	Interrupts a waiting (blocked) thread. This unblocks the thread and throws an exception to it. If the thread does not catch the exception, it is aborted.
	Join	A call to this method pauses the code until the thread the method is called on has finished. This method also has an alternate use. A time interval can be passed to Join as a parameter and the method returns a Boolean value to indicate if the thread has finished.

Table 3.3.1 - Instance members of the .NET type System.Threading.Thread.

These are the thread specific functions that are supported.

Properties		
	CurrentContext	This references the context that contains the current thread.

CurrentPrincipal	This property is related to the security system. Essentially it is the role of the user (the group the user is in). See the section on code security for more information about principals.
CurrentThread	This returns a reference to the Thread instance that is currently executing.
Methods	
AllocateDataSlot	These methods are used to control the thread data slots.
AllocateNamedDataSlot	
FreeNamedDataSlot	
GetNamedDataSlot	
SetData	These allow access to the data slots on the current thread.
GetData	
GetDomain	A thread can operate in more than one application domain. While it can cross the boundary between application domains, it can only be in one application domain at any one time. This method returns the current application domain.
ResetAbort	Resets a previous Abort
Sleep	This method is similar to Suspend, except: it always acts on the currently running thread as opposed to a specific thread (it is static) and it suspends the thread for a specified duration.

Table 3.3.2 - Static members of the .NET type System.Threading.Thread.

This is the functionality that either applies to all threads or just the currently executing thread.

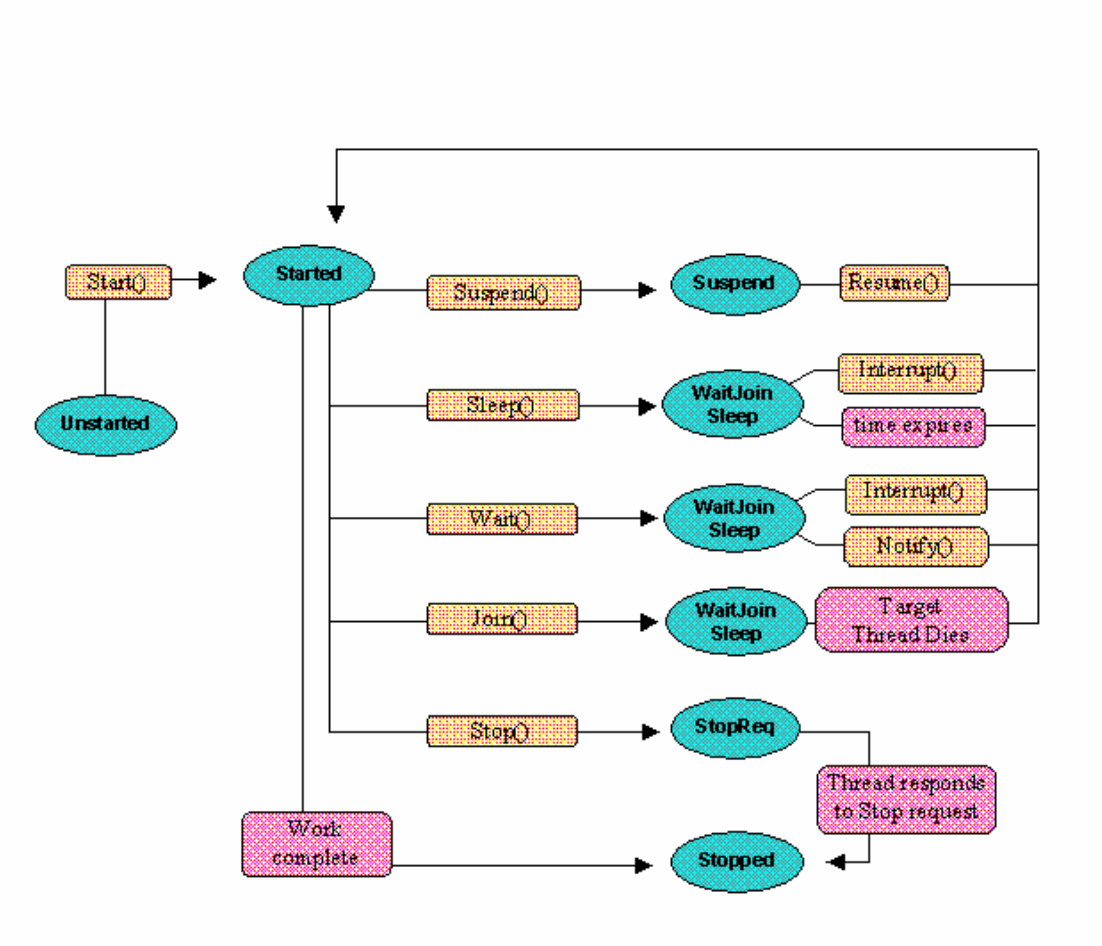


Figure 3.3.1 - .NET Thread state diagram. (taken from the .NET Framework Developer Specifications in the .NET SDK)

Synchronisation

When there are several threads in an application that need to access the same resource, the order in which the threads get access to it cannot be known. This will give a synchronisation problem for the resource. To solve this, the threading system needs to have a method of controlling thread access to the resource, as a database which is used by more than one user at a time must implement locks to maintain data integrity. Synchronisation is performed in .NET using critical sections (referred to here as synchronisation blocks) and monitoring. The type in the .NET threading system which is used to provide synchronisation is `System.Threading.Monitor`. As this class is purely for control it contains only static methods. With any code that accesses a protected resource it must be locked, so that it can only execute once the lock has been removed (i.e., when there is no other thread accessing the resource). It is therefore placed within a synchronisation block). There are two ways of doing this in .NET. The example in Table 3.3.3, below, is taken from the C# language specification, and compares the standard .NET synchronisation lock block syntax and the C# specific syntax.

.NET monitor block syntax	C# syntax

System.Threading.Monitor.Enter(x);
try
{
…
}
finally
{
 	
System.Threading.Monitor.Exit(x);
}
	lock(x)
	{
	…
	}

Table 3.3.3 - Comparison of standard and C# synchronisation lock syntax.
(C# Language specification 8.12 - Microsoft, 2000)

The C# syntax is preferable not only because it is much simpler, but also because it automatically notifies the runtime when it exits the block. The Exit call in the .NET syntax is in the finally block as it must inform the runtime that it has exited the protected section for the runtime to be able to release the resource to another thread. The parameter "x" in Table 3.3.3 is the object on which the monitor lock is granted. Upon calling Enter (or lock in C#), the runtime checks to see if a monitor lock has been made on that object. If it has not, the thread is allowed to enter the block, and a monitor lock is placed on the object until the thread exits from the lock. A thread will be blocked if there is another thread that has already entered a monitor block for that object. When the thread exits the block (i.e., makes the Exit call or finishes the block in C#) the blocked thread is unblocked and can then enter it. The object on which the lock is gained (passed as the parameter x) is usually "this" or typeof(name) where name is the name of the class in which the monitor block is found, which place the block on the object containing the monitor block or the static members of the type respectively. When the system allows access to a synchronisation block to a thread, it is actually allowing access to all synchronisation blocks locked with that object. If using the framework syntax, there is an alternative to Enter called TryEnter. This method does not block the thread if another thread has a monitor lock on the synchronised block as Enter does. Instead it skips the block and returns a Boolean to say whether it entered or not. For example, this is useful for a block which only needs to be executed once (i.e., to update a variable shared by all threads).

The threading system maintains a queue of the threads that are waiting for the release of an object's synchronisation lock. These are allowed access in order as each of the previous threads releases the lock. This system of allowing access to only one thread at a time is the simplest form of synchronisation. However, there are times when a thread needs to pause in the middle of a synchronisation block and allow access to another (either to the same block or another locked with the same object). This allows the blocks themselves to control thread access. To this end, there are two other important methods in Monitor: Wait and Pulse. These can only be used

within synchronisation blocks. When Wait is called within a synchronisation block, that thread enters the queue of threads waiting to be reawakened by it. The next thread waiting to enter a synchronisation block locked with that object will then be allowed to enter the block and continue as the active thread. When a thread has done all it can in a block and wants to allow a thread that is in the wait queue to resume, it calls Pulse. This method pulses the queue waiting within the synchronisation blocks, which wakes up the next thread in the queue, which resumes when the thread that called Pulse releases its lock. As a waiting thread needs to be woken with a pulse, its use has to be controlled to avoid the thread being stuck waiting indefinitely. There is also a variant on Pulse, PulseAll. As its name implies, PulseAll wakens and moves all of the threads in the waiting queue to the ready queue. When the current thread releases its lock, the next thread in the queue resumes and so on until the ready queue is empty again.

3.9.2 Threading in Java

In Java the base class for threads is also called Thread (or in full: java.lang.Thread). The first complication with Java's threading system as compared with .NET's is again the lack of delegates in Java. This means the starting method for a thread to call cannot simply be referenced and passed to the thread. Java provides two ways of doing this. If a thread is going to be created often that starts at the same method, then the Thread class itself can be extended and its run method overridden. This means that the code itself can also be placed within the thread's run method to create a thread that performs a particular task independently. This is a cumbersome way of creating a thread as it results in unnecessary classes being created, but may prove useful for common standalone tasks. The second way of referencing the method to start a thread is to just to put the run method into that class. That method can then either call other methods to perform tasks or perform them itself. The class that is to have the run method to start a thread must implement the Runnable interface. To instantiate a thread using this second method, an object of type Runnable must first be instantiated and passed to the thread object in its constructor. Table 3.3.4 describes the members of the Java thread class.

To synchronise access to a method, the modifier *synchronized* must be used when declaring the method (i.e., `public synchronized int getIndex() {…}`). This then disallows more than one thread from accessing the method at a time. When a thread calls the method it is locked, and any other threads calling the method must wait in a queue to access it sequentially. This is roughly equivalent to enclosing the entire method body within a `lock(`*`this`*`)` block. However, in .NET any other synchronisation blocks in the same method that were also locked using *this* are also blocked, whereas in Java all synchronised methods are treated as separate. There is no analogue of the Monitor class in Java, however the base object class contains the methods wait, notify and notifyAll that perform the same duties as Wait, Pulse and PulseAll in .NET's dedicated monitor class.

Instance Methods	
run	As described above.

	CheckAccess	Checks the permission of the current thread to modify this instance.
	GetThreadGroup	Returns the group the thread is in.
	getName, setName	These are similar to the .NET versions.
	getPriority, setPriority	These are similar to the .NET versions.
	interrupt	This is the similar to the .NET version.
	IsInterrupted	Checks to see if the above method has been called.
	IsDaemon, setDaemon	The Daemon attribute is equivalent to Background in .NET
	isAlive	This is the similar to the .NET version.
	Join	This is the similar to the .NET version.
	Start	This is the similar to the .NET version.
	Destroy	This is not implemented (as of Java 1.2) as it would cause the same problems as below.
	CountStackFrames	Deprecated (relies on suspend)
	stop	Deprecated as the lack of finally blocks attached to synchronisation blocks means threads cannot be safely aborted in Java without leaving inconsistent information in them. In .NET the finally blocks are carried out before an abort, so synchronised blocks can ensure they maintain their integrity.
	Suspend	Deprecated as it may result in deadlocks. This should not be a problem for any halfway competent programmer, so .NET still includes this method.
	Resume	Deprecated as a result of the above.
Static Methods		
	currentThread, sleep	These are similar to the .NET versions.
	activeCount	Returns the number of active threads in the active group.
	DumpStack	Returns the stack trace for the current thread.
	Enumerate	Puts references to every active thread in this group into an array.
	Interrupted	Returns true if current thread has been interrupted.
	Yield	The current thread suspends to allow higher priority threads to continue, then it resumes.

Table 3.3.4 - Selected methods from the type java.lang.Thread.

3.9.3 Conclusions

While Java's might initially look the more flexible system, the run method is analogous to the *main* method and as such is a more procedural approach to threading. It also requires that the application developer know in advance when a type is going to be used to initiate threads. This reduces the flexibility

of components to be linked together by an application developer. If a developer is calling multiple methods from within a code block and decides it would be more efficient to run them in separate threads, in Java a sub-type of Thread would have to be written for each one. The only alternative to this (using Runnable) is for each method to be part of a different type, and for each of these types to only support threads starting on that one method. Both of these are significantly more inefficient than simply passing the method name to the delegate when constructing a thread in .NET. The greater variety of control over synchronisation of threads in .NET affords a great deal more power and flexibility to the programmer than Java, which is more simplistic.

3.10 Code security permissions in .NET

The code security mechanism in .NET is based around a system of permission objects. A piece of code either requests or demands the permissions it needs. The security system in the runtime decides whether to grant those permissions. It does this when the assembly is loaded. If the code *demands* any permissions that are denied, the runtime will not run it or any dependent code. To access a protected resource, all methods in the calling stack must have the required permission (in other words, the calling application and any code it has called on the way). The runtime decides whether to award permissions based on the security policy. There are three types of permission. The most common category of permission is the code access security permissions. These are used to protect code or resources from being used by code that has no authorisation to do so. A basic analogy would be the sandbox of Java applets: the code on an Internet server has no permissions to access files on the user's computer. The second type of permission is identity permissions, which are used to verify the claims of an assembly to have a particular identity. These are related to of code access permissions. The last type of permission is role-based security permissions. All permission objects share one thing: they all implement the `System.Security.Permissions.IPermission` interface. This provides the base set of functionality that all permissions need. Selected methods from IPermission are detailed in Table 3.4.1.

While the Boolean operations enable more complicated permissions to be used, it is Demand that is the core of any Permission object. This is the method that initiates the security check to see whether the user or application has got the appropriate permission. A call to Demand on a permission object before a piece of code results in the piece of code only being executed if the user/assembly (depending on whether it is a role-based or code access permission) has the associated permission.

Demand	This method calls the runtime to perform a security check to see if the user or assembly has been granted the appropriate permission. Implementing classes will perform this in various ways but the following is usually true. If the security system grants the permission, the code after the demand is executed. If permission is

	denied, a SecurityException is thrown.
Intersect	This method returns a permission that is the Boolean intersection of this object and the permission object passed in as the parameter.
IsSubsetOf	Returns true if the current permission object is a subset of the specified permission object.
Union	Returns a permission object that is the Boolean union of this object and the specified permission object

Table 3.4.1 - Selected members of the IPermission interface.

This is the base interface that defines a permission and the base functionality it must have.

3.10.1 Code access security

Code access security is the mechanism by which an assembly is only given permission to access the resources that the administrator trusts it to use responsibly. This is to take account of the fact that the standard model for installing software has changed from that at the time when the previous DOS/Windows programming libraries were developed (i.e., STL, MFC and also to an extent Borland's VCL). These libraries were developed when it was assumed that code was bought from the developers and installed from the provided media. The current model for software distribution is vastly different, with code being available from the Internet, through local area networks and so on. The move towards distributed software, though it is not certain to what extent this will happen, also has to be allowed for in a modern API. This means that not all code can be trusted, and hence that there must be some way of authentifying code and deciding what permissions it should have. The process of deciding what permissions code can have is described in the section below on the .NET framework security policy.

If it were just the security policy, many programs would fail if a necessary permission was not granted. The code itself needs to be able to know about permissions. The code access security mechanism allows the programmer to demand essential permissions, request additional permissions and refuse permissions that would be unnecessary/harmful. If a permission is demanded that is refused by the security system, the runtime will not execute the code. This prevents untrusted code from doing harm (or, more accurately, from doing anything). The refusal of a permission that is merely requested is not fatal for the code, but whatever operation that it needs the permission for will fail. For example, if a program such as a word processor was refused a permission to write files on a system it would still run, but could only open files as read-only (as with MS Word Viewer).

For a piece of code to access a resource, not only must *it* have the required permission, but so any code that calls it must have that permission. This prevents a piece of "wrapper" code that has the necessary permissions from circumventing the security system by opening a protected resource to any unsafe code that calls it. This would cause a lot of problems if there was no way for code to find out if it lacks the required permissions to call a piece of code, so the security system also allows code to specify what permissions are required to run it. Permission objects can be used within code to issue

demands to the runtime. This causes the runtime to perform a stack walk to check that all of the callers in the stack have the required permission. If they do not, an exception is thrown and the code following the demand is not executed. This allows custom permissions to be created and required by the code.

All code access permissions derive from `System.Security.Permissions.CodeAccessPermission`, which itself implements IPermission (see above). To create a custom permission, this or an existing sub-class of it can be extended. The methods that provide the basic functionality of the permission are described in Table 3.4.2. As stated, the key method for a permission is Demand, but there are also methods to control other aspects of the availability of the resource protected by the permission.

Instance methods	
Assert	This method is dangerous. While it can make certain tasks easier for a programmer, it can also compromise the permission and lead to a hole in the security system. This method stops the permission check on code earlier in the call stack than the code that calls this method (by default, all calling methods in the stack must have the correct permissions) and asserts that they qualify. To do this, the code that calls this method must have the correct permissions. While code without the correct permissions may use the code that calls this method to access protected resources, it should be noted that this could be malicious code. The assertion is valid until the code that calls this method returns.
Demand	This is the implementation of the IPermission method in Table 3.4.1. Upon this method being called, the security system performs a stack walk and examines the permissions of all code in the call stack. Fails and throws an exception if not all of the code has the requisite permissions.
Deny	Stops any code earlier in the call stack than the calling code from accessing the associated resource (i.e., whatever comes after a call to Demand).
PermitOnly	The inverse of Deny. Where Deny stops code from accessing a protected resource, PermitOnly stops the code from accessing any resource *except* the one specified by this permission object.
Static methods	
RevertAll	Calling this method is equivalent to calling all of the below methods. All of the revert methods are static: they act on *all* instances of the permission.
RevertAssert	If Assert has been called, neutralizes the effects (essentially undoes the assert). All code in the stack once again has to have the correct permission.
RevertDeny	As with the above, but undoes the effects of a Deny call. The resource is once again available.

RevertPermitOnly	As with RevertAssert, but undoes the effects of a call to PermitOnly. All other resources are again made available (dependent on permissions).

Table 3.4.2 - Selected methods from the .NET type CodeAccessPermission.

There is a type of code access permissions known as *Identity permissions*. These require the code that is trying to access a protected resource to have a particular identity. These differ from the role-based permissions below in that it is the identity of the code, rather than the identity of the user that needs to be proven. Identity permissions are granted to an assembly if it can give the runtime security system the appropriate evidence to support it having that identity. Table 3.4.3 lists the five identity permissions built in to the .NET framework and describes the evidence needed for the code to be granted that permission. By demanding an identity permission in a piece of code, the assemblies that can access it can be controlled. Whereas the security settings can be altered by an administrator to change the permissions granted for other code access security permissions, the identity permissions allow absolute control over what assemblies can access the code. For example, if a developer is writing a class library to be used with their products only the library could use the PublisherIdentityPermission to demand the calling code must have the developer's Authenticode certificate. Use of the StrongNameIdentityPermission would allow plug-ins to be used only by the application they were written for, and so on.

PublisherIdentityPermission	An Authenticode X.509 certificate that certifies the publisher of the assembly.
SiteIdentityPermission	The host provides the server address upon which the assembly is situated (for remote code).
StrongNameIdentityPermission	The strong name of the assembly.
URLIdentityPermission	Similar to SiteIdentityPermission, but requires the complete URL of the assembly.
ZoneIdentityPermission	The security zone in which the site of the assembly is categorised (taken from the MS Internet Explorer security settings).

Table 3.4.3 - The .NET Identity permissions and the evidence needed for them to be granted.

3.10.2 Role-based security

While code access security determines whether the code is trusted to access a resource, it does not know about the user of the application. This is the job of the role-based security system: to allow access to resources only to trusted users. As with code, users are collected into categories, or *roles*. These will vary depending on the computing system, but a user may also have an individual identity. For example, in a development environment the roles could include: designers, programmers, artists, testers and so on. To continue this example, one of the designers may have an individual identity as the project lead. When the code has to decide which resources it can

access, it will look at the role/identity of the user. Each user will therefore have different rights within the application according to their role and/or identity. This allows the same application to be used for different purposes according to who is using it (e.g., reviewing files and modifying them).

The role and identity of a user is represented in the framework as a *principal* object and an *identity* object, which is referenced by the principal object. There are three kinds of principal in the framework: generic principals, windows principals and custom principals. In a system running a Microsoft Windows NT/2000 system, the login groups and users can be used directly as roles and identities. These are the windows principals. The generic principals are not tied to the operating system. These would perhaps be best used for the above example, as not all systems in such an environment will be compatible (i.e., not running on NT). As .NET is ported to more operating systems, generic principals become more preferential. Custom principals are used on a per-application basis, and allow a developer to create several specific roles that define the application's users. For example, if the testers are split into several groups to test different aspects of the software they could have an helper application that has custom roles for each of these groups. The types of identity are the same as the types of principal. The principal object contains the identity object and the role(s) the user is in. The identity object usually contains only: the name of the user, the type of authentication that was used, and a Boolean to represent whether the user was authenticated. The type of authentication is in the form of a string and is passed to the identity by the authentication provider (in the .NET plan this would ideally be Passport or Windows and so on). Again, custom identities can be created if needed. Custom principals and identities are created by implementing IPrincipal or IIdentity.

Where there are many different permissions in code access security, in role-based security there is only one. This permission, System.Security. Permissions.PrincipalPermission, compares the current principal (as provided by the runtime) with the specified principal. If they match, the permission is granted. As with other permissions, an instance of PrincipalPermission is instantiated and its Demand method is called before the code that is to be protected. It is in the constructor that the roles and/or identities that are to be permitted are specified. The constructor takes the following arguments: a string to match to the user name in the identity, a string representing the role of the principal and (optionally) a Boolean specifying whether the identity is authenticated. All of these must match the current principal for permission to be granted, but a null value can be passed in place of one or both of the strings which acts as a wild card and disables checking of that parameter. It is also possible to just call the IsInRole method of the current principal object to perform a basic check without using the security system. The current principal object is attached to the current thread, and can be accessed as the property Thread.CurrentPrincipal. The following example demonstrates the usage of a role-based security check:

```
PrincipalPermission pp = new PrincipalPermission(
                              "Paul", "Author", true);
```

```
pp.Demand();
// The code following from here on is only executed // if the
current principal matches the specified
// details.
```

The above code snippet shows a security permission demand where the principal must be in the role “Author”, and the principal’s identity must have a name of “Paul” and must be authorised. As stated above, replacing the first string with `null` will grant permission to all authorised members of the “Author” role and so on.

3.10.3 Security Policy

There are several levels on which the security policy operates: enterprise, machine, user and application domain. The security policy that is set for the enterprise is the default, unless it is overridden by the machine policy, which can be overridden by the user security policy. The lower levels can impose stricter policies than the higher-levels, but cannot ease restrictions set at a higher-level. The level for the application domain policy is more specialised and allows extra rules to be set within each application domain, but this is more the purview of the application developers than the system administrator and, again, permissions can not be allowed that are denied by the three standard policy levels. A piece of code on the system can be assigned permission sets for each of the security policy levels. For example, it may be allowed every permission on an enterprise level, but on some machines (such as a student computer cluster) may not be allowed file IO permissions.

This would quickly become an administrator’s worst nightmare if every permission had to be assigned to every piece of code on the system, so both permissions and code are assigned to logical groupings. These groupings are similar to (and presumably based on) the zone settings in MS Internet Explorer. An ironic side effect of this system means there is a good chance that IE will be a vital and inseparable part of the next generation of the windows operating system, which would be an interesting twist to the current Microsoft antitrust case. The default permission sets offer a range from nothing to full trust, where more permissions are allowed as the level of trust increases. The permission sets are: Nothing, Execution, Internet, LocalIntranet, Everything, FullTrust. These are fairly self explanatory, but suffice to say the more trusted the code group is, the higher the permission set can be allowed. In addition to these default permission sets, custom ones can be created by the security policy administrator if needed.

In a similar manner, code is divided up into groups, and each of these groups are assigned a permissible permission set for each policy level. To continue the comparison, the default code groups are an extension of the web site groups, or zones, in IE: Local, Intranet, Trusted sites, Internet, Restricted sites and All Code. This makes sense as .NET is largely targeted at the development of distributed applications so the security system is just being implemented at a lower level. As with the permission sets, the administrator can set up custom code groups. If the system administrator had to put all the code into groups it would be very inefficient and if the

code did it the security system would be compromised, so it falls to the runtime to put code into groups. This is done by the runtime examining the evidence provided by the code which differs according to the groups. For example, the URL of the code is needed as evidence for the trusted site group. The groups are arranged in a hierarchy and the code can belong to more than one group.

3.10.4 Code security in Java

It should be noted that Java's security features have been changed greatly since it was first released. In its original form, local code (applications) had full access to the system and remote code (applets) were executed in a "sandbox" and could not use any of the local resources. To allow access to the local resources for applets the concept of certified applets were introduced (roughly equivalent to the PublisherIdentityPermission in .NET). As a standard, it will be assumed that the base version of Java in use is 1.2. This has a more comparable feature set to .NET, introducing security policies to Java. It should also be noted that although both .NET and Java support cryptography features, they are not discussed here (chiefly as in Java a separate cryptography extension, the JCE is needed).

In Java, the security policy defines *domains*, into which code is grouped. The set of permissions allowed for each domain is then set by an administrator. This allows domains that cover everything from full access to the sandbox. This is somewhat like the code grouping in .NET's security policy. This is illustrated in Figure 3.4.1. There is much less flexibility for apportioning code into these domains than in .NET, as there are only two ways to group code. The evidence that allows the runtime to assign code to a domain is either the codebase of the code (e.g., the URL) or the signature, or both. The codebase is an absolute path/URL but a wildcard can be specified to include it's sub-folders. Within the security policies "*" indicates all files in that directory and "-" indicates all files in that directory and all its sub-directories. The signature is name of a locally stored certificate that contains the public key to match the private key of the applet. The policy configuration is more simplistic than that in .NET, with policies stored as ASCII text files. A policy editing tool is again provided, but is not vital (or integrated) as in the case of .NET. The policy editing tool is shown in Figure 3.4.2 with one domain present.

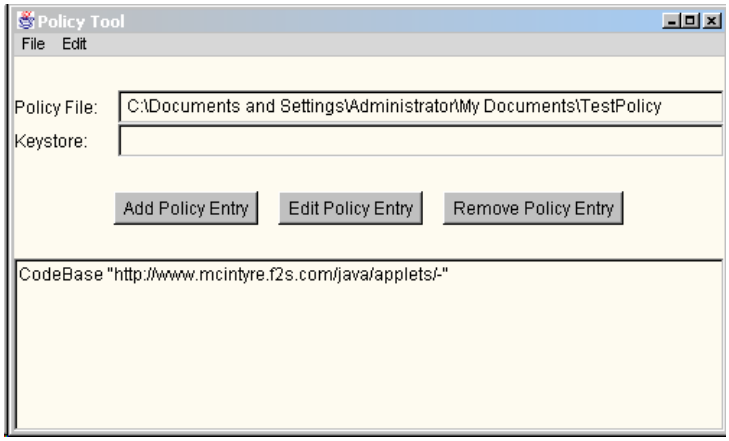


Figure 3.4.2 - An example Java security policy file.
This shows the Java policy tool being used to edit a simple security policy.

The permissions are assigned to each of these domains individually, as shown in Figure 3.4.3. Some of these permissions represent several different resources which need to be specified when granting the permission. The permission types in Java are described in Table 3.4.4. Assigning individual permissions in this way can be complicated and time consuming, as all have to be specified (unlike .NET where the runtime can automatically assign code to domains) putting undue stress on the administrator for a fully protected system. This may result in unduly excessive permissions being granted in some cases with resultant security leaks.

java.security.AllPermission	Gives unrestricted access to resources.
java.awt.AWTPermission	Gives access to windowing resources such as the clipboard and the event queue.
java.io.FilePermission	Gives access to files and directories. The type of access to be allowed must be specified (see Figure 3.4.3). This can be granted to all files or just the specified file/path.
java.net.NetPermission	Gives access to some resources for networking.
java.util.PropertyPermission	Allows code to read and/or write system properties.
java.lang.reflect. ReflectPermission	Allows access to all members of reflected objects, regardless of their visibility.
java.lang.RuntimePermission	Allows access to resources of the runtime itself. Some of these permissions are very dangerous to grant to *any* code.
java.security.SecurityPermissio n	Allows access to resources within the security system. These are dangerous permissions to grant.
java.io.SerializablePermission	Allows code to customise serialisation classes.
java.net.SocketPermission	Allows code to connect to the specified host.

Table 3.4.4 - Permission types in Java.

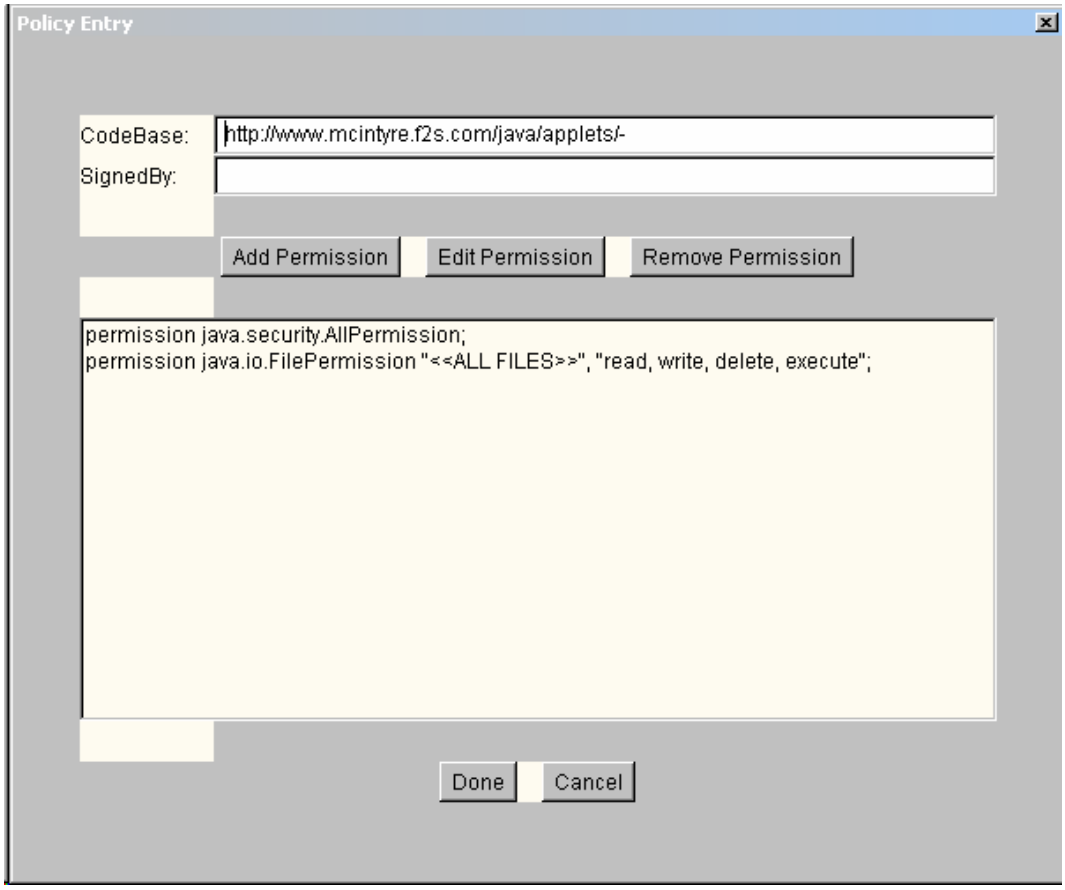


Figure 3.4.3 - Adding permissions to a Java security policy.

There are two ways of applying a policy file (as created in Figure 3.4.2 above). The first is to pass it as a parameter into AppletViewer. This allows applet-specific policies to be created, but this also limits the availability of the policy to applets called in this way. For the policy to be automatically available (such as when encountering an applet during browsing) the policy must be added to the system's policy list (the file java.security in the runtime's lib/security directory). The policy file created and shown in Figures 3.4.2 and 3.4.3 above is listed below. It is possible to create policy files without using the utility, so it is also possible for any user with access to the policy files (or the policy list) to alter the system policies. This is an additional risk that has to be blocked manually by an administrator, whereas in .NET this is largely taken care of as the management console is generally only available to administrators on NT systems.

```
/* AUTOMATICALLY GENERATED ON Tue Mar 05 16:27:59 GMT 2002*/
/* DO NOT EDIT */
grant codeBase "http://www.mcintyre.f2s.com/java/applets/-" {
  permission java.security.AllPermission;
  permission java.io.FilePermission "<<ALL FILES>>", "read,
            write, delete, execute";
};
```

The above policy file snippet specifically applies to applets, as applications run by default without any security checking. To execute an application with a security manager, it must be specified as a command-line parameter. For example, to run a program called applicationName: "java -Djava.security.manager *applicationName*". This need to use a command line variable also allows security vulnerabilities, as by default any downloaded code is run without restrictions. It also prevents a developer from securing their own code, as seen with the .NET identity permissions where code in an assembly could be made to run only when called by code with the correct certificate, for example. The security system in Java seems to be an afterthought, where in .NET it is one of its central parts.

3.11 Garbage Collection

One fundamental change that .NET presents to the traditional C family of languages and windows programming is a feature that initially gained a lot of popularity for Java: the garbage collector. The garbage collection system cleans up resources when they are no longer used which prevents software bugs such as memory leaks. While it will eventually collect and clean up resources that are no longer used, it is more inefficient than de-allocating them as soon as they are no longer needed so is a safety feature more than a convenience feature. Several aspects of the collection system are discussed here to describe how it functions. To identify garbage, the collector needs to be able to identify which objects are in use by the program. It uses the *roots* of the application to do this, described below.

3.11.1 Roots

The roots of an application are all of the object references that are currently visible to it. Consider a tree diagram: there are various roots of the tree from which branches split off, which may also have sub-branches and so on. An analogy would be a class inheritance hierarchy. The garbage collection system identifies "live" objects by studying the object reference hierarchy. To do this, it looks at the roots (i.e., all those objects that are currently visible to it) and examines each in turn to see what objects it references and so on to trace though each node and build up a graph of all the objects in the tree. If it encounters an object in a branch that is already on the graph (i.e., two branches merge) then it can skip it as it already found all its referenced objects. All of the objects in this graph can be reached by the application. There is no way for the application to reach any of the objects that do not appear in this graph, so they are "dead" objects. After tracing the roots, the garbage collector can then compare the graph with the heap and delete any objects that are garbage. It cannot delete any object which is on the graph.

The following (adapted from Richter, November 2000) is a list of an application's roots:

- Global object references.
- Static object references.
- Local variables on a thread's stack.
- Parameter references on a thread's stack .

- CPU registers containing object references.
- Objects referenced in the *freachable* table (see the section on finalization for details).

All of these (aside from the last one which is a special case) can be summed up to mean all objects visible to the application. The first two are merely those variables that are always visible to the application. The next two are the variables visible to the currently executing method (including the parameters passed in to it) and those visible to each of the methods which will resume execution as each previous method returns (i.e., each method in the execution stack). As there is a different execution stack for each thread then every thread's stack is a root. The last of the standard roots, the CPU registers, contains the objects that are currently being accessed by the CPU. Through these roots, all of the objects that can be accessed by the application are visible. By tracing through each of these objects, all of the objects that might be visible to the application can be found. If an object exists outside these roots, there is no way for the application to find or access it, so it is garbage. The runtime tracks the active roots of the application and the garbage collection system can then access these to trace through them and find all the live objects in the application, as described above.

3.11.2 Strong and weak references

If an application fills the heap with objects that are *all* live, it runs out of memory and no more can be created. For applications that need to use a lot of memory, the collection system can let the application specify live objects can be deleted if it needs the memory. These delete-able objects can then stay in the stack as normal, but if the collector cannot free enough memory to create a new object it will delete as many of these as it needs to until there is enough space. To accomplish this, .NET provides two different ways to reference objects: strong and weak references. Strong references are the standard reference type, as used when declaring an instance of an object or a variable, for example. By default every object reference is strong, so all references found through the roots are strong. Weak references are those that allow the collector to delete the referenced object if necessary. A weak reference is created simply by creating an instance of the type `System.WeakReference`. This has a `Target` property that points to the object. As mentioned previously all objects reachable through the roots of an application are considered live, so to allow weakly referenced objects to be deleted the garbage collector can not trace through a WeakReference object to its target object. If an object is only referenced through a weak reference, it is therefore automatically considered to be garbage.

To prevent the collector deleting weakly referenced objects when not needed, there needs to be some way for it to know they are not automatically garbage, despite not being on the graph of live objects. This is done courtesy of two tables stored inside the runtime. These tables are a list of the addresses of the objects on the heap which are referenced by weak references. It is the entry in one of these tables that is returned by the

WeakReference instance's Target property. There are two tables for the two types of weak reference: short and long. Short weak references are that standard. The only difference between the two types is that long weak references keep tracking an object if it is reincarnated, whereas short weak references do not. Reincarnation can result in the corruption of the object so it, and by extension long weak references, are not widely used.

If an object referenced by a weak reference is collected, its entry in the appropriate table is set to null. When (and if) the application wishes to access it again, the WeakReference instance can be interrogated to find if its entry on the table is still pointing at the heap. If the object is still present, the object reference returned by the target property can be assigned to a strong reference to allow it to be used again (for an example see Table 3.5.1). This must be done as once a strong reference exists the object cannot be collected, otherwise there is a chance of the object being collected whilst it is still being used. If the object has been collected while it has been weakly referenced, the application must replace it. For this reason weakly referenced objects should not contain information that cannot be recreated if necessary.

Member	**Description**
Constructor one argument: object	Initialises the weak reference and sets the target to the specified object. Equivalent to calling the other constructor with *false* as the second parameter.
Constructor two arguments: object and Boolean	As above, but the second parameter allows the specification of the TracksResurrection parameter.
IsAlive	This is a Boolean property that returns false if the target object of the WeakReference instance has been collected by the garbage collector.

Target	This is a weak reference to the object specified in the constructor. To use the specified object, provided it has not been collected, this object must be assigned to a standard (strong) reference, such as a variable. For example, if a weak reference instance named "temp" references an instance of type "Item" then it can again be used in the application by the following:
	`Item notTemp;`
	`if(temp.IsAlive) notTemp = temp.Target;`
	If the object referenced by this parameter has been collected, this parameter's value is null, so in the above code there is no real need for the *if* statement unless partnered by an *else* that handles the loss of the object (for example, by re-reading a file to recreate it).

TracksResurrection	This is a Boolean value indicating whether this instance tracks the target object after resurrection. If this value is true, the instance is a *long weak reference*. If it is false (which is usual as it is much safer) then it is a *short weak*

	reference.

Table 3.5.1 - The .NET type System.WeakReference.
The WeakReference type is a simple type as demonstrated here as it is a reference object.

3.11.3 Generations

If the entire heap were collected each time, the objects that have long lifetimes would go through the collection procedure every time. This is not efficient, so the garbage collector needs some way of separating out the objects in the heap. To do this it uses *generations*. As the name implies, this is done by categorising the objects by their place in the lifecycle of the application. This uses the assumption that those objects that are created earlier in the application have longer lifecycles, whereas those that were created recently have shorter lifecycles. While there will always be exceptions, in general this rule holds true. For example, when a method creates a variable which is local to that method, it is then very recent and will usually have a short lifecycle. However, if that method is at the top of a chain of called methods, then compared to another variable in a method at the bottom of the chain, the original variable object is older and has a correspondingly longer lifetime. Another assumption is made that variables of the same generation are often interrelated. For example, two objects created in the same method are often used together. With the managed heap new objects are placed consecutively on the heap and the objects of a generation are all concurrent, so they will be close together on the heap as well which improves performance.

There are three generations in the garbage collection system (currently, see the description of MaxGeneration in Table 3.5.2 for more details) and these are numbered. Generation 0 is the current generation (i.e., the newly created objects that have not yet entered a collection phase), 1 is the previous generation (those objects which were first collected in the previous collection cycle) and generation 2 contains objects that are relatively long-lived and have survived more than one collection. Newly created objects always go into generation 0. When the first garbage collection of the application's execution occurs all the objects in the heap are in generation 0. When a collection has been completed, the number of each generation is incremented (i.e., the new objects that still remain in generation 0 after the collection are compacted and become generation 1). There are no higher generations than generation 2 as the lifetime of objects tends to be lengthy after a certain point so objects in the highest generation are much less likely to be garbage, which would reduce the efficiency of constantly collecting them. As stated in Table 3.5.2, Microsoft may change the maximum number of generations if it is found to increase the efficiency of the garbage collector.

This system allows the performance of the garbage collector to be improved as generation 0 can be collected whenever more memory is needed, and that will normally free up the space as the newer objects tend to be shorter lived (hence most of the garbage will normally be in generation 0). As only one generation is being collected, there is much less of a performance hit than having to check all three generations. If more memory needs to be found, the collector just progressively checks more of the generations and accepts

the increased performance hit as being necessary. When performing a quick (one generation) collection, objects in older generations can therefore also be ignored by the collector as it checks the roots unless the object has been recently modified and may contain references to newer objects. This further increases the efficiency of the collection system.

3.11.4 Finalization

There are some types that have resources which must be cleaned up before being collected. The collector can not know how to do this for a type it has never seen before, so the types that require it must do it themselves. This takes the form of the finalizer (.NET naturally uses the US spelling throughout, so this document does also). The finalizer is a method which is called before the object is collected in which code can be placed to clean up any resources used by the object. This method is added to a class by overriding the `Object.Finalize` method. This is usually the only contact a developer will have with the garbage collection system while writing an application. It is also potentially the greatest performance bottleneck. This is largely due to the collector having to call each of the finalizer methods before it can clear the objects out of memory. If it were required to do this before the collection cycle could finish, it would cause a delay in the execution of the application, which would be unacceptable. This is because the threads within an application must be suspended for a collection to take place (as objects are being moved around). To this end, objects with finalizers are not deleted in the collection, but are temporarily resurrected, while a separate thread in the runtime executes the finalizer methods. Once this thread has executed the finalizer, the object then becomes garbage once again and is removed in the next collection.

There are two main problems that result from this. The first is that as stated before, each finalizer must be executed, which can amount to a significant extra load. The second problem is that a garbage object with a finalizer therefore takes at least two collection cycles to be removed from memory (it could be more if there are a lot of objects to be finalized as in the previous case). When the finalizer is executed, it is possible for it to permanently resurrect the object (for example, by assigning the *this* reference to a global variable). This returns the object to the roots, and the application, but as the object has been finalized it is very likely to cause bugs and is therefore only used rarely. For these reasons, it is important only to use finalizer methods when absolutely necessary (i.e., only where there are resources within the object that need to be specifically closed).

The garbage collector keeps track of object finalizers using two tables: the *finalization* table and the *freachable* table. The finalization table is a list of references to all the objects present in the heap that contain the Finalizer method. When an object is added to the heap, if it has a finalizer it is added to the table. When one of these objects becomes garbage the collector checks the finalization table to see if it needs to be finalized. When it finds an object on this table, instead of collecting it, it moves the reference from the finalization table to the freachable table. As the freachable table is a root of the application (see above) the object is therefore temporarily resurrected (hence the name: the finalization requires the object to be reachable). The

previously mentioned thread that is used to run finalizers is activated when that finalization cycle is complete and the application restarts. It runs in the background and calls each of the objects' finalizers in turn. When a finalizer has been executed, the object is removed from the freachable queue. As no reference to the object now exists in the roots or the finalization table, it will then be removed from the heap by the next collection cycle.

The garbage collector controls when the finalizer is executed, so this is not a good way to clean up any resources which may be needed promptly by the application. To this end, many of the framework types have a Close or Dispose method which is explicitly called by the application to trigger an immediate cleanup of the type instance's resources. This also avoids the other problems associated with finalizers. The only problem with these methods is that if they are not called the resources are not cleaned up. However, to get the best of both ways these methods can be combined with a finalizer. This is made possible because the type `System.GC`, which is used to control the garbage collector, has methods that enable an object to be added and removed from the finalization table (see Table 3.5.2). This allows a Close method to remove the object from the finalization table so when it becomes garbage its finalizer will not be executed, which avoids all the aforementioned performance problems. If the type allows the resources to be reassigned (i.e., an Open method to reverse a Close method) then that method can add the object back on to the finalization list. This allows the increased efficiency, without the problems associated with forgetting to call an explicit close method. For a more thorough discussion of this, and the problems associated with finalizers, see Richter, December 2000.

Member	**Description**
MaxGeneration	The maximum number of generations used is not yet fixed. This property is therefore used to find out the current value. Generations are numbered sequentially, with the newest always being zero, rising to a maximum for the oldest supported generation. At the present time there are generations 0, 1 and 2 available to the collector so this property returns 2.

Collect	This method tells the runtime to perform an immediate collection. An optional integer parameter allows the highest generation to collect to be specified. For example, `Collect(0)` performs a collection of the newest generation only, and is the equivalent of calling this method without a parameter. Calling `Collect(2)` will collect all generations, although using the above `MaxGeneration` instead safeguards against the number of generations being amended at some time in the future.
GetGeneration	This method returns the number of the generation that the object specified as the parameter is presently in on the heap. A weak reference instance can be passed to this object instead, in which case the object it points to (its Target object) is found.
GetTotalMemory	This method returns the number of bytes currently allocated on the heap. If the parameter passed to this method is true, a full collection is performed before doing this (i.e., `Collect(GC.MaxGeneration)` is called).
KeepAlive	The collector does not consider object references in unmanaged code when determining whether an object is still in use. This method keeps the object which is passed to it from being collected *before* the point it is called. This essentially places a reference to the object in the code, which makes the object reachable when tracing the roots. After the call, the collector can then collect it.
ReRegisterForFinalize	This method places the object passed as a parameter back on to the finalisation queue within the collector. If this method is called multiple times consecutively, the object will be referenced several times in the queue. As such, it will cause bugs if not strictly controlled. There are two reasons for calling this method: the object has been previously removed from the finalisation queue by the `SuppressFinalize` method, or the object has already been finalised but was resurrected (in which case it may be called as part of the resurrection process). This is a source of problems for applications as resurrection can corrupt objects.

SuppressFinalize	This is the opposite of the above method. If an object does not need to be finalised, this method marks the object passed to it as not needing finalisation. This can be reversed by calling the above method and vice-versa. It should be noted that this method does not simply remove the object from the finalisation queue so it cannot remove multiple entries if the above method is called more than once without intervening calls to this method. This method is more useful than the above as it increases the efficiency of collection for objects that do not always need to be finalised.
WaitForPendingFinalizers	There is a separate thread in the runtime for calling any pending finalisers in the *freachable* queue. This thread is only active when there are items in that queue and runs until all the finalisers have been executed. This method halts the currently executing thread until this has happened and the finalisation thread has suspended. This allows the developer to make sure all resources are properly closed before continuing.

Table 3.5.2 - The .NET type System.GC.
All members are static as it is used for controlling and monitoring the garbage collector.

3.11.5 The *new* operator and the managed heap

The heap is a section of memory that is allocated by the runtime for that process. This memory is a continuous, contiguous block. The structure of the heap, along with the four tables held in the garbage collector that are used to reference objects on the heap, is shown in Figure 3.5.1. When an object is added to the heap, it is at the location of the heap pointer, and so generation 0 is a complete, unfragmented, block. The first part of the managed memory system that an object encounters is the *new* operator. This is one of the most basic keywords and is used to create an instance of a type. It is responsible for placing objects on to the managed memory heap. The heap contains a pointer, which points to the end of the used space in the heap (as the contents are compacted during every collection the heap is not fragmented). The new operator looks at the size of the type it is creating an instance of and if there is not enough space, it triggers a garbage collection cycle to make space. It then creates an object of that type on the heap at the location of the pointer (i.e., the start of the free space) and initialises it with the constructor that follows the *new* statement. It then moves the pointer to the end of the object it has just created. The *new* operator returns a reference to the object on the heap (i.e., to the variable it is being assigned to).

When the garbage objects are collected, then each generation that was included in the collection will contain gaps where the removed objects were. All data in the memory after each gap is then copied forward to fill the gaps (as with defragmenting a hard drive, except the order of the data is

maintained). As such the data in the heap is compressed to fit in the least space possible. As a collection has just occurred, what had been generation 1 then becomes part of generation 2, generation 0 becomes generation 1, and the pointer is moved to the end of this new generation 1 in preparation for the next new objects to be created. This constant copying of data is very inefficient for large objects so there is an auxiliary heap for objects over 20,000 bytes where this copying is not performed.

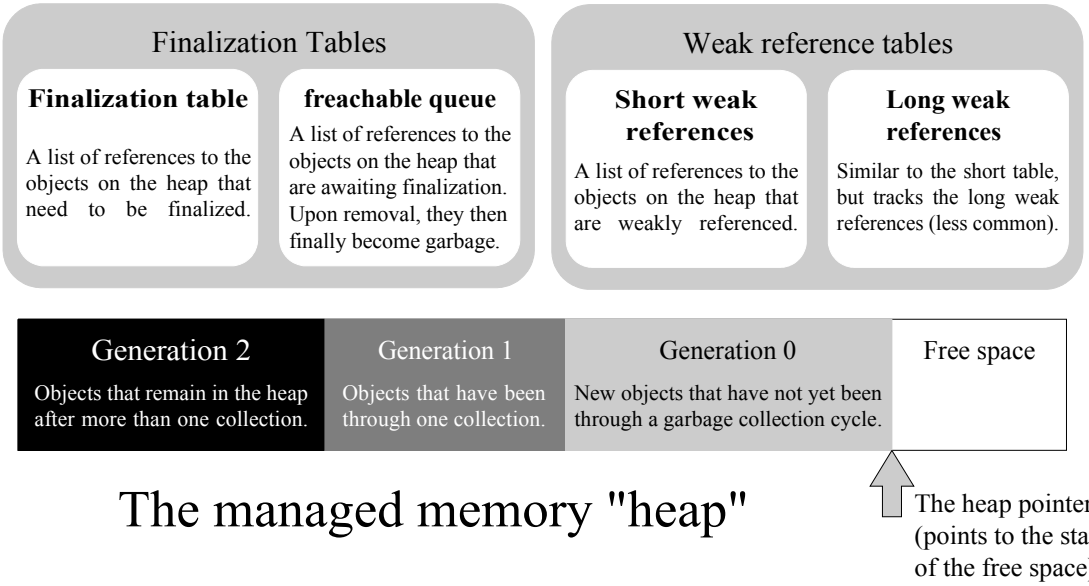


Figure 3.5.1 - The structure of the managed heap in .NET.
As well as the allocated memory space (bottom), there are the tables that are used to interface with it.

3.11.6 Collection

As stated, a collection occurs either when there is not enough room to add a new object to the heap, when one of those collections fails to free the required memory (in this case a collection of more generations), or when a collection is explicitly requested by calling the `GC.Collect` method in code. The collection cycle proceeds as follows:

- Firstly, the garbage collector examines the list of active roots in the runtime and iteratively traces through each element to build up a graph of the reachable objects.
- The collector then examines the short weak references table to find all of its elements that point to objects on the heap that are not in this graph. It then sets the value of those entries to **null**, which informs the appropriate weak reference instances that their target objects have been collected.
- The collector then finds those entries in the finalisation table reference objects that are not in the reachable graph. These entries are moved from this graph to the freachable table. This allows the finalization thread to execute the finalizers when the collection cycle completes and the application execution resumes.

- The collector then has to amend the graph of reachable objects to include those entries that were just added to the freachable table (as they have now been temporarily resurrected).
- The collector then sets the entries in the long weak reference table whose objects are not in the (now expanded) graph to null. This is the same procedure as occurred for the short weak reference table, but because this is done after the freachable table's objects were added to the graph it includes those objects that were resurrected.
- At this point, the collector now knows which of the objects in the heap can be removed, so it marks these objects as garbage. It then compresses the remaining objects by moving them forward to take up any free space.
- As the objects on the heap now have different addresses, the collector then has to go through all of the object references in its tables and the object graph and set the address pointed to by each reference to the new address of the object.
- As a final step, the collector then moves the heap pointer to the end of the used space ready for the creation of new objects. At this point the collector has completed the cycle and the application resumes its execution until the next collection cycle.

3.11.7 The Java Garbage Collection system

As stated in the introduction to this section, Java also uses garbage collection. For the developer the collector is similar in use: finalizers are created in the same way and a collection can be started by running the System.gc method. The heap in Java is dynamic, and not as rigorously ordered as the one in .NET. This makes the performance of the collector inherently better as there is no copying of objects during collection, but is less efficient than .NET's as the used space is not compacted. The collector traces through the object references in the same way, and marks the objects on the heap as live rather than in an internal table. There are no generations as the main reason for them (to compensate for the performance loss of copying objects around the memory) is not valid in Java. Another result of not moving the objects in memory is that it is possible for the JVM (on certain systems) to run in the background and not stop program execution (unless necessary to reclaim more memory).

The reference system in Java handles weak reference functionality. These types are found within the java.lang.ref namespace. The Java equivalent of the reference tables in .NET is provided by the ReferenceQueue type. It is more comprehensive than the .NET system, but also more complicated.

4 Important parts of the programming libraries

4.1.1 Introduction

In addition to the properties of the runtime, as discussed in the previous section, there are other parts of the .NET and Java programming libraries that are important to developers. The most important of these are the collection classes. Indeed, in Java it may be fair to say they are a part of the system as arrays are treated as a primitive type, however .NET's collection system is completely object-oriented and need not be used at all if an alternative is available. In addition, the serialisation and file handling types are essential for developers. The third part of this section is the reflection system. This is only dealt with in a minimum of detail as it is used relatively infrequently. The final part of this section looks at the database connectivity APIs. These are also a vital part of a programming library. The breadth of features and ease of use of these basic libraries has a major impact on the efficiency of a development team, so whichever system can implement them more effectively should theoretically be the best one to use from a developer's perspective.

4.2 Collections

The most basic form of collection is the array. The most obvious difference in the collections in .NET and Java is therefore that .NET arrays are objects whereas Java arrays are not. This makes the array integral to the collection system, whereas in Java it seems an additional construct. There is also a related difference in that primitive types in .NET are also objects (and again Java's are not, as described elsewhere) so wrapper objects are not needed when using them in collections. Array types in .NET derive from the abstract superclass `System.Array`. When an array object is created for a primitive type, as normal, it is an instance of the appropriate subclass of Array that is created. For any type that does not have a dedicated Array subclass a generic `System.Object` array type is used. As these .NET arrays are objects, methods can be called on them as with the other collection types.

The drawback to arrays in .NET is in the multiple language functionality (see section 6). The Array type supports arrays based at any index (for example arrays can be zero-based, one-based, … , n-based). This can be a useful feature when matching up an individual array for a specific purpose (i.e., for code readability in for loops), but can cause problems when arrays are used by different modules. Particularly, the support for arrays in visual basic straddles zero-based and one-based arrays (arrays in code ported from previous versions are increased in length by one and include a null element at zero). If the details of arrays that are going to be used are not known, then they need to be tested, which adds unnecessary complexity.

The collection base types and interfaces in .NET are situated in the `System.Collections` namespace, except for the basic array types discussed

above, which are with the primitive types in the System namespace. There are additional (sub-type) collection classes existing elsewhere, but these are the base types that are used. Figure 4.1.1 shows a hierarchy of these classes. The generalisation relations in Figure 4.1.1 represent the implementation of the interfaces. The structure, DictionaryEntry, is a key-value pair for use by the dictionary types (i.e., those that implement IDictionary). These dictionary types are the map types. The list types are indexed lists, including the array types mentioned above and these types implement the Ilist interface. Any collection types that do not need either of these feature sets can directly implement ICollection (for example, the Stack type which is a simple last in, first out buffer). As all collection types ultimately implement the IEnumerable interface, all of the collection types have an associated enumerator.

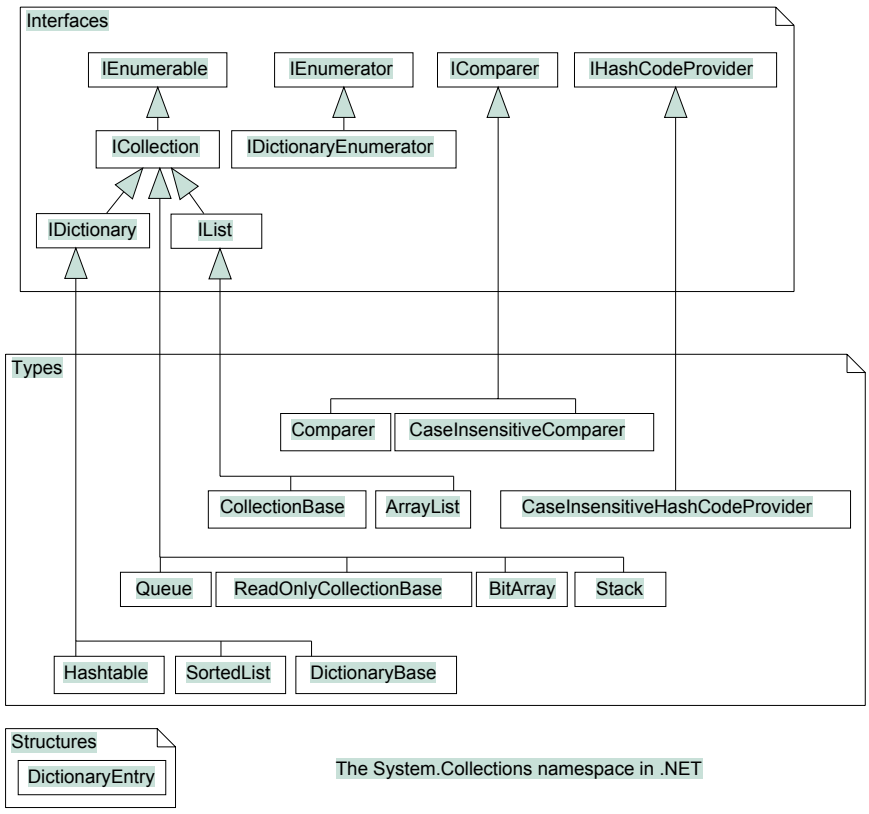


Figure 4.1.3 - Contents of the .NET framework namespace System.Collections.

The collection framework in Java resides in the java.util namespace. The types are shown in Figure 4.1.2 below. The generalisation arrows from the type classes to the interface classes represent implementation relations, while within each group they are extensions. The first comparison to be made is that there are more classes in java.util, and that this package contains many other classes (such as the assorted date types). The .NET collection types are grouped together, which makes them easier to use. As is shown in Figure 4.1.2, and unlike .NET, not all of the collection classes implement the Collection interface. .NET does not have sets as a separate

category of collection, although a list type could be used or a dictionary type could be substituted. The Map.Entry interface fills a similar role to the DictionaryEntry in .NET. The classes Arrays and Collections are utility types containing static methods for manipulating the appropriate types. The Arrays type is needed to try and provide some of the functionality that is missing from arrays in Java as they are not objects.

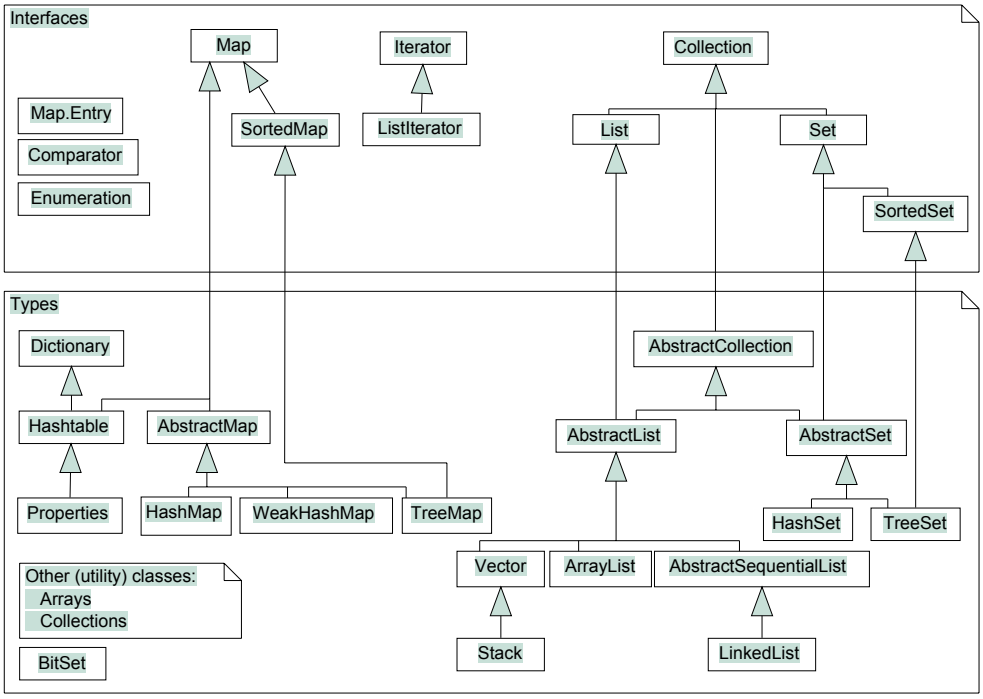


Figure 4.1.4 - The collection classes in Java.

4.3 Serialisation and File I/O

Serialisation is the conversion of objects to a byte stream (i.e., to save into a file) and back again that allows the state of objects to be stored. This allows object members, objects and webs of objects to be saved to a file and restored later on. In both systems, attributes are used to specify what is serialised. Serialisable objects can then be passed into an output stream and serialised. In .NET, the attribute `SerializableAttribute` specifies a class as serialisable, while the attribute `NonSerializedAttribute` specifies members that should not be serialised. In Java, a class must implement the `Java.IO.Serializable` interface while members marked `transient` are not serialised. In both systems, classes can control their own serialisation by overriding methods in the Serializable interface (`ISerializable` must be implemented in .NET).

4.3.1 .NET

The namespace `System.Runtime.Serialization` and its sub-namespaces contain the classes that are used to provide serialisation functionality. For standard binary object serialisation a `BinaryFormatter` object is created. This

formats objects into binary streams and vice-versa using Serialize and Deserialize methods (shown below).

```
BinaryFormatter bf = new BinaryFormatter();

FileStream in = new
FileStream("test.sas",FileMode.Open,FileAccess.Read);

FileStream out = new
FileStream("test.sas",FileMode.Create,FileAccess.Write);

bf.Serialize(out,artSys);

artSys = (ArticleSystem) bf.Deserialize(in);
```

In this example, the object artSys, of type *ArticleSystem*, is serialised in the file "test.sas". The FileStreams created are simple dedicated read / write streams. To serialise the object the output stream and the object must be passed to the formatter (the object and any referenced objects must be defined with `SerializableAttribute`). Serialised objects are stored as standard `System.Object` objects so they must be cast on being returned. Any objects that are referenced by the object will also be stored, and upon de-serialisation will also be de-serialised.

Java

Serialisation is handled in much the same way as with .NET. The appropriate utility classes are contained within `java.io`. In Java, there are dedicated object streams that are used directly for the serialisation of objects. Serialised objects are stored as generic `Object` types, in the same manner as in .NET.

```
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("test.sas"));

ObjectInputStream in = new ObjectInputStream(new
FileInputStream("test.sas"));

out.writeObject(artSys);

artSys = (ArticleSystem) in.readObject();
```

As this example shows, the object streams wrap the file streams to format them (as opposed to .NET where the formatter accepts the file stream as an argument) and so separate ones need to be created for input and output and for each file. In use, the serialisation syntax in Java is similar to that of .NET and the only real differences are the use of a standalone formatter in .NET and in the file streams.

File streams

FileStreams are only one of five types of stream that can be used with a formatter in .NET, the others are: BufferedStream, MemoryStream, NetworkStream and CryptoStream. The buffered version wraps around another stream to buffer it and the CryptoStream is for encrypted streams,

while the memory version uses memory as the store instead of disk space and the network variant sends and receives through network sockets. The streams in .NET are all byte streams, and can be opened for reading, writing or both - the `FileAccess` enumeration that is a parameter in the FileStream constructor above can have a value of Read, ReadWrite or Write. The constants in the `FileMode` enumeration specify how the file is opened (i.e., Append, Create etc.). The `FileShare` enumeration (not shown in the above example) allows the programmer to specify how the file can be shared by other file streams once it has been opened. In Java, there are a greater quantity of stream types, which are either dedicated input or output streams. The byte streams are shown in Figure 4.2.1, and there are also dedicated character streams ("readers" and "writers"), as in .NET, for dealing with text.

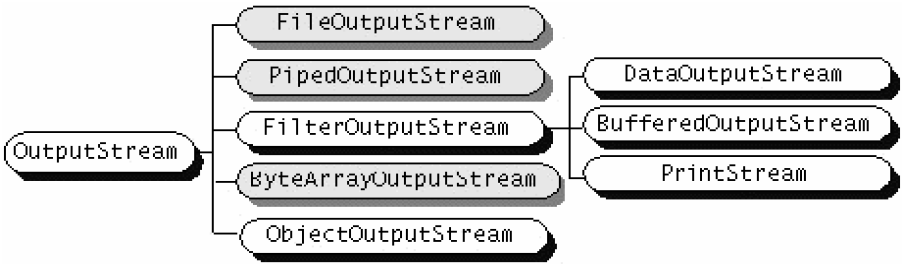


Figure 4.2.1 - The byte streams in Java (Campione & Walrath, 1998)

Entries in grey are the ones that read/write to/from a data source, whereas the white ones perform processing on another stream (see the previous example of ObjectInputStream and FileInputStream.

Formatters

In Java the stream formatters are either built in to some of the byte streams or they wrap them (the white entries in Figure 4.2.1), whereas in .NET the formatters are separate entities as seen in the code excerpts above. The supplied serialisation formatters are found within the `System.Runtime.Serialization.Formatters` namespace and implement the `IFormatters` interface. .NET comes with two types of serialisation formatter. The first is `BinaryFormatter`, which is the standard formatter for serialisation and is like that used in Java. This serialises to and from binary and allows objects to be saved to disk as files, for example. The second is `SoapFormatter` which has no equivalent in Java, which serialises into the industry standard SOAP format, which is discussed by Gunton (2001). This is used mainly for transporting objects across the Internet. For example, it can be used for de-serialising data returned by a web service.

4.4 Reflection

The core class of .NET's reflection services is `System.Type`. This is the class that is used to represent and interrogate .NET types. Type includes several different methods for getting the type of an object, depending on its location. These are static as they are used to return Type instances; the Type instances are created from types, not assigned to them. The base Object class also includes a GetType method that returns the Type instance for an object. The instance members of this class are too numerous to fully

describe here, but can be summarised as follows: the properties of the Type object represent the properties of the *object* (not the property members defined within it) such as whether it is serializable, and the methods allow information about the internal structure and contents of the object to be found. The Type instance does this by accessing the type's metadata.

The reflection system also includes the types `System.Reflection.Assembly` and `System.Reflection.Module`, which represent and interrogate assemblies and modules in much the same way. A module is merely an individual file within an assembly (i.e., a DLL file or an executable). While application assemblies frequently consist of numerous modules, some (such as a DLL to be installed in the global assembly cache - see section on assemblies for details) consist of only one. For example, these individual files can be identified via the Assembly type and the Module type returns a module's filename and path.

Java also features reflection services. The Java equivalent to .NET's Type class is `java.lang.Class`. The Java Object type features a `getClass` method which functions in the same way as .NET's GetType method. There are four types in the `java.lang.reflect` namespace that are used to provide the rest of the functionality of .NET's Type class: Field, Method, Constructor and Modifier. These types can be used to get more information on each of the appropriate parts of the type. Again, these are too complex to fully describe here. It can be noted, however, that the Method class is the nearest Java equivalent of delegates (although only used for this purpose in Java). One important difference between the reflection systems is that the Java reflection system cannot analyse JAR files in the way .NET can analyse assemblies and modules, which limits the usefulness with released packages.

4.5 Database connectivity

4.5.1 ADO.NET

ADO.NET (ActiveX Data Objects.NET) is .NET's mechanism for providing access to data sources. It abstracts away the differences in data sources and their access methods in much the same way as the .NET framework itself does with the Internet. It is essentially an attempt to update ADO for .NET, and tries to fix the problems that ADO had. Data providers wishing to support access to their data sources via ADO.NET can implement an interface that is supplied. For backwards compatibility, ADO.NET includes interface classes that allow access to OLE DB compatible data sources (which includes ADO ones). This system for accessing data sources is simple in use, and relies on four parts, the connection to the data source, the commands that can be issued to it, the adapter to connect the code to the data source, and a store for any data received. These four components are described here, followed by an excerpted example. All of the types mentioned here belong to the `System.Data` namespace (and where specified, the sub-domains such as `System.Data.OleDb`), which contains the ADO.NET types.

A connection to a data source is opened by creating a *Connection* object. There are two types of connection object in the .NET framework (both function in the same way): a generic OLE DB connection object (of type `OleDb.OleDbConnection`), and an SQL (Standard Query Language) optimised connection object (of type `SqlClient.SqlConnection`). Microsoft are keen on users using this optimised version (for example, at present the help files are more complete and it was stable while Microsoft were still altering the generic version) and it is more efficient and, unsurprisingly, can only be used with a Microsoft SQL server. This connection is performed in a similar manner to ADO: when initialising the connection object, a connection string is passed to the constructor that contains any needed values (see the SDK help files for details). These properties can be subsequently changed, but only when the connection is closed. A transaction can also be created for data sources that are likely to be updated often. The connection should be closed manually as soon as the program no longer needs it as it will not be closed automatically upon going out of scope. The garbage collector will get around to it, but this is inefficient.

To issue commands to a data source (i.e., to select from it or update it), a *Command* object is needed. As with the above, there are two types of command object (`OleDb.OleDbCommand` and `SqlClient.SqlCommand`). The command object is assigned to the appropriate property of the data adapter object (see below), which uses this if it is called upon to make a change to the data source. The command object features three methods that can be used to command the data source: `ExecuteReader` which should be used for commands that return data from the source, `ExecuteNonQuery` which should be used for commands that perform actions on the data source and do not return data and `ExecuteScalar` which should be used for commands that only return one value from a data source. The Microsoft SQL specific version, SqlCommand, has an additional method: `ExecuteXmlReader`. This is similar to ExecuteReader, but returns the data in XML format.

A *DataAdapter* object is used to request information from a data source through an open connection (as a writer is used to write to an open stream, for example). As with the connection objects, there are two types of data adapter (similarly `OleDb.OleDbDataAdapter` and `SqlClient.SqlDataAdapter`). The information request (i.e., an SQL query) and the connection that the request is to be made from are passed to this object by the developer. Command objects can be created and assigned to one of four properties in the DataAdapter object: SelectCommand, UpdateCommand, InsetCommand or DeleteCommand. These commands are then used by the DataAdapter object when it is required to carry out that action on the data source.

Data is received from a data adapter request as a *DataSet* object (of type `DataSet`). DataSets are containers for data. They can be either typed or generic, and hold data in tables. The data adapter object has a method which fills an empty DataSet. The DataSet type contains a method *GetChanges* that returns a dataset which only contains rows where changes have been made, which enables only the changes to be written back to the data source.

```
private System.Data.SqlClient.SqlConnection sqlConnection1;
private System.Data.SqlClient.SqlCommand sqlCommand1;

private System.Data.SqlClient.SqlDataAdapter sqlDataAdapter1;
private System.Data.DataSet dataSet1;

sqlConnection1 = new System.Data.SqlClient.SqlConnection();
sqlCommand1 = new System.Data.SqlClient.SqlCommand();
sqlDataAdapter1 = new System.Data.SqlClient.SqlDataAdapter();
dataSet1 = new System.Data.DataSet();

sqlConnection1.ConnectionString = "data source=MCKAL;" +
  "initial catalog=Northwind;integrated security=SSPI;" +
  "persist security info=False;workstation id=MCKAL;" +
  "packet size=4096";

sqlCommand1.Connection = this.sqlConnection1;
sqlDataAdapter1.SelectCommand = this.sqlCommand1;

StreamReader sr = new StreamReader(filename);

sqlCommand1.CommandText = sr.ReadToEnd();

sqlDataAdapter1.Fill(dataSet1);

dataGrid1.SetDataBinding(dataSet1,"Table");
```

This example demonstrates the simplest type of connection to a data source: connecting to a local database (in this case just the sql server example database Northwind) and putting the result of a query into a DataSet. The first block is just the initialisation of the appropriate objects, and demonstrates that all four of the discussed types use default constructors. Following that is the assignation of the connection object's connection string property (the string has been broken to fit onto multiple lines). Here the local sql server is MCKAL, which is also the name of the workstation. The connection string is quite clear as the token names are descriptive and value pairs are separated by semicolons. This connection object is then assigned as the connection to be used by the sql command object. The command object is then assigned to the data adapter's select command parameter. This will then be used automatically by the data adapter as the select statement. If the application wanted to update the data source then command objects representing the update. Insert and delete statements would be assigned as well. This means that different connections can be used for each of these commands. For example, this would enable an unchanged database for selecting and any inserted rows could be sent to a buffer database. This would be transparent to code using the data adapter. The next piece of code just loads an sql query from a text file (located at *filename*) and assigns it as the select command objects command text. This is the actual query that is executed on the data source. The final piece of code in the example demonstrates the usage of the data adapter. It executes the select statement by filling the DataSet object. While it is extraneous to this example the final line shows the usage of the DataSet object. The data grid is a windows forms visual component. That command is all that is needed to automatically display the table that was returned in the DataSet in the grid.

4.5.2 JDBC (Java Database Connectivity)

Java uses JDBC to access data sources. There is an updated version of this standard with new features known as JDBC2, but as with other recent additions to Java that may not yet be supported, it is not covered here. The JDBC functionality is provided by the types in the java.sql namespace. There are four classes that are core to the operation of JDBC: DriverManager, Connection, Statement and ResultSet. The driver manager class (as its name suggests) manages the drivers for the data sources that may be accessed (as such it is not instantiated; all its members are static). The connection class represents the connection that is made to this data source. The last two classes represent the SQL statement and the returned query results. These classes are similar to those in .NET. The following piece of code (from the Java documentation) demonstrates a typical connection example.

```
Connection con = DriverManager.getConnection (
                "jdbc:odbc:wombat", "login", "password");

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");

while (rs.next())

{  int x = rs.getInt("a");
   String s = rs.getString("b");
   float f = rs.getFloat("c");

}
```

Here the connection string for a JDBC driver is “jdbc:protocol:database name”. In this case the protocol that is being used is the JDBC-ODBC bridge to access ODBC data sources. The database is local and called wombat. Like .NET the connection string can be more complicated (see the appropriate SDK documentation). The registration of database drivers with Java is not covered here, but is not complicated. While the driver manager class has other uses, its main one is to create a connection object from the connection string and optionally the login id and password, as seen here. The connection object is the connection to the data source. Once this has been created, it acts as the intermediate between the code and the data source. The SQL statement objects are created from the connection. Note that, unlike .NET, there is no conceptual object to tie the different types of SQL statement together and conceptually represent the data source in an application. If many different commands are to be performed on the data source Java’s method may be more efficient, but if similar actions are to be performed (as in an application just designed to update and check a stock database, for instance) then the .NET system is more intuitive. The statement object has methods for running a query to return data (as above) and for performing an update on the database. A simple select statement is shown above which returns three columns to the ResultSet.

It is here that JDBC’s major flaw lies: it was originally a separate API and was not in the original release of Java. This is a problem because, where .NET’s DataSet object is a fully supported .NET type (for example, is directly supported by visual components), the ResultSet class is known only within the JDBC types. Getting information out of the ResultSet is similar to using a

string tokeniser to interpret a saved file. As can be seen above, the ResultSet's *next* method moves its focus on to the next line, and each columns values are retrieved by using a get method with the column name (or index). Additionally, there are separate get methods for each SQL type, and the correct one must be used for each column. The ResultSet does have a method to return metadata for the set that allows information such as the data type of each column to be found, but this means that if the type of a column is not known, a *very* large `switch` statement would be needed, for example. This is not very efficient, so JDBC is not the best choice for using tables that are not pre-existing and known.

5 Support for development teams

5.1 Introduction

There are other factors to those mentioned in the previous section that can have an effect on the efficiency of a development team. The support each system gives for packaging is undoubtedly important, so this is discussed first. While this covers the internal organisation of the application, the external organisation (of the compiled files) is also important. This is most noticeable with units of reuse within multiple development team environments. This is discussed in the second part of this section. The final part covers the support of each system for adding documentation to classes. This is of great importance in development teams as the more documentation that can be added to classes, and the easier it is to add, the easier it is to use the finished class. The easier it is to access that information, the more efficiently it can be used.

5.2 Packaging

When developing an application, it should be divided up into logical packages. In .NET the C++ convention of namespaces is used. When an application or project is created it is placed within a new namespace. This has the same name, so calling an application “System”, for example, is a bad idea. When a sub-project is created (such as a project containing a class library and an application as sub-projects) it will be in a sub-namespace and so on. In reality, any namespaces can be used, but these are the conventions used in Visual Studio .NET, so will be generally used. To place a class in a namespace, it must be created within a block of that namespace. This is through similar syntax to a class declaration, and Figure 5.1.1 shows this. Each class must be declared within a namespace block, and as many blocks as are needed can exist for each namespace. To make the contents of a namespace visible the `using` directive must be used, as can be seen in Figure 5.1.1. The exact syntax varies according to language; the base (C#) syntax is shown. Sub-namespace names are delineated with a period, for example: the `Forms` namespace in the `Windows` namespace (which is itself in the root framework namespace of `System`). When a namespace block is declared, the full name (including periods, as seen in the using directives) must be used.

```
using PMArticle;
using System;
using System.IO;
using System.Net;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.ComponentModel;
using System.Runtime.Serialization.Formatters.Binary;

namespace Browser
{
    /**/
    public class ArchiveDialog : System.Windows.Forms.Form...
}
```

Figure 5.1.5 - Using packages in .NET.

Java uses a similar method of packaging. Rather than using a block, a package statement is placed at the start of a class file. This means that one file cannot contain classes for more than one package, which is a sensible demarcation. As can be seen in Figure 5.1.2, the package declaration is at the top of the file, and is again the fully qualified package name. In Java the import directive replaces the using directive. What is not made clear in Figure 5.1.2 is that it is a special case in that it refers to specific classes. More generally they will be imported on a package level, as in the .NET example above. This is done using a wildcard. For example, the statement "`import java.util.*;`" will import all of the classes in the util package. In both .NET and Java, classes can be accessed without importing their packages if their fully qualified name is used.

```
package PMArticle.Series;

import PMArticle.Entries.Entry;
import java.util.SortedMap;
import java.io.Serializable;

public class Series extends Entry implements Serializable
{
```

Figure 5.1.6 - Using packages in Java

5.3 The unit of reuse and version control

The unit of reuse in .NET is the Assembly. In Java the unit of reuse is nominally the JAR (Java Archive file), but individual class files can also be used. JAR files differ from Assemblies in that they can be compressed (as they are based on ZIP files). However, JAR files do not enforce the security features as .NET's assemblies do when given a strong name. This is partly due, as mentioned elsewhere for other parts of Java, to JAR files not existing in the initial release of Java. In this way JAR files are more like collections of resources, rather than a conceptual entity like assemblies. JAR files were designed specifically to make the downloading of applets more efficient, rather than as a genuine unit of reuse. They differ from ZIP files in that, like assemblies, they can have a manifest that describes the files within the assembly. This can allow for "self-executing" JAR files (in the Java sense this means the JAR filename can be given as a parameter to the runtime, instead of a class file). They remain as being optional, as can be seen particularly in

applets where the usage of individual class files still appears to be more common than JAR files. In both Java and .NET the unit of reuse is designed to store a logical grouping of types and resources (when the JAR file is taken as being Java's unit of reuse).

5.3.1 Assemblies

An assembly can be compiled as either an executable assembly (on windows computers an .EXE file) or a class library (similarly a .DLL file). For a type to be valid in .NET, it must be in an assembly (i.e., unlike Java, a class in a code file cannot be used independently). As the unit of reuse, all types are scoped according to their assembly. In addition to being a unit of reuse in the programming sense, the assembly is the atomic unit used in many parts of the .NET framework. For example, the code access security system assigns permissions on a per-assembly basis. There is a discussion of some parts of assemblies in the chapter on the common language runtime, particularly the manifest and metadata. The System.Reflection.Emit namespace allows dynamic assemblies to be created. These are beyond the scope of this document. An assembly can contain resources as well as types so icons can be stored with the components that use them, for example.

When an assembly is to be released it can be secured by giving it a *strong name*. The strong name is created for the assembly using the developer's private key, so the runtime can verify the source of the assembly. This allows developers to be sure that the assemblies that they reference are by the correct developer, and have not been altered since they were released. When an assembly has been signed with a strong name, all those assemblies it references must also have strong names. This allows a developer to be sure that their application will only run with the libraries that are supplied with it. In addition to the public key that the runtime uses to verify it, the strong name includes the digital signature and the version number.

The runtime examines the available assemblies upon program execution to check that they have the correct permissions, but also to check their version. When an assembly has been given a strong name (i.e., released), it is given a version number. As each successive version of an assembly is compiled the version number of the outputted assembly is increased. The assembly's manifest contains a list of the assemblies it is dependent on and their versions. The runtime will then only run the assembly if the correct versions of those assemblies are present. In the global assembly cache (see below) multiple versions of assemblies can coexist as each different version is considered to be a separate assembly. This avoids the problems traditionally caused in windows by different versions of DLLs. These are either due to newer versions of DLLs causing bugs in applications that require an older version, or by an older version overwriting a newer one (described in more depth in Pratschner, 2000). With the global assembly cache, the program just uses the version of the DLL it was programmed with. It is possible to override this behaviour if a newer version of an assembly offers enhanced performance. This is done simply by amending the configuration files of the application (or system).

The version number of the assembly is in four parts. The first two are the standard major and minor version numbers (as in Windows 3.1, etc.). The third is the build number, which increments with each subsequent compilation of the assembly. The last is the revision number. Together these form a unique identifier for the assembly. A text version of the assembly version can be added to the assembly's metadata. This is to allow a developer to add a description of that version and allows another program (for example an install program) to display it to the user.

The Global Assembly Cache (GAC) is a store that is used by the runtime on a system to store assemblies. Global assemblies are those that are installed on to the system for potential use by all applications, as opposed to local assemblies which are installed to the application's directory and are used only by that application. This is equivalent to installing a DLL into the Windows system directory rather than into the same directory as the rest of the application. As applications are installed, any global assemblies they require can be installed into the GAC. As mentioned above, the GAC differs from a standard directory in that the assembly name *and* the version number are used to fully identify the assembly, so there may be multiple files with the same filename in the GAC.

5.4 Class documentation

At the moment, the only .NET compiler is Visual Studio .NET, so this section may be specific to that, although Borland should implement the documentation features when their .NET compilers are released. It is also unclear whether all languages will support the addition of documentation blocks. In the same manner as Javadoc, .NET includes tokens that can be placed in a piece of code to allow documentation to be automatically generated. These tokens are XML tags. This is done via metadata as described in the chapter on the CLR. These comments allow classes to be self-describing. If a class in a project (or a referenced assembly) has these comments added, any other classes in the assembly will then see the descriptions of the class and members as they use it (in Visual Studio, the update is instant). In addition to the documentation comments appearing as descriptions within Visual Studio, the compiler can also strip them out into an XML file.

The documentation comments are similar to standard comments (specifically those following "//", C++ style), except they are preceded by three slashes and contain the tokens that describe the nature of the comment. They must also be in specific locations. For example, the following code snippet shows how to add a description to a class:

```
/// <summary>
/// Summary description for DocumentationExampleClass.
/// </summary>

public class DocumentationExampleClass

{

}
```

Here, the token <summary> marks the comment as a description of the class. This description block must immediately precede the class declaration. If an instance of this class is declared or the class is referenced in another class, this description is used to describe it. Figure 5.3.1 shows the popup that appears when holding the mouse pointer over a class name in Visual Studio. Figure 5.3.2 shows the description being displayed by a box listing the available classes, as happens when instantiating objects.

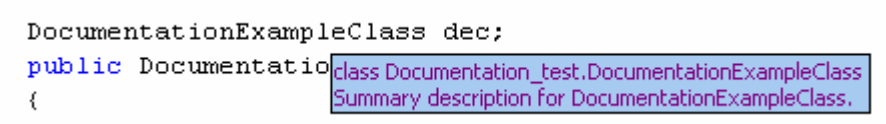


Figure 5.3.1 - A popup description box in Visual Studio .NET describing a class.

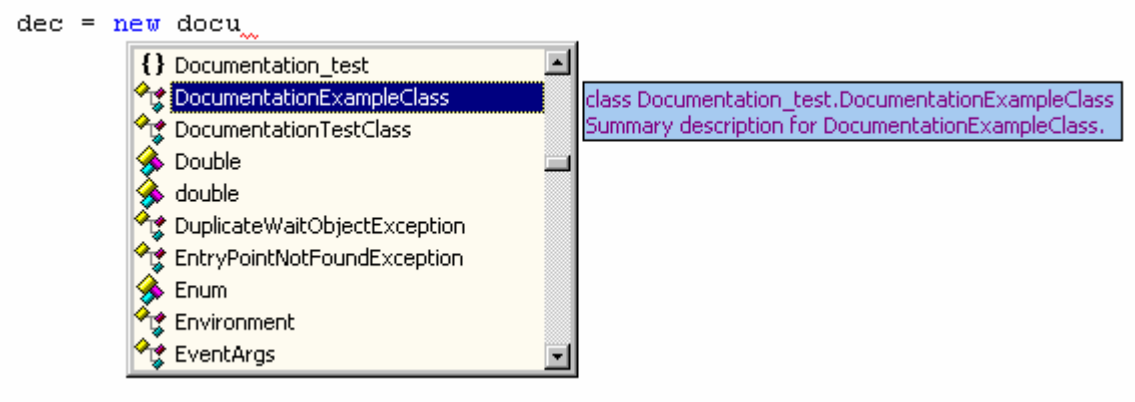


Figure 5.3.2 - The description of the class shown when choosing a class from a list.

The same method is used for adding descriptions to members of the class. The following example shows two constructors for the example class with descriptions added:

```
/// <summary>
/// A description of the default constructor goes here.
/// </summary>
public DocumentationExampleClass()
{
	//
	// TODO: Add constructor logic here
	//
}
```

```
/// <summary>
/// A description of the overloaded constructor goes here.
/// </summary>
/// <param name="index">A description of the first parameter
/// goes here</param>
/// <param name="inverted">A description of the second
/// parameter goes here</param>
public DocumentationExampleClass(int index, bool inverted)
{
      ind = index; inv = inverted;
}
```

As can be seen from the default constructor, the application of a description block to the method is identical to the class. A todo block can also be seen in this method. These blocks are not documentation blocks (note the two slashes as opposed to three), but are used to add items into a to-do list for the project. Whether this feature will be transferred to other compilers is less certain than the documentation features. The second constructor demonstrates two additional features. Firstly, the description between the opening and closing of a token block can carry over onto multiple lines if necessary. Secondly, method parameters are also described. The token <param> includes the name of the parameter to describe, so all of the method parameters can be described individually. As the constructor is overloaded, it has two separate sets of descriptions. These are shown in Visual Studio by a set of up/down arrows with numbers assigned to the overloads, as shown in the circled area of Figure 5.3.3.

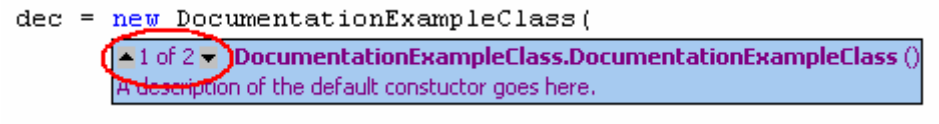


Figure 5.3.3 - The description of the first overload of the constructor.

Clicking on the arrows or pressing the appropriate cursor key on the keyboard moves between each overload of the method. As is shown in Figure 5.3.4, in the case of methods with parameters, the method description is not displayed. Instead a description of the first parameter is displayed. When the first parameter is entered and the delineator between parameters has been typed (in this case a comma) the description of the second parameter replaces that of the first, as shown in Figure 5.3.5. This continues for all of the parameters. When the last parameter has been entered and the method call closed (i.e., “)” typed), the description returns to that of the method.

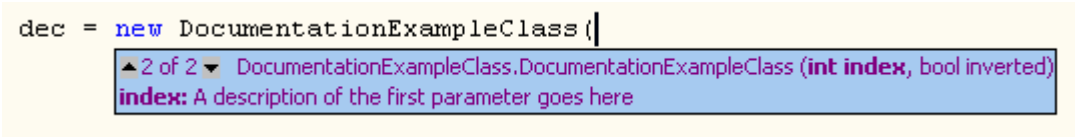


Figure 5.3.4 - The description of the second overload of the constructor.

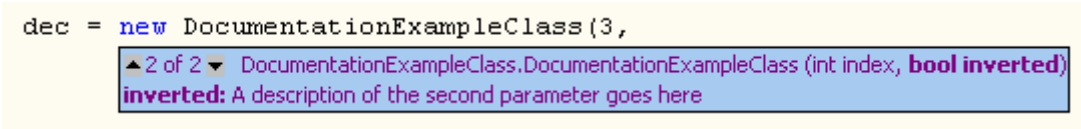


Figure 5.3.5 - The description of the second parameter of the method.

The following snippet shows the last of the features for documenting methods, the return value:

```
/// <summary>
/// A description of the method goes here.
/// </summary>
/// <param name="inverted">A description of the parameter goes
/// here.</param>
/// <returns>A description of the return value goes
/// here.</returns>
public int getIndex(bool inverted)
{
      if(inverted != inv) return 0;
      return ind;
}
```

Here, the <returns> token documents the return value. This has no obvious use in Visual Studio at the moment. As can be seen in Figures 5.3.6 and 5.3.7, when choosing the method from the picker it is the method description that is shown, and when the method has been selected/typed it is the parameter(s) if any that are shown. Either this description is used elsewhere, or the documentation support in the .NET framework is more extensive than Visual Studio as of yet uses. The same is also true of adding descriptions to members that cannot be seen outside the class, although in this case it is because they are not visible outside the class.

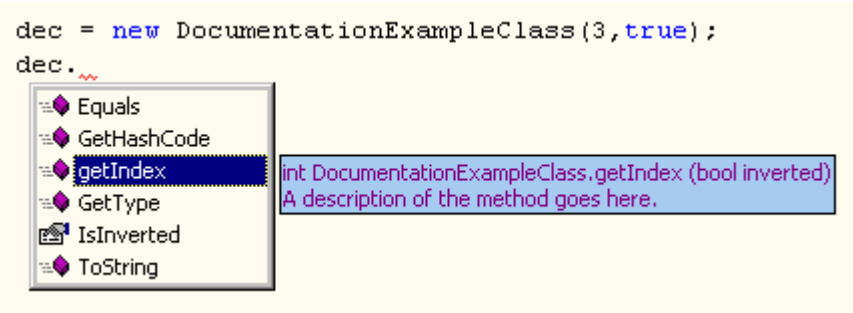


Figure 5.3.6 - The member picker displaying a method description.

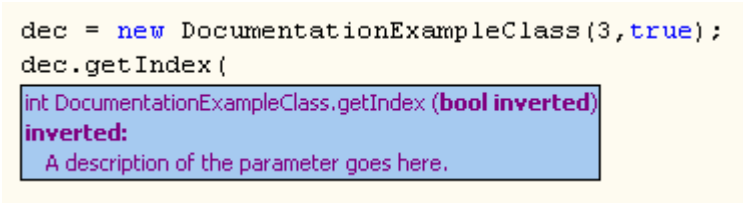


Figure 5.3.7 - The description of a method while typing.

The description of other members is carried out in the same way as methods. The following example shows a description applied to a property and Figure 5.3.8 illustrates its usage. There is also another tag, <value>, which can be used to describe a property. This is more specific to properties, whereas <summary> is generic.

```
/// <summary>
/// A description of the property goes here.
/// </summary>
public bool IsInverted
{      get
       {      return inv;
       }
       set
       {      inv = value;
       }
}
```

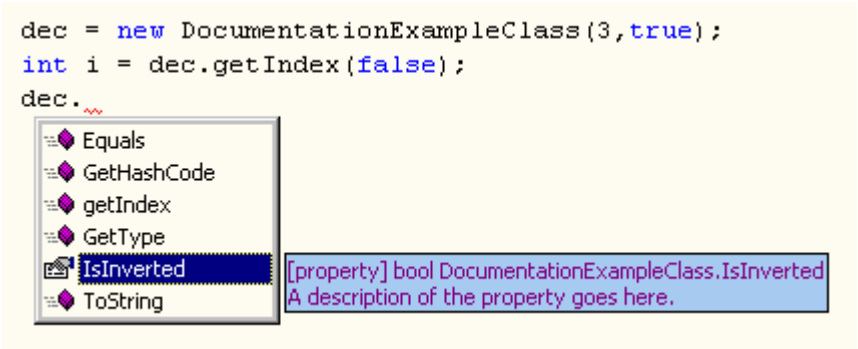


Figure 5.3.8 - The description of a property.

There are other tags that can be mixed with those shown above, such as <example> which allows a usage example to be appended to the description, and <remarks> which allows further description going into greater depth than the summary. The tags <exception> and <permission> are used as specialised tags for the appropriate Types. There are also several tags that are used within comment blocks to apply formatting to the comment when it is used, such as <para> and <code> while some (such as <see>) can reference other Types.

There is another token that is used by Visual Studio to make the code itself a little easier to follow, "#region". This is not an XML tag but is used in the same manner as normal comments, to make it easier to understand a piece of code rather than to reference it. In Visual Studio, methods, comments etc. can be rolled up (i.e., collapsed to a symbol) to lessen the clutter. The region token allows a named region to be specified that can be rolled up to just the name. As with the symbols representing collapsed method bodies and comments, holding the mouse pointer over the name displays a popup containing the hidden code. This is shown in Figure 5.3.9. While this is not part of the documentation system, it does allow the code to be collapsed into explanations of what is going on so it can help a programmer unfamiliar with the code to understand it.

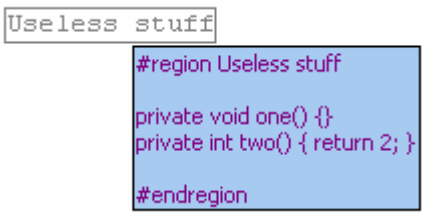


Figure 5.3.9 - A collapsed region with the mouse-over popup visible.

5.4.1 Java

The documentation features of Java and .NET share many features, and in some respects .NET's version seems like a slightly matured update of Java's. For example, where .NET formats its documentation with XML, Java uses HTML (as both predate XML). Where .NET documentation comments follow "///" instead of "//", Java's lie between an opening "/**" and a closing "*/"

(the double star in the opening token differentiating it from a normal comment. By convention, a star begins each line within the comment. Like .NET, each of these documentation comments must be situated immediately before the Type/member to be described. Some of the formatting tags of HTML can be used within the comments to format the resultant text (as can XML in .NET comments).

The first sentence of the comment (delineated by the period) is used as the summary of the member, so only single sentence summaries can be used (as opposed to with a separate tag like <summary>). Any text following that is the description of the member (equivalent to the <remark> in .NET). Following this general descriptive text, there are several tags that describe the specifics of the member. Some of these are direct parallels of tags in .NET: @see, @param and @return. One of the tags, @exception, has a different use to <exception> in .NET. In Java it is used to signify which exceptions a method can throw, whereas it is used to document exception classes in .NET and the exceptions thrown have to be commented on in the remarks section. There are also some tags in Java that are not in .NET: @author allows the author of the class to be named and @version allows a version identifier to be added. Putting the version number in the documentation is simple way of supporting version control, but .NET has a full version control system, and the version number is automatically saved with the class. The author tag has no equivalent in .NET, which does have support for certificates and so on. To create the documentation for a class, an external program in the SDK must be run, and the javadoc comments are extracted.

5.4.2 Summary

While documentation is often ancillary to a sole programmer the availability and presentation of descriptions is paramount in a team environment, particularly where multiple teams are involved. When using Visual Studio .NET, the close integration of the documentation and the ease of adding it greatly increases the ease of programming and decreases the time spent reading the reference documentation. This should make development teams using it more efficient than otherwise, and is an advantage over the more separate style of Java.

6 Multiple language development

6.1 The Common Language Runtime

Microsoft .NET was designed using the principle that each programming language is suitable for different kinds of tasks. Traditionally, the language is chosen because of the environment (i.e., within a particular company) and is standardised across the project. With .NET, the .NET framework is used across the project, but the language chosen for each task within it can be the one most suited to it. Indeed, much of the .NET framework was programmed in C# (Templeman, 2001) This move offers fundamental benefits to developers: "Though I risk sounding like a Microsoftist here, the current Babel-esque lack of interoperability that divides languages is an impediment to productivity, innovation, and true openness." (Alexander Staubo in Udell, 2000). The interoperability of the languages is supported through the *common language runtime* (CLR). "The CLR supplies the common infrastructure that allows tools and programming languages to benefit from cross-language integration."(from the Framework SDK). The CLR specification (in the SDK) describes it as follows:

The CLR provides the following services:

- Code management
- Software memory isolation
- Verification of the type safety of MSIL
- Conversion of MSIL to native code
- Loading and execution of managed code (MSIL or native)
- Accessing metadata (enhanced type information)
- Managing memory for managed objects
- Insertion and execution of security checks
- Handling exceptions, including cross-language exceptions
- Interoperation between .NET Framework objects and COM objects
- Automation of object layout for late binding
- Supporting developer services (profiling, debugging, etc.)

The runtime contains: the common language specification (CLS), the common type system (CTS), the intermediate language (MSIL), metadata and the virtual execution system (VES). The VES has little bearing on the individual languages and so is described in an earlier chapter.

For a component to be reused by code written in a different language, it must use only the features specified in the CLS. The CTS defines the basic types that are native to the CLR, and as such forms the base of the CLR by ensuring all languages use the same basic types. The compiler compiles the code into MSIL and stores any additional information in the associated metadata. MSIL acts as a machine independent language in much the same

way as Java .class files and acts as an intermediate between the source code and native code. For the runtime and other tools to correctly interpret a program, they need to know additional information about each segment (such as the language that it was programmed in). This additional information is stored in the metadata, which can also be used to extend the features of the runtime (i.e., adding types). The VES implements the CTS, loads the code and verifies it. It then uses a JIT to compile the code into native code and then executes the compiled code and manages it. The VES uses the metadata to link the program and add any extra features. Each of the components of the CLR (aside from the VES) are described in more detail in the subsequent sections of this chapter. The CLR is also outlined in the MSDN library (see Watkins, 2000) and in the common language infrastructure specification documents (available from Microsoft, October 2000). Further information about the support of some languages is also available, for example: C++ (Sells, 2001), COBOL (Kadhim, 2000) and Eiffel (Simon, 2000).

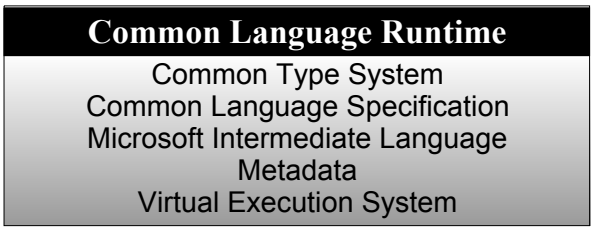


Figure 6.1.2 - CLR summary.

6.2 The Common Type System

> The term **type** is often used in the world of value-oriented programming to mean data representation. In the object-oriented world it usually refers to behavior rather than to representation. In the CTS, type is used to mean both of these things: two entities have the same type if and only if they have both compatible representations and behaviors. (.NET Framework SDK)

For object code written in different languages to be compatible, the types used in them must be the same. The common language runtime includes the Common Type System (CTS) for this purpose. The CTS defines a set of common types that are native to the runtime (shown in Table 6.2.1), and defines rules for compilers to enforce consistency in their use of types. The CTS is fully extensible to allow for any extra functionality a developer needs to add, and has received favorable reports on its efficiency (for example, Kennedy and Syme, 2001). Types in .NET are either classes, interfaces or value types. Class and interface types are *reference types*. In other words, when objects are passed to methods, they are passed by reference and it is the same object that is used in the method. Value types represent a simple

value, an atomic piece of data. These are the traditional built in data types of languages (integers, booleans, etc.). They are passed to methods as arguments by copying them; it is a *copy* of the value that is used in the method. In addition to these types, the CTS also contains *Delegates*. These are essentially type-safe function pointers. Pointers are probably the biggest cause of error in C++ programming, but they are incredibly useful. Delegates are an attempt to get the advantages of pointers without the disadvantages.

Name in MSIL assembler (CTS)	CLS Type?	Name in class library	Description
bool	Yes	System.Boolean	True/false value
char	Yes	System.Char	Unicode 16-bit char.
class System.Object	Yes	System.Object	Object or boxed value type
class System.String	Yes	System.String	Unicode string
float32	Yes	System.Single	IEEE 32-bit float
float64	Yes	System.Double	IEEE 64-bit float
int8	No	System.Sbyte	Signed 8-bit integer
int16	Yes	System.Int16	Signed 16-bit integer
int32	Yes	System.Int32	Signed 32-bit integer
int64	Yes	System.Int64	Signed 64-bit integer
native int	Yes	System.IntPtr	Signed integer, native size
native unsigned int	No		Unsigned integer, native size
typedref	No	System.TypedReferen ce	Pointer plus runtime type
unsigned int8	Yes	System.Byte	Unsigned 8-bit integer
unsigned int16	No	System.Uint16	Unsigned 16-bit integer
unsigned int32	No	System.Uint32	Unsigned 32-bit integer
unsigned int64	No	System.Uint64	Unsigned 64-bit integer

Table 6.2.1 - Built in types. (taken from the .NET SDK documentation)

Class types

A class in the CTS can contain four different types of member: methods, properties, events and fields. The CTS supports single inheritance, while multiple interfaces can be implemented by a class (as in Java). Classes can be abstract for use as base classes. A class can have certain characteristics which the compiler translates from the language syntax used (standard modifier keywords such as *private*) into standard metadata. The CTS specifies a base set of characteristics. Not all languages support all of these basic characteristics and additional, custom ones can be added through *attributes* (see section on metadata for more details). The CTS standard characteristics are described in Table 6.2.2.

Characteristic	Description	Nearest Java equivalent

sealed	Can not be inherited	final
implements	Complies to specified interface	implements
abstract	Can not be instantiated	abstract
inherits	Derived from specified base class	extends
exported	Visible outside the assembly	public
not exported	Not visible outside the assembly	[default]

Table 6.2.2 - CTS Type characteristics.

The CTS also defines a set of basic characteristics for the class members, which are described in Table 6.2.3.

Characteristic	Description	Nearest Java equivalent
abstract	Implementation needs to be supplied in subclass	abstract
final	This member can not be overridden in a subclass	final
overrides	Implementation replaces that in the base class	[implicit]
static	The member is shared among all instances	static
overloads	Specifies an overloaded method	[implicit]
virtual	Implementation can be overridden by a subclass	[implicit]
synchronised	Can only be accessed by one thread at a time	synchronized
[**Visibility**]	Visible only within the same class:	
private	or a nested class	private
family	and its subclasses	[N/A]
assembly	and in the same assembly	[default]
family and assembly	and the subclasses *in the same assembly*	[N/A]
family or assembly	and subclasses *and* in the same assembly	protected
public	and everywhere else	public

Table 6.2.3 - CTS Member Characteristics.

Some of these characteristics have no comparison in Java as they explicitly declare features that Java declares implicitly (through the lack of the opposite keyword). Some of the CLR languages are the same, but the characteristic is explicitly added in to the metadata by the compiler. Some of the visibility options have no comparison in Java due to the difference in assemblies and packages (see section on units of reuse).

6.2.1 Interface types

An interface specifies a contract which implementing classes must fulfil. It can contain method definitions, but cannot supply implementations for dynamic methods. It can however contain static members that can be used to provide generic functionality and hold appropriate constants. As such, a class that uses an interface must implement the methods declared in it. Inside the interface, there can be no constructor and all members must be public (as interfaces can never be instantiated). Interfaces can also specify that any implementing classes must also implement other interfaces.

6.2.2 Value types

The runtime allows custom data types to be created by a programmer that will still be passed by value. In effect, this allows extra primitive types to be added to the framework. However, value types differ from primitive types in that they can have methods defined. The .NET Framework's basic types are built-in value types (those listed in Table 6.2.1). Value types can be used as local variables and so on, in the same way that a primitive type could. They are instantiated, and are always initially zero. They do not require a constructor to be called upon instantiation. Storing a value type does not incur the same overhead as a class instance.

Value types are created by extending **System.ValueType**. For a value type to be efficient it must be relatively small (as it is passed by value), otherwise it would be better to use a standard class type and pass it by reference. The runtime automatically creates the corresponding "boxed type" (a reference type that refers to the value type). Boxed types have not been fully implemented at present, so the details will not be discussed here. Value types can have all four kinds of member and can implement interfaces, but do not require constructors. All value types are sealed (see definition in class types, above).

There is a special instance of value types known as enumerations, which wrap a primitive type. The enumerated type defines aliases for its values, as seen in languages such as C (enums). To create an enumeration, it must extend **System.Enum**. Each enumeration has a type (most commonly an integer) that must be one of the CTS built-in types. It contains a set of static fields representing constants, the value of which can be automatically assigned. Two fields can be specifically given the same value, and the runtime considers one of these to be primary to avoid confusion. An enumeration can contain no other members aside from these fields and cannot implement interfaces.

Delegates

Delegates serve as type-safe function pointers. They are also used for event handling and callbacks. When a delegate is instantiated, it is assigned to a particular method on an object. Delegates are managed objects, so they will always point to a valid object and cannot corrupt the memory occupied by other objects. The runtime supplies the implementations of methods in a delegate, not the user.

Delegates must be created by extending either **System.Delegate** or **System.MulticastDelegate**. The delegate contains one method if the former is used, or multiple methods if the delegate extends the latter (which contains static methods to add and remove method references from the invocation list). The invocation list of a delegate is the list of delegates that are executed when the delegate's invoke method is called. When a delegate is declared, the parameters and return type are specified (which must match the method that is being referenced). The delegate is then declared with the method to be referenced passed to the constructor of the delegate. The following example (from the SDK) illustrates the declaration, instantiation and use of a delegate with no parameters which returns nothing.

```
public delegate void MyDelegate();
MyDelegate TheDelegate = new MyDelegate(MyClass.Method1);
TheDelegate();
```

In event handling the event delegate is used as a go-between for the object that sends the event and the object that receives it. This is described in more detail in the section on event handling.

6.3 The Common Language Specification

The Common Language Specification (CLS) is a set of language features specifying a basic level of behaviour for the runtime. To make C++ compliant, a version known as managed C++ is used, as described by Scanlon (2001). This uses the components and features of the .NET framework and the CLS as specified below. A component needs to use *only* these features if it is to be reused in any language (it is then said to be a CLS-compliant component). For CLS compliance, only those features specified by the CLS must be used: in the definitions of public classes and their public members, the definitions of members visible to sub-classes, and in their parameters. If a class or member cannot be seen outside the code (i.e., in a private class), then it can use any features available in that language and still remain CLS-compliant. For example, unmanaged C++ (traditional) can be used for non-visible code. The implementation of the class can also use any available feature without compromising the compliance. The details of the CLS are only needed by tools developers and those porting languages and are described in depth in the .NET SDK in the tool developers guide (also available at Microsoft, October 2000). The CLS is summarised in Table 6.3.1 and Table 6.3.2.

	CLS Feature Description
Primitive types:	System.Boolean (true or false) System.Char (2-byte unsigned integer) System.Byte (1-byte unsigned integer) System.Int16 (2-byte signed integer) System.Int32 (4-byte signed integer) System.Int64 (8-byte signed integer) System.Single (4-byte floating point number) System.Double (8-byte floating point number) System.String (A string of zero or more characters; null is allowed.) System.Object (The root of all class inheritance hierarchies)
Arrays:	Can have known type, known (>= 1) dimension (rank), with zero lower bound. Element type must be a CLS type.
Types:	Can be abstract or sealed (but not both). Can implement zero or more interfaces; different interfaces can have methods with the same name and signature. Can be derived from exactly one type and override or hide members provided by that type. Can have zero or more members, which are fields, methods, events, or types. Can have zero or more constructors. Can have public or assembly visibility, but only public members are considered part of the "public" interface of the type. Value types must inherit from System.ValueType, unless they are enumerations, in which case they inherit from System.Enum.

Table 6.3.1 - Summary of CLS features.

	CLS Feature Description
Type members:	Members can override or hide other members in another type. Argument types and return types must be CLS-compliant types. Constructors, methods, and properties can be overloaded. Members can be abstract, but not if the type is sealed. Can have public. Private, family, assembly, FamilyAndAssembly, or FamilyOrAssembly visibility, but only Public, Family and FamilyOrAssembly are considered part of the "public" interface of the type.

Methods:	Can be one of virtual, instance, or static. Virtual and instance methods can be abstract or concrete (static methods must be concrete). Virtual methods can be final (or not).
Fields:	Can be static or non-static. Static fields can be initonly or literal.
Properties:	Can be exposed as Get and Set methods instead of using property syntax. Return type of the Getter and the first argument of the setter must be identical; this is the "property type." Cannot differ by the "property type" alone. If an X property is defined, you cannot define Get_X and Set_X methods in the same class. Can be indexed. Must follow this naming pattern: get_<PropName>, set_<PropName>.
Enumerations:	Underlying type must be Byte, Int16, Int32, or Int64. Each member is a static literal field of the enum's type. Cannot implement any interfaces. Multiple fields can be assigned the same value. Must inherit from System.Enum.
Exceptions:	Can be thrown and caught. Must inherit from System.Exception.
Interfaces:	Can require implementation of other interfaces. Can define properties, events, and virtual methods.
Event:	Add and remove methods must be either both provided or both absent; each of these methods take one parameter, which is a class derived from System.Delegate. Must following this naming pattern: add_<EventName>, remove_<EventName>, and raise_<EventName>.
Custom Attributes:	Can use only the following types: Type, String, Char, Boolean, Byte, Int16, Int32, Int64, Single, Double, Enum (of a CLS type), Object.
Delegates:	Can be created and invoked.
Identifiers:	The first character must come from a restricted set. Case cannot be used to distinguish between identifiers within a single name scope (i.e., types within assemblies, members within types).

Table 6.3.2 - Summary of CLS features (continued).

6.4 Microsoft Intermediate Language

When source code is compiled by a .NET compiler, it is into a file made up of two parts: the metadata (described elsewhere in this chapter) and the compiled code. This file is known as a portable executable (PE) file. The compiled code within the PE file is usually in a standard format known as the Microsoft Intermediate Language (MSIL). The MSIL allows greater flexibility when using different languages and is examined further by Gordon and Syme (2001). MSIL is not native code but is instead very much like a compiled Java ".class" file, in that it is portable across different platforms. It differs from the Java equivalent as it is compiled into native code by a Just-in-time compiler (JITter) on initial execution whereas Java interprets the code at runtime every time. This allows the same PE file to be executed on any supported platform. The goals of MSIL are to be platform independent, to facilitate easy compilation from numerous languages, and to be compiled to native machine code on each platform with maximal efficiency. MSIL comprises a set of instructions (summarised briefly in Table 6.4.1) which define exactly what the native code will do. Chien (2001) discusses the issues associated with the multi-platform IL approach as it may affect viruses.

The compiled PE file may not always contain MSIL as there are two alternatives: a subset of MSIL known as the Optimised Intermediate Language (OptIL) and it can be pre-compiled into native code. The code can also be compiled into native code the first time the program is executed if necessary. OptIL is a specialised subset of MSIL that uses embedded annotations for a compiler to include further information. These annotations can be ignored by non-supporting tools allowing an OptIL file to be seen as OptIL by tools that support it, or as standard MSIL by those that do not. These extensions allow a supporting JITter to compile the OptIL to the same standard as MSIL, but with greater efficiency and speed. MSIL is described in more depth in the tools developers guide (also available at Microsoft, October 2000).

Instruction type	Description
Arithmetic operations	The standard operators supported by CTS types.
Logical operations	
Control flow	Supports for/while/if/switch statements, method calls/returns, exceptions and breakpoints.
Direct memory access	
Stack manipulation	Supports pop & push.
Argument and Local variables	Done using the stack
Stack allocation	Used for memory allocation.
Object model	Used for converting between value & reference types (box/unbox).
Values of Instantiable types	can create/copy values of types

Critical region	synchronisation
Arrays	1D, starting from 0, bound checking
Typed locations	

Table 6.4.1 - Summary of the MSIL instruction set.

6.5 Metadata

Metadata is one of the core aspects of the .NET framework. When a class or an assembly is compiled, metadata is created for it by the compiler and added to the compiled file. This metadata acts as the description of the class or assembly and allows loaded classes to interact; it contains all the information needed for the code to be used. For a component, the complete description of it by its metadata (*component metadata*) is necessary in order for it to be self-contained. When a compiled class is loaded into the runtime, it is composed of two parts: the code and the metadata. The runtime loads the associated metadata into data structures that it references when it needs information on the class and these are used to locate and load the members when they are needed. In normal use the compiler automatically handles the metadata so an in depth knowledge is needed only for tool and compiler programming.

The framework provides two different methods for accessing metadata. This is via two different APIs (class libraries): managed and unmanaged. The unmanaged API is a set of unmanaged COM interfaces. These are more basic and allow lower level access to the metadata than the managed API. (i.e., they return only the member declarations from a class). The managed API is part of the framework API, and is also known as the Reflection API. The reflection API allows the data structures that are created inside the runtime from the metadata to be examined. These services are more automated than the unmanaged API and provide easier access for normal use. However the unmanaged APIs give the programmer more control and access to more information so may be needed for more in-depth usage.

Contents of a class' metadata

> Consider a type definition such as a C++ class. The metadata for that type would completely describe the class, including the methods and their parameters, their calling conventions, the class's data members, and the visibility of all class members. In Visual Basic®, these concepts would extend to the events the class can fire. Metadata is intended to be the union of all such attributes exposed by any language. If you've programmed in the Java language, you might notice that .class files expose much of the same information as metadata. (Pietrek, 2000)

The metadata of a class describes every type and member defined or referenced by the class, as well as additional information about the class. The description of the class includes: its name, version (see R.E.P.), culture, public key (to verify the source of the class), declarations of the exported types (to introduce them into the CTS), the identity of assemblies upon which the code depends and the security permissions that the code needs before it can run.

The metadata contains a description of the types present in the code. For each type, it holds: its name, visibility, base class, the interfaces that it implements and descriptions of any members of the type. The members are methods, fields, properties, events or nested types. For each method, it records: if the code is managed by .NET, if it is written in the Intermediate Language or in native code and the location of the method body in the code. The metadata also contains *attributes* for the code, which are additional descriptive elements that can modify types or members. Whereas most metadata is generated by the compiler, the attributes are added by the programmer to get more control over the runtime behaviour of the code. As well as the attributes already present in the .NET Framework, custom attributes can be created for even more control.

Metadata attributes and extensibility

Attributes are tokens that can be added to a piece code to alter its behaviour. Existing examples of these include keywords such as public and private in many languages. They can also be used for storing information that describes the code (such as the name of the file). They can be added to a piece of code by using an Attribute object as shown below, and are automatically stored in the metadata of the compiled class. As well as using predefined attributes (.NET's built-in keywords, for example) custom attributes can be created by the programmer. This is accomplished by using a custom class that extends the framework class *System.Attribute*. This ability to create custom attributes greatly expands the potential uses of metadata.

An attribute is added to a piece of code by initialising it immediately before the code it is to act on. The following code snippet (in C#) is taken from the SDK and shows the use of an attribute called MyInfo on the Main function:

```
using System;
using System.Reflection;

//Import attribute namespace.
using MyNamespace;


public class MainApp{

   //Call attributes between brackets in C#.

   [ MyInfo("information") ]
   public static void Main(){

      Console.WriteLine("Hello world!");

      }
```

```
        }
```

The initialisation of the attribute (`[ MyInfo("information") ]`) is followed by the method definition for Main() so it acts on that method. In effect, the declaration becomes `MyInfo public static void Main()`. The constructor for the attribute allows information to be passed into the attribute to customise the instance if needed. The corollary code snippet from the SDK illustrates the definition of an attribute:

```
//declare namespace
namespace MyNamespace {


        using System;
        using System.Reflection;

        //What will this attribute modify?
        [AttributeUsage(AttributeTargets.All)]

        public class MeaningOfLifeAttribute : System.Attribute {


              private int MeaningOfLife = 0;
              public  MeaningOfLifeAttribute(int MeaningOfLife)

              {

                 MeaningOfLife = MeaningOfLife;

              }


              //This method is provided so we can later retrieve data
              public int get(){
              return MeaningOfLife;

           }
```

This defines an attribute called MeaningOfLifeAttribute (it is a naming convention for custom attribute names to end with Attribute). It is recognised by .NET as an attribute as it extends System.Attribute, as stated earlier. This particular attribute merely allows an integer to be stored along with any element of code. This integer can be recovered using the get method. The custom attribute itself has an attribute used upon it. This built-in attribute (`[AttributeUsage(AttributeTargets.All)]`) is part of the reflection API and is used by the framework to define how the custom attribute can be used. AttributeUsage has three members which can be specified by the constructor: AttributeTargets, Inherited and AllowMultiple. In this example, a value of All is used for AttributeTargets, which specifies that MeaningOfLifeAttribute can be used on anything. Other valid values are Type, Class and so on. The purpose of the AttributeTargets member is just to specify what elements of user code the attribute can be used upon. The Inherited member is a Boolean property of AttributeUsage allows the programmer to specify if the attribute will be inherited by the subclass when the target element is extended. Similarly, the AllowMultiple property specifies whether an object can sustain multiple instances of the attribute through inheritance, or if only one instance can exist. More than one of

these values can be used in the constructor of AttributeUsage by OR-ing them together (i.e., the "|" operator).

To make use of a custom attribute it needs to be identified and interrogated at run time. The class that is extended to create an attribute has a static method (Attribute.GetCustomAttributes) that returns the custom attributes present in the code element. Information stored in a specific attribute can be obtained by creating an instance of the attribute and using any supplied get methods. The following code snippet (again from the SDK) demonstrates how the attribute defined above can be recalled:

```
using System;
using MyNamespace;

class MainApp{

   public static void Main() {

      //Call function to get and print the attribute.
      GetAttr(typeof(MyClass));

    }

   public static void GetAttr(Type t){

      int Ret= 0;

      //Create an array from object class in which to store
      //array information.
      Attribute[]attributeArray = Attribute.GetCustomAttributes(t);

      //Loop through array and get all instances of MeaningOfLifeAttribute
      foreach(Attribute a in attributeArray){

         if(a is MeaningOfLifeAttribute){

            //Put found attributes in a new instance of
            //MeaningOfLifeAttribute.
            MeaningOfLifeAttribute attributeTmp = (MeaningOfLifeAttribute)a;

            //Call the get method to retrieve value.
            Ret = attributeTmp.get();

           //Print value.
           Console.WriteLine(Ret);

         }

      }

   }

}
```

Attribute.GetCustomAttributes is used to get an array of the attributes on the class `MyClass`. Each of these is tested to find which are instances of MeaningOfLifeAttribute and those that are, are cast into an instance. The get methods of the instances can then be called to retrieve the information stored in the attributes.

There are many attributes built into the .NET framework. For example, there are attributes that: describe the preferred layout of class instances by a tool or compiler, specify the size of the class for when it is needed, describe the packing size for a class for alignment purposes, allow static functions or data members that are not visible outside the file and so on.

Structure of metadata

The raw metadata is hidden, so one of the two APIs mentioned above must be used to access it. This allows the physical structure to be altered if necessary, while the APIs present a consistent (virtual) view of the metadata.

The metadata is arranged in a hierarchy that follows that of the code elements.

Figure 6.5.1 shows the basic structure for metadata in an assembly. The key at the bottom right corner shows the colour coding: for each instance of a code element, there will be collections of its sub-elements. Each instance within the metadata is the metadata for the appropriate code element. An assembly can contain a number of modules, although often only one (as shown here with only one instance - called “Module”). Each of these modules can contain types (i.e., classes) and global methods. Global methods (those external to a class) can be written in languages such as C++ and are not present in the common language specification so need to be expressed in the metadata (see the next section for more details). The Types each contain methods, fields, properties and events. The methods contain the parameters that are passed to and returned from them.

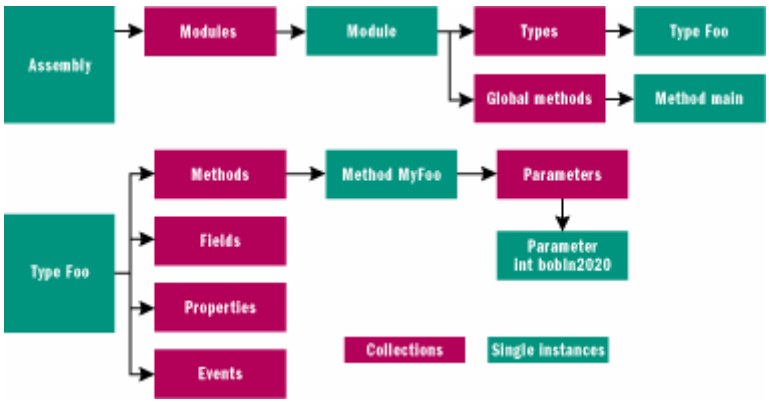


Figure 6.5.1 - The Metadata Hierarchy for Assemblies. (Pietrek, 2000)

The metadata for the assembly itself (i.e., the top of the hierarchy) contains a version number: the Module Version Identity (MVID). This ID is incremented every time the assembly is rebuilt and forms the basis of .NET’s version support. The type metadata records its base type as well as metadata acting on the type as a whole. The fields, properties and method parameters will all record their type using a type signature. Each method of a type will have a unique method type signature based upon its name and the number and type signatures of its parameters (this allows method overloading). Additional information may be available (as well as that in the type signature) in a method parameter definition, but this is optional so may not be present.

Type signatures

Type signatures are a series of bytes that uniquely identify a type. They are designed to be compared for equality checking. The first byte describes the calling convention of the type (a modifier such as __stdcall in C++). The second byte is the number of parameters passed to it. The third byte represents the type of the returned value. The remaining bytes represent the types of the parameters.

```
<Calling convention>
<Number of parameters >
<return type value>
<parameter 1>
<parameter 2>
•••
<parameter n>
```

(Pietrek, 2000)

In addition, a type modifier can be inserted before a byte representing a type. This allows the full range of available types to be represented. The most common of these modifiers signify pointers and arrays, where they are of the type specified in the immediately following byte. The other common type modifier represents a class: this specifies that the type is a custom class and is followed by a metadata token representing the class.

Uses of metadata

The metadata of a file is used to represent it to the runtime environment and hold any additional information needed by the programmer. The information that is emitted is independent of the language in which it was written and remains consistent. The metadata can also be used by compilers, tools and the runtime for them to communicate with each other.

The runtime uses metadata to identify and describe the code to it. This includes using it to perform the runtime linking of the code and adding any defined types to the Type system. Through the attributes that are embedded in metadata, metadata is used in the generation of native code, the marshalling of data when being passed to unmanaged code and to enable the calling of methods present in unmanaged code. During program execution, the metadata is used by the runtime to facilitate memory management (the garbage collection system) and manage direct memory access (the assignment, protection and monitoring of memory locations). The runtime also uses the metadata to keep track of allowed access privileges for security purposes. The runtime uses the metadata to create a representation of the code (stored in data structures in memory), which allows it to locate Types and members. For example, the tokens representing a parameter call are used by the runtime to locate the appropriate method in the in-memory data structures.

The metadata is used by programmers to store information, and to enable additional functionality for the program. This includes full descriptions of the methods and their parameters to make the components self-describing to the programmer (see section on Class documentation). This makes .NET class libraries fully self-describing which offers great convenience to the programmer. As this description includes the physical aspects of the class, a compiler allows a class to be written which extends or uses a pre-existing class written in any language, as long as it is managed code - no interfaces are needed. The metadata ensures that two components can always get enough information from each other to interact. Metadata is also used to support additional or language specific features that the CLS does not support, for example global static members (as seen in *Structure*, above). The ultimate example of metadata supporting non-CLS compliant features

is in providing access to unmanaged code using the file's Import & Export Address Tables. Metadata can also allow code to interact with COM classes to provide compatibility with existing systems (see Grimes, 2001).

Assembly metadata

> An assembly is a logical unit of functionality that serves as the primary unit of reuse in the CLR. Effectively, assemblies establish a name scope for types. (from the SDK)

The assembly is the unit of access to resources within the runtime. Assemblies also contain metadata, also known as the assembly's *manifest*. The manifest makes assemblies self-describing, as component metadata does for components. It exports a set of types that are exported from the components within it. A reference to a Type is scoped by the identity of the assembly in which it is present. When the runtime tries to access a Type, it must query the assembly in which it is scoped. This allows a developer to use the assembly metadata to control how the assembly responds.

Some of the types within an assembly will be private to the assembly, so it specifies which of its resources are public and which are private. The assembly can control the mapping of an internal implementation on to the external reference of a resource, which allows the developer to alter the packaging of the types while keeping the external representation of the assembly the same. For exported types, the assembly can also supply runtime configuration information. The assembly metadata can also give version binding rules for its resources, by specifying the specific version of an external assembly that is needed by a reference from one of its resources. These rules are enforced by the runtime, although the developer can override them (see section on version control for more details).

6.6 Java and multiple languages

Although .NET was built around the idea of using multiple languages, it is not the only platform to support them. The Java virtual machine (JVM) does support languages other than Java itself. The multiple language capabilities of Java are not one of its strengths however. This is basically because Sun supports only the Java language, as it sees it as being integral to the platform. Any additional language support is provided by independently motivated development teams, although as the JVM is specifically targeted at the Java language any other languages using it incur a performance penalty over Java. This is commented on by Per Bothner (in Udell, 2000): "Well, language neutrality was clearly not a JVM goal. … Some features are harder to implement, especially efficiently."

> However, any language with functionality that can be expressed in terms of a valid class file can be hosted by the JVM. Attracted by a generally available, machine-independent

> platform, implementors of other languages are turning to the JVM as a delivery vehicle for their languages. (Java virtual machine specification, p.3)

This statement shows that while Sun have allowed for other languages to target the JVM, they see it as a tool to be used by them - a delivery vehicle. This is opposed to the .NET viewpoint: that a language becomes integrated with .NET as another tool for use by .NET developers. This lack of integration means that components written in one language can not be used with another language that uses the JVM, for example.

The JVM acts as a generalised object model for a language, however the language must be ported to it first. JVM native objects created in one language can be used in another if both languages are properly supported, and non-VM native objects are not supported. The compatibility of two languages and the JVM depends on the teams that ported each language, and so it becomes much more likely that bugs will occur between them. There is a significant problem, in that for the JVM there is no centralised point of contact for multiple language programming. The JVM lacks a common language element such as MSIL and metadata, along with the other aspects of the common runtime which enforce boundaries between the languages. Again, individual coding teams are seeking to rectify this - for example Per Bothner generalising the intermediate representation from GNU Kawa (see Bothner, 1998) - but this also adds another link in a lengthening chain of developers.

While it is becoming possible to use multiple languages with the JVM, and will no doubt become more complete with time, to do so requires the unification of the work of several independent developers (see Tolksdorf, 2001). This makes the JVM approach to multiple languages a viable one only for small, self-contained teams or possibly for large open-source development situations where the developers are used to some degree of configuration. For the group of open source programmers who do not wish to consider a Microsoft product, this is the only alternative (for a partisan view see Kuhn, 2001). A simpler use for alternate languages with the JVM is for scripting and customising applications and this is where it is most useful. However, for enterprise applications and large development teams (especially those with many teams co-operating to produce the final application) producing tight code, these approaches are very inefficient. The multiple language capabilities of the JVM are examined in more depth in Sessions, 2001. For these situations, the different aspects of a common language environment need to be unified and with a common point of contact (i.e., support) to remove the complexities of maintaining this environment from the developers. This is the situation that .NET was designed for. This perhaps explains Sun's reluctance to support multiple languages: it takes a lot of work to manage proper interoperation, and Sun are beginning to focus more on enterprise applications for Java (as is shown by the constant expansion of the J2EE platform). As the JVM supports a greater number of platforms than .NET it will remain the only choice for those who wish to support all of them for the time being, although Microsoft will no doubt try and port .NET on to these platforms as quickly as possible.

Other developers are also working on open source development tools for .NET on other platforms (Taft, 2002). It may be that many developers will choose to run both systems side by side, as discussed by Adhikari (2001).

7 Testing

7.1 Introduction

As Java has had time to mature, where .NET has not, it would not be fair to directly compare their performance. This is perhaps most applicable with the performance of the garbage collector, which in Java must be assumed to be more finely tuned than .NET's. Initial impressions seem to show .NET applications as being more responsive than Java applications. This should be disregarded, however, as this is on a Windows system and Microsoft have a considerable advantage here. When (and if) .NET runtimes appear for other operating systems then the relative performances can be evaluated more accurately (although the performance will vary with every version of the runtime). With this in mind, the criteria chosen for the evaluation of .NET and Java relative to each other (based on instead on usage) were as described in Table 7.1.

Criterion	**Reason for inclusion**
Ease of programming	This is the primary factor that affects productivity with a programming library.
Features	To a certain extent the features of the systems are unimportant as additional libraries can be obtained. However, there are a certain amount of features that should be considered necessary in a modern programming environment (such as comprehensive database access).
Flexibility	If the system can not be easily adapted to fit the circumstances in which it is to be used, it is a serious impediment to productivity.
Group working factors:	
Units of reuse	The unit of reuse is vitally important in a group environment.
Interoperability for multiple programmers	If a team can not work together on a project efficiently and easily, it will take far longer.
Documentation abilities	Programmers need to be able to quickly understand other code.

Table 7.1: The criteria used to evaluate .NET and Java.

It was decided that, with more known about Java, it would be more effective to attempt to produce the application in .NET and then to replicate the same functionality in Java. In that way, the ease of use of .NET could be evaluated using Java as a baseline for the comparison. This ensured that functionality was not added in the Java version that was not possible or irrelevant in .NET.

7.2 The Application

With both .NET and Java focused on networking and distributed applications, the test application needed to contain both a local and remote component. With both systems being capable of producing dynamic web content, the ideal application would contain both a local and remote section. An obvious form for the application to take to fit in all of the required functionality is an application for use either over the Internet or from a client which can be updated with the latest information. In this manner some online shops now allow their price database to be downloaded for comparison to the High Street, for example.

7.3 The class library PMArticle

It was decided to use a common data structure at the core. This took the form of the class library PMArticle (see Figure 7.1). This allowed a separate unit of reuse to be tested. A database was not used for this data structure partly so that the multiple language infrastructure of .NET could be demonstrated, and partly due to unfamiliarity with the Java database routines. This was a grievous error in terms of application design, and caused severe problems later on. These problems were masked by the initial inability to get server-based projects to work in .NET (probably a bug in beta 1, beta 2 fixed the problem). It was decided instead to attempt to transfer a copy of the data file serialised from this data structure to the client if an update was required, and to update the copy on the server using a web application. In .NET it was considered using web services as the base for this functionality. For the database component of the application it was decided to instead have a store of user IDs and passwords to simulate accessing an employee database.

The instance of the ArticleSystem type is the container for the associated types. If it is serialised the web of objects is serialised along with it allowing the overall data structure to be easily stored. The contents of the PMArticle package are shown in Figure 7.1 and the types are described as follows:

- ArticleSystem: Represents the overall system. The root of the object hierarchy.
- Topic: The topic is a demarcation of the articles. There may just be one topic in the system, or there may be many depending on the size of the system. Articles may belong to several topics if the topics represent different views of the same overall collection of articles.
- Entry: Represents an entry in the system. Combines the content with the description, comments etc.
- Series: A series is an entry which itself contains multiple entries. This represents a physical series of articles: i.e., in the ECMA submission for the CLI specification (see the chapter on the common language runtime) the document is split into several logical partitions.

- Part: Represents a part of a multi-part series. Basically adds a description of the relationship of the entry to the series. This only needs to be used by the class "Series" so can be an inner class or equivalent.
- Article: Represents the physical article. In a full application it would be an interface with display methods to allow various different types of article. In this it will merely say what type of article it is.
- TextArticle: Represents a purely textual article. Returns a loaded file.
- Comment: An abstract superclass allowing different types of comment to be added to an entry. (For example, author description, staff review, review body recommendations, etc.)
- Review: Represents a basic review of the entry.
- Description: Represents a basic description of the entry.

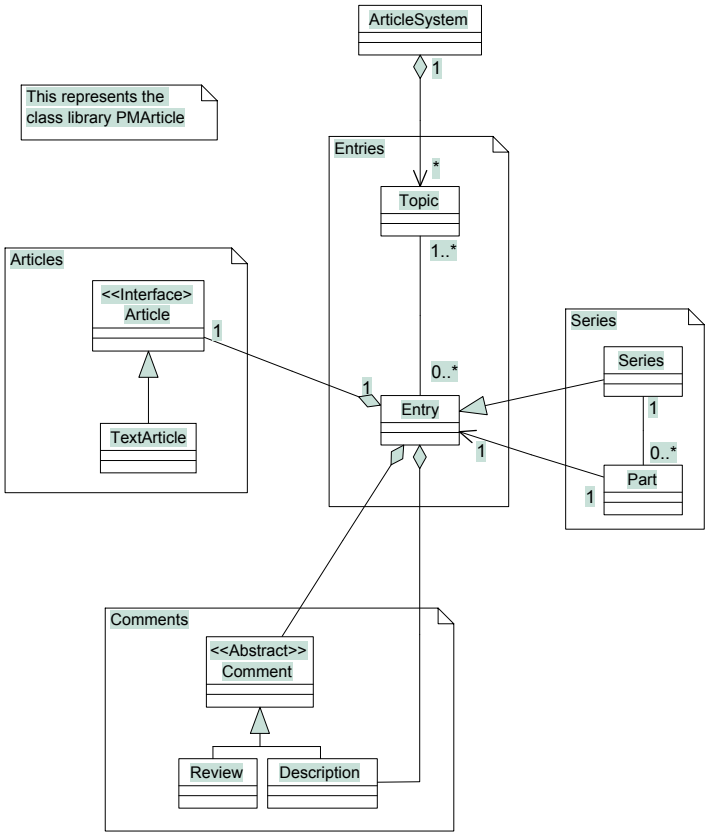


Figure 7.7 - UML diagram of package PMArticle.

7.4 The Browser

The browser was the client part of the application (or the standalone part, depending upon the implementation of the data updating system). It was intended (as the name implies) to be used for viewing the data structure only. The design was the simplest possible: to be able to open (de-serialise) a data file and display the contents, allowing each article to be viewed and the related information (i.e., the description) to be displayed. As an extra

function, a dialog box was also to be added that could import a data file from the server, display it and serialise it to disk. The browser was therefore a test of the standalone application type and the user interface classes as well as a basic test of the networking capabilities of the systems. The interface (in Visual Studio .NET design mode) can be seen in Figure 7.2, and consists of a tree view to select the articles and various fields to contain the other information. In .NET the system for controlling the layout of components is not as easy to use as Java's (in particular there is no choice of layout managers), although it is still functional. There were still some bugs in the system as of beta 2, but these should have been removed from the final version.

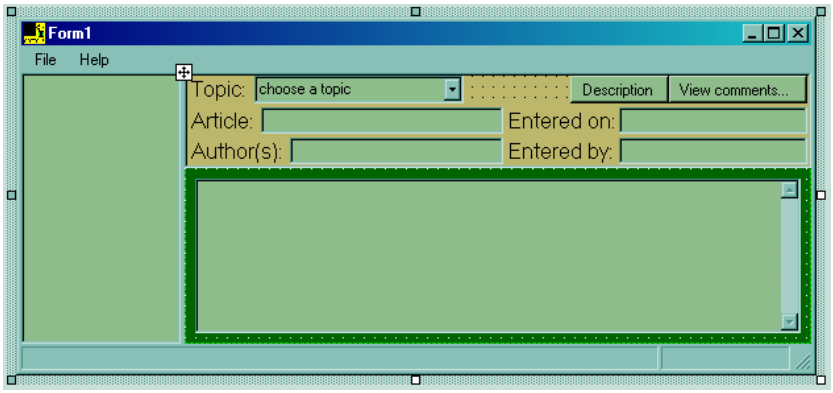


Figure 7.8 - The browser application interface.

The update dialog box was not fully implemented for two reasons. The first is that in .NET it was decided to use this as a test of web services and to get the data file from a specialised web service. This involved serialising the data file to a SOAP stream and returning it across the Internet. The returned data file was not successfully assigned back to a PMArticle object (due to a casting error). The most likely explanation for this is that the library containing PMArticle needs to be added to the global assembly cache (it is intended to do this but has not yet been tested). This could also be due to an issue with SOAP encoding, although not enough is currently known about this protocol by the author. The second reason the update dialog box was not completely implemented is that the PMArticle library did not support this: to test the implementation of interfaces, the article classes were designed to be interchangeable. As a result of this the text of the article was to be stored in a separate text file. This would then have needed to be transferred along with the serialised data file. This could be solved by moving the text of the article in to the article object, but then this does not allow other types of file (i.e., spreadsheets) from being used. The simplest solution would be to have a more fully featured server-side application and to connect to it to download the directory containing the files.

7.5 Dynamic web site

It was decided that the web front end for the application should also be responsible for adding items to the article system. This took the form of ASP.NET pages and JavaServer pages respectively. It was briefly considered using web services in .NET to provide this functionality with the web site being merely a front end, however this is not easy to replicate in Java so that idea was rejected. The .NET version was created with a reasonable amount of ease. The Java version was not so easy, as JavaServer pages proved to be far more complicated to create. This was undoubtedly exacerbated by using JBuilder 3 which has no support for JavaServer pages, as opposed to Visual Studio .NET which has excellent support for ASP.NET pages.

Web Services in .NET proved to be easy to program, and easy to use from within the client application. If the application had used a database as the central data store then in .NET web services could have been used to provide all of the application functionality. This would have made creating multiple front ends very quick. If this was to be used across a large company, this would allow an Intranet version, a remote version and a mobile version to be created with very little extra overhead, for example. In all of these cases, it would be the same core routines at work.

7.6 Examining the testing criteria

Referring back to Table 7.1, the testing criteria can now be evaluated. The first criterion is the ease of programming of the systems. This is heavily effected by the development environment, and especially the documentation. The main reference documentation used in each case was that supplied with the platform SDK to allow a more useful comparison. Even with no prior knowledge of .NET, it was easier to program than Java. This was due to the more coherent API and much greater integration of type and member descriptions with the editor in Visual Studio.

The features of both systems have been discussed in the preceding sections. While there are some areas where Java is more comprehensive, the numerous revisions and extensions have left it feeling fragmented. Microsoft took the opportunity to include those features that were added to Java in the initial version of .NET. This integration greatly benefits it. For example, the database types of .NET use a data storage type that is shared with user interface types. The Java database types do not share this. While it varies according to the feature, in general the greater coherence of the .NET programming libraries is one of the most notable advantages .NET has over Java.

It takes a greater amount of time to test the flexibility of the systems, so it was not possible to draw complete conclusions on them. Both systems are designed to be flexible. Java has a much wider range of runtime environments available, so remains the most flexible in this respect.

The units of reuse in .NET are a great improvement over Java. The main advantage to large development teams is the strong version control support, which is built in at the level of the operating system (for Windows systems,

probably just at the runtime level when ported to other operating systems). The ability to then store all the versions of a class library in the global assembly is a great improvement, particularly when applied to the .NET Framework libraries: an application will still use the version it was compiled and tested with.

The support for interoperability for multiple programmers is improved in .NET from this greater support for units of reuse. The ability to program in multiple languages and allow different parts of the team to use whichever language is best for each task could be a major advantage. Java has more support for different platforms and development tools that are available on more than one platform (i.e., Forte) and so it has an advantage for programming teams that use different environments. For example, open source development is more effective if each programmer does not have to use the same environment. When either system can offer both of these things reliably it will gain a large advantage over the other.

7.7 Overall experiences

The simplest conclusion that could be drawn during testing was the experience of returning to Java after using .NET to create a similar component: Java, the system initially known by the author, seemed far more complicated and difficult than it had previously to using .NET. At the time of writing, similar views are being aired in the first reviews of Visual Studio .NET (For example, Collingbourne & Hogan, 2002). As stated in the previous section, the decision was made to attempt to produce the sample application in .NET first to avoid adding functionality in Java that was prohibitively complicated to then add in .NET. In effect, the reverse of this in fact occurred. Many of the features of Java that were added after the initial release are more complicated than they could be and so were far harder to implement.

8 Conclusions

Table 8.1 defines some of the main attributes of the frameworks, such as reusability, compatibility, documentation, portability, and so on. It can be seen that .NET scores well in most areas, especially in team working, reusability, documentation, and speed of code, whereas Java has advantages in costs and compatibility. In terms of reusability, if Javabeans are seen an additional component, Java scores quite badly, as compared with .NET. Another area that .NET does well is with team working, as for it has improved documentation and units of reuse. Multiple versions of assemblies can coexist, and so on, so that part of a team can update a new version of a component when they want, rather than when it is finished.

Java still wins on costs, but, with big development teams, it is more irrelevant as maintaining Java is more tricky. It is also best for compatibility, as it is available for a wider range of systems. This may reduce in the future, as .NET is created as open-source.

While applications usually get more stable as they get are updated, with programming languages/systems the reverse is often true due to new features being added that add functionality but can complicate the language. Unfortunately in some areas Java now acts as an example of this. There have been some radical additions and modifications to Java over its short lifetime, with the result that Java now seems fragmented. For example, the switch from the AWT to the Swing user interface components can cause problems for developers if they are not sure that the target runtime is up to date. The many deprecated types and methods within Java can also cause incompatibilities if reusing old code. There are some features of Java that were not included until recent rehashes of Java, or are still regarded as add-ons (for example, JSPs and JDBC). This constant redefining of Java makes life difficult for developers who have to keep up with the changes while trying to use the most current version. Another strange quirk of Java is the value types (i.e., integers, floats and so on). These are not objects, and so wrapper objects must be used when operations need to be performed on them (in .NET, all value types are objects). This does not comply with the object-oriented model, and makes these operations much more inefficient for the developer. There are other cases of wrapper classes being needed, which suggests the component model had not been entirely accepted by Sun. Javabeans are much more compliant to the component model, but these are another revision to the Java standard.

Whether .NET will also undergo a similar metamorphosis only time will tell, but it appears to be much more feature-complete and have more in the way of initial design. Considering it is the brand new system, it appears much more mature than Java. In this way it is reminiscent of the first impression that many people had when it was announced: a competitor to Java, designed by Microsoft to beat Java in all respects. Therein lies the main disadvantage to .NET: *designed by Microsoft*. There are many developers who are biased against Microsoft and consequently believe that .NET is just another attempt to monopolise every area of computer software. The relative performance of the systems cannot be accurately measured until

.NET has had a time to stabilise, but for this reason it is also assumed that Java has a performance advantage at present. This assumption may be erroneous as, even in beta form, .NET applications appear to match Java. This document focused more on .NET than was initially planned for, with the sections on Java reduced compared to the more voluminous .NET explanations. While this is partly due to the similarities between the systems to avoid repetition, it is also in great part indicative of the superior documentation provided with the SDK. This was more coherent than the Java equivalent which, in most cases, reduced the use of the Java documentation to reference and comparison while the .NET SDK was used to learn the basics. The other factor that curtailed the explanation of some features was the lack of space. This is due to attempting to go into slightly too much detail (or conversely attempting to explain too simply).

Ultimately the conclusion was drawn that .NET is an improvement over Java. The basic system is more comprehensive and easier to use.

	Score (1 – Poor, 10 – Excellent)		
Attribute	**.NET**	**Java**	**Comment**
Reuse	8	6	If looked from the point of view that Javabeans are an additional component, Java scores quite badly.
Compatibility	4	9	
Portability	8	7	
Current developer experience	10	4	
Documentation	9	6	
Language syntax	8	7	Assuming C#
Integrated development systems	9	7	
Costs	6	8	Java wins on that one, no problem. With big development teams it is more irrelevant as maintaining Java is more tricky.
Availability of code	8	8	
Team working	10	5	.NET scores well in this area due to the documentation and units of reuse. Multiple versions of assemblies can coexist, and so on. so that part of a team can update to a new version of a component when they want, rather than when it is finished
Speed of code	8	4	.NET is optimised for a Windows environment.
Embedded systems	6	6	Both systems support them, Microsoft is making a push in this direction.
Version control, integration with Software Engineering tools	8	8	Version control is very good on .NET. Both systems support from third party applications

Table 8.1: Comparisons between .NET and Java

9 Appendix 1: Research Proposal

9.1 Student details

Name: Paul McIntyre **Matriculation no:** 00400017

BSc or MSc: MSc **Full-time/Part-time:** Full-time

Telephone no: 07712 580 804 **(day)** 07712 580 804 **(eve)**

Project outline

Proposed title: Comparison of .NET with Java

Brief description of research area (150 words MAX)

The newest version of Microsoft's programming suite, Visual Studio, incorporates a new foundation known as the .NET framework. It is designed as a way of removing the device dependence of the programs that use it (similar to Sun's Java). It also serves to increase the interoperability of the different languages that support .NET. This system is likely to become a direct competitor to Java as it seeks to solve many of the same problems. While it does have similarities to Java, it also has some marked differences. It remains to be seen whose approach will be more successful. The differences in specification from Java mean the systems are going to have different strengths and weaknesses. They are also likely to suit different types of programmers. What niche .NET will fill will not be known for some time yet.

For software development projects please state:

Hardware you will use: A standard home PC

Software you will use: Microsoft Visual Studio .NET beta and Borland JBuilder 3 Professional (Java compiler)

State one or two main questions your research will address

What are the fundamental differences in both systems, and what is the difference in their focus?

How well do they perform at different types of task, and how flexible are they?

Give THREE key references to published work in your research area:

(General background) Hogan, D. "Visual Studio.NET Beta 1 preview" PC Plus, issue 176 [Future Publishing]

(More specialised) Microsoft ".NET Framework Developer's Guide" (beta) [WWW Document] URL http://msdn.microsoft.com/library/default.asp?URL=/library/dotnet/cpguide/cpguide_start.htm

(More specialised) Banerjee, A. (2001,May 15) "Will Java and .NET Framework Co-exist?" [WWW Document] URL http://www.c-sharpcorner.com/Articles/DotNetforJava2.asp

Please complete the following project outline, using the emboldened text as a framework.

The idea for this research arose from.... A magazine article previewing Visual Studio.NET

The aims of the project are as follows: To get an understanding for .NET and determine whether it will improve upon Java, and where its niche will lie.

The software development/design work/other deliverable of the project will be: A comparison of the features and abilities of the systems, a series of sample programs using each system (designed to accomplish the same tasks), a comparison of programs using the different languages within .NET and conclusions based on the relative performance of each language.

The project will involve the following research/field work/experimentation/evaluation:

Evaluation of the relative performance and efficiency of each language using .NET and Java at each of the tests undertaken. Each test will look at a different aspect of the systems. Evaluation of other features of the programs. An evaluation of the language support of .NET and any issues arising from the intermixing of languages within a project.

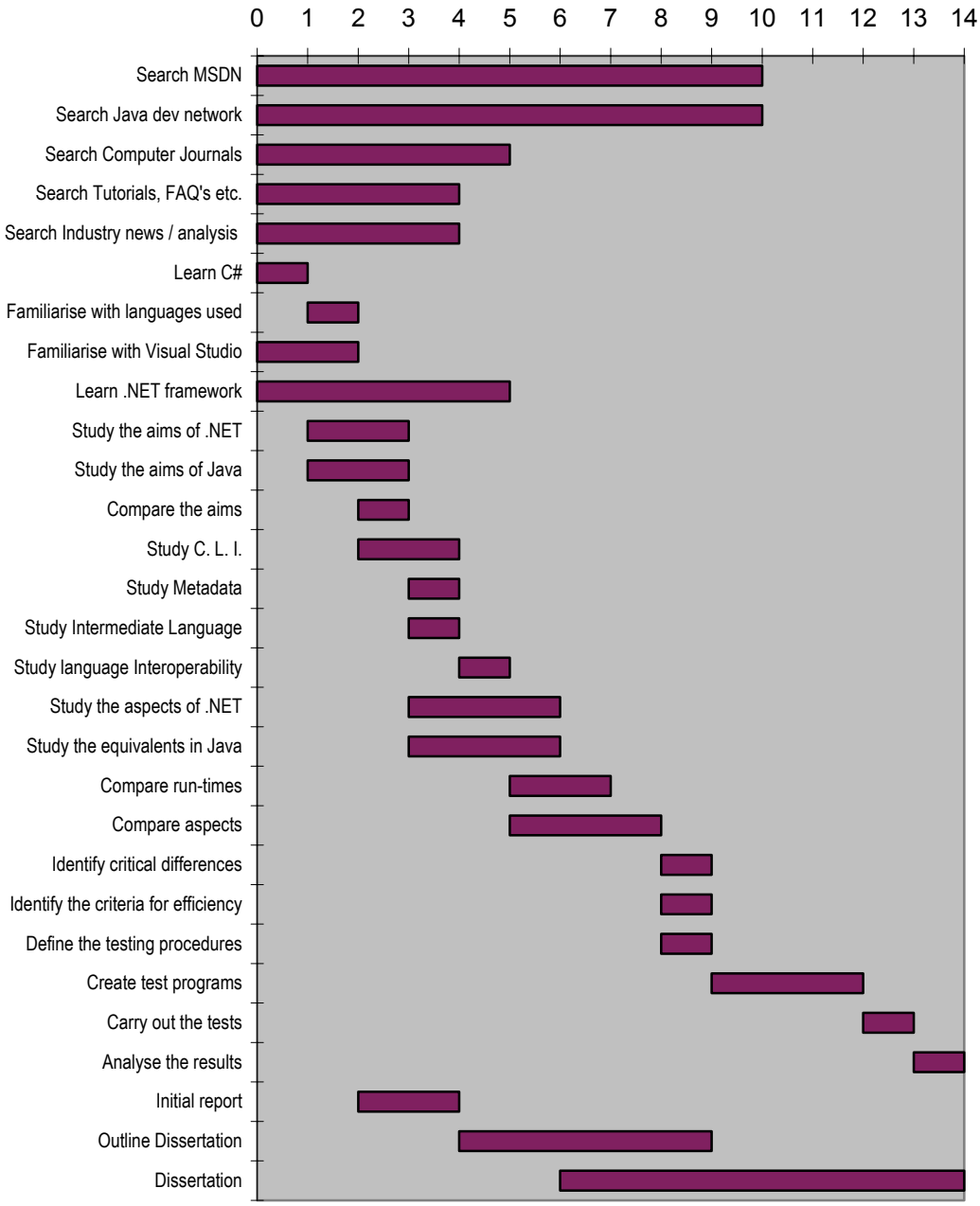
The project is being undertaken in collaboration with... Not known at present

The main difficulty foreseen is .... At present .NET is still a beta product, so the features may be changed and performance in specific areas may be subject to change. The evaluation must bear in mind potential bugs and try to compensate for them to give a better insight into the finished language.

Appendix 2: Work Plan

The most obvious change that was made to the work plan was the change from full time to part time in week five. The plan remained unchanged, except in that the axes were made correspondingly longer. The testing phase was also made integral with the programming phase.

There has been one major delay to the project. There was an incompatibility between Beta 1 of .NET and my computer that left me unable to create or work on “web projects” which delayed the complete start of the programming phase by two weeks. This was eventually solved only by the release of Beta 2 of .NET which worked immediately. At first, there were also numerous small delays caused by intermittent problems with my computer’s hard drive. Fortunately, I managed to recover my project files the first time it happened and they are now stored primarily on a zip disk, which rendered subsequent relapses little more than an irritation. The change to part time work essentially removed this problem as I could then fix it in my own time.


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Search MSDN
Search Java dev network
Search Computer Journals
Search Tutorials, FAQ's etc.
Search Industry news / analysis
Learn C#
Familiarise with languages used
Familiarise with Visual Studio
Learn .NET framework
Study the aims of .NET
Study the aims of Java
Compare the aims
Study C. L. I.
Study Metadata
Study Intermediate Language
Study language Interoperability
Study the aspects of .NET
Study the equivalents in Java
Compare run-times
Compare aspects
Identify critical differences
Identify the criteria for efficiency
Define the testing procedures
Create test programs
Carry out the tests
Analyse the results
Initial report
Outline Dissertation
Dissertation

10 References

Adhikari, R. 2001. Java and .NET can live together, *Application Development Trends*, **8**/12, 31-33.

Anderson, T. 2002. XML is top of the class, *Application Development Advisor*, **6**/4, 30-37.

Bothner, P. 1998. Kawa Compiling Dynamic Languages to the Java VM. URL: http://citeseer.nj.nec.com/bothner98kawa.html [August 2001]

Campione, M., Walrath, K., 1998. *The Java Tutorial: Second Edition*. London: Addison Wesley

Campione, M., Walrath, K., Huml, A., et al. 1998. *The Java Tutorial Continued: The Rest of the JDK*. London: Addison Wesley

Chien, E. 2001. The effects of Microsoft .NET on malicious threats. In Virus Bulletin, *Proceedings of the Eleventh Virus Bulletin International Conference*. Abingdon, UK: Virus Bulletin.

Collingbourne, H., Hogan, D., April 2002, On Test: Microsoft Visual Studio .NET, *PC Plus*, Issue **188**

Databases Journal. 2002. The arrival of "Web services" on the J2EE platform, *Databases Journal*, **35**, 41-43.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. London: Addison Wesley

Gordon, A. D., & Syme, D. 2001. Typing a multi-language intermediate code, *Sigplan Notices (Acm Special Interest Group on Programming Languages)*, **36**/3, 248-260.

Grimes, R., April 2001, C++ Attributes: Make COM Programming a Breeze with New Feature in Visual Studio .NET, *MSDN Magazine*, 04/**2001**

Gunton, N. 2001. SOAP: simplifying distributed development, *Dr. Dobb's Journal*, **26**/9, 89-95.

Kadhim, B. July 2000. COBOL for the Microsoft .NET Framework. URL: http://msdn.microsoft.com/library/en-us/dndotnet/html/pdc_cobol.asp [July 2001]

Kennedy, A., & Syme, D. 2001. Design and implementation of generics for the .NET Common Language Runtime, *Sigplan Notices (Acm Special Interest Group on Programming Languages)*, **36**/5, 1-12.

Kuhn, B. M. February 2001. JVM to .NET: I'm Not Dead Yet! URL: http://www.onjava.com/pub/a/onjava/2001/02/15/jvm.html [July 2001]

Microsoft. October 2000. C# and CLI specification documents. URL: http://msdn.microsoft.com/net/ecma/ [August 2001]

Middlemiss, J. 2002. IT challenge: Web services, *Wall Street & Technology*, **20**/8, 40-43.

Mingins, C., & Nicoloudis, N. 2001. .NET: a new component-oriented programming platform, *Journal of Object-Oriented Programming*, **14**/4, 44-81.

Pietrek, M., October 2000, Avoiding DLL Hell: Introducing Application Metadata in the Microsoft .NET Framework, *MSDN Magazine*, 10/**2000**

Pratschner, S. September 2000. Simplifying Deployment and Solving DLL Hell with the .NET Framework URL: http://msdn.microsoft.com/library/techart/dplywithnet.htm [July 2001]

Rapaport, L. 2002. Java vs. .NET: Which route to web services?, *Transform Magazine*, **11**/5, 56.

Reilly, D. 2001. Threading and the .NET framework., *Dr. Dobb's Journal*, **26**/8, 30-38.

Richter, J., November 2000, Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework, *MSDN Magazine*, 11/**2000**

Richter, J., December 2000, Garbage Collection - Part 2: Automatic Memory Management in the Microsoft .NET Framework, *MSDN Magazine*, 12/**2000**

Scanlon, J. 2001. A tour of managed C++, *Software Development*, **9**/11, 29-32.

Sells, C., July 2001, Visual Studio .NET: Managed Extensions Bring .NET CLR Support to C++, *MSDN Magazine*, 07/**2001**

Sessions, R. June 2001. Is Java language neutral? URL: http://www.objectwatch.com/issue_33.htm [August 2001]

Simon, R. et al. July 2000. Eiffel on the Web: Integrating Eiffel Systems into the Microsoft .NET Framework. URL: http://msdn.microsoft.com/library/en-us/dndotnet/html/pdc_eiffel.asp [July 2001]

Taft, D. 2002. Mono project opens up dot-Net, *IT Week*, **5**/30, 15.

Templeman, J. 2001. ABC, *Developer Network Journal*, **23**, 26-30.

Tolksdorf, R. 2001. Programming Languages for the Java Virtual Machine. URL: http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html [August 2001]

Udell, J. December 2000. JVM And CLR. URL: http://www.byte.com/documents/s=505/BYT20001214S0006/index.htm [June 2001]

Watkins, D. October 2000. Handling Language Interoperability with the Microsoft .NET Framework URL: http://msdn.microsoft.com/library/en-us/dndotnet/html/interopdotnet.asp [June 2001]

Weiss, A. 2001. Microsoft's .NET: platform in the clouds, *Networker*, **5**/4, 26-31