Session 4. Relationships

In this session we our focus shifts from reviewing the basic programming concepts to object-oriented concepts and their implementation in the C# programming language. complete our review of the pre-requisite content covering arrays, a more detailed review of methods, and scope among other topics. We also examine some of the more obscure/advanced topics, which may or may not have been covered in pre-requisite studies, but are common in object-oriented programming such as method overloading.

Session Objectives

In this session, you will learn:

- The connection between object relationships and class relationships, their differences, and how to implement them in your programs;
- How dynamic memory is used by programs and the function of the garbage collector;
- How constructors and destructors are used to initialise new objects and clean up removed objects;
- How to copy objects successfully and how to solve the shallow copy problem;
- How to apply delegates, indexers and the ToString method in your programs; and
- How to prepare formatted reports.

Unit Learning Outcomes

- *ULO 1. Apply object-oriented concepts including abstraction, encapsulation, inheritance, and polymorphism.*
  We continue developing our knowledge of object-oriented concepts in this session, focusing on the relationships between classes and objects.
- *ULO 2. Solve programming problems using object-oriented techniques and the C# programming language.*
  We continue developing our understanding of object-oriented techniques and how they are applied in the C# programming language in this session.
- *ULO 3. Modify an existing object-oriented application to satisfy new functional requirements.*
  We will not be directly addressing this ULO, however understanding of the C# programming language will contribute to the skills required for this outcome later.
- *ULO 4. Construct object-oriented designs and express them using standard UML notation.*
  We will not be directly addressing this ULO, however the knowledge gained from this session is directly related to the design of object-oriented applications.

Required Reading

Additional reading is required from the textbook Deitel, P., and Deitel, H., "Visual C# 2012: How to Program", Fifth Edition, Pearson, 2014, as follows:

Required Reading: Chapter 10, Sections 10.1-10.9, Section 10.11, and Section 10.13
This chapter examines most of the remaining concepts needed for developing classes before we examine the key object-oriented concepts of inheritance and polymorphism.

Required Tasks

The following tasks are mandatory for this week:

- Task 4.1. Object Creation and Linkage
- Task 4.2. Report Writing

4.1. Introduction

In the previous sessions we have completed a review of the fundamental programming concepts. Our focus now shifts to object-oriented concepts, beginning in this session by examining the concept of a relationship.

Previously we described object-oriented applications as consisting of objects that interact with each other. These interactions occur because there is a relationship between these objects. In the same way that the attributes and methods of an object are defined in a class, so are the relationships between objects defined by relationships between the classes. There are three class relationships we consider in this session: association, aggregation, and composition. The final class relationship, inheritance, is examined in Session 5.

As we will be creating and destroying objects more regularly, we also consider a number of related concepts. Dynamic memory management addresses how memory is used in the creation and destruction of objects at run-time. We then re-examine constructors, considering the different types of constructors, introducing the copy constructor that allows us to copy objects. Destructors are also examined, which are similar to constructors, except that instead of being automatically invoked when an object is created a destructor is automatically when an object is destroyed.

Delegates are another data type, which allow us to store references to methods, similar to how a variable can store a reference to an object. Indexer properties allow us to use the indexer operator (`[]`) with our classes, just like the indexer for an array. We then examine the `ToString()` method, which is used to obtain a `string` representation for any object. Namespaces are then covered, which affect the names of data types (class, etc.) in our program code in the same way as sub-directories affect files on a disk – dividing those names into sections to make them easier to manage. We finish the session by examining report writing, referring to the use of formatted output concepts to build reports for display on a screen or for printing.

4.2. Objects and Class Relationships

Object-oriented applications consist of a collection of interacting objects. For two objects to interact with each other, there must a relationship established between them, i.e., they must know about each other in some way, known as an object relationship. Recall that objects are instantiated from classes and that classes act as templates for objects. In the same way, an object relationship is an instance of class relationship. As such, the set of potential object relationships that occur between objects are defined by the class relationships between the respective classes of those objects. Importantly, not all class relationships result in an object relationship: the inheritance relationship, examined in Session 5, does not define an object relationship.

4.2.1. Object Relationships

An object relationship is known as a link, which defines a relationship whereby one object can request the services of another object. Links are usually unidirectional, meaning that requests can only go from one object to the other, but not vice versa. Note however that data can still pass in both directions, whereby the second object returns a result to the first object. Links can also be bidirectional, where the objects can request services from each other. The direction of the link is often referred to as navigability, i.e., whether one object can “navigate”, or request the services of, the other object.

For example, consider representing an order of merchandise for an online store’s web application. Many objects would be used to represent each item available to purchase from the store, and another object would represent the order itself. In order to determine the contents of the order, it would be necessary to be able to navigate from the order object to the individual items in that order, i.e., to determine the items that have been ordered. Although less likely to occur, it may also be useful to be able to navigate from individual items to each of the orders that they appear in, e.g., to be able to quickly locate all the outstanding orders when new stock arrives, or to determine the history of sales/orders for an item.

Navigability is clearly seen in a C# program through the use of references. Where one class contains a reference to another class, the objects of that class will be able to navigate to objects of the second class, i.e., the object of the first class could invoke the methods defined in the object of the second class. However the object of the second class would not be able to invoke the methods defined for the first object, unless it also had a reference to the object of the first class. Regardless, as discussed in Section 3.4 data can pass in both directions during a method call through a combination of parameters and return values.

4.2.2. Class Relationships

Fundamentally, there are only two types of class relationships: associations and inheritance. Inheritance is examined in Session 5. An association is a general relationship

between two classes, whereby one class is able to performs actions on objects of the other class, or the objects of one class are acted upon by objects of another class. Associations are usually bi-directional, meaning objects of each class can find objects of the other class, i.e., peer-to-peer. An association can be further refined into an aggregation, which in turn can be refined further into composition.

An aggregation relationship is a tighter form of association representing a whole/part hierarchy, whereby one object is part of a larger whole. Unlike an association, in an aggregation relationship, the 'whole' object (the aggregate object) can usually navigate to the 'part' object, however it is unusual for the 'part' object to be able to navigate to the 'whole'. Importantly, note the use of the phrase 'part of', underlined above. This phrase can actually be used to test whether aggregation is reasonable or not, e.g., a Book is part of an Order is a reasonable statement, whereas an Picture is part of a Person is not reasonable. Another phrase that can be used is has a, e.g., a University has a number of Students.

Finally, composition is a tighter form again, whereby the 'part' object can only be part of one 'whole' object. The lifetimes of the objects in a composition relationship are said to be tightly coupled, meaning that the 'part' object is usually created and destroyed at the same time as the 'whole' object. This is unlike aggregation, where the lifetimes are loosely coupled and thus are created and destroyed independently. Aggregation also has the advantage that the 'part' object can be part of more than one 'whole', e.g., a Book can be part of (listed in) more than one Order.

4.2.3. Relationships in C#

When considering how to implement the above relationships in a programming language such as C#, the problem is somewhat simplified by the use of references, which define a unidirectional link from one object to another. Through this link, the first object (referred to as the client object) is able to invoke the methods of the second object (the supplier object). To establish such a link, there are two tasks that must be completed:

i. A variable must be declared in the client object to refer to the supplier object (a reference);
ii. The memory address of the supplier object must be stored in the client object's reference.

To explain how this is done, we will explore a simple example where objects of one class `Unit` has links to several objects of another class `Student`. The `Unit` and `Student` objects are created separately, and `Student` objects are then enrolled in/linked to the `Unit`.

We have already seen how to declare variables as required for task (i), and this is relatively straightforward. As for attributes (see Section 2.5), the variable should be declared as private, i.e.,

```
private Student _SingleStudent;
```

```
private Student [] _StudentArray = new Student[size];

private List<Student> _StudentList = new List<Student>();
```

Three examples are provided: `_SingleStudent` for where an individual `Student` object would be linked, or more appropriately for our example, `_StudentArray` (an array) and `_StudentList` (a `List` generic collection) for when multiple students would be linked (multiple students would be enrolled in a single unit). We will use the `_StudentList` object for this example.

The second task (ii) is more complex and will depend on how the application being developed has been designed. From the description of the problem above, the `Unit` and `Student` objects are created separately before `Student` objects are enrolled into (added to) the `Unit` object. Thus for this purpose, we could define an `EnrolStudent` method, as follows:

```
public void EnrolStudent(Student student)
{
    _StudentList.Add(student);
}
```

In the above method, the `Student` object has been passed in as a parameter which is then recorded in the `_StudentList` `List` object by the `Unit` class. Once the memory address is recorded, working with the `Student` objects stored in the list is the same as described previously in Section 3.2.

How the memory address is provided changes depending on the application and how it is designed, however in general one object must obtain the address of the other object. This can be achieved by:

- The first object creating the second object itself;
- The first object may retrieve the address of the second object from a third object (acting as a broker); or
- The first object is given the address of the second object as a parameter to its constructor, another method, or a property.

For a bidirectional link, the same approaches can be used, or the first object may provide the address itself[1].

By following the above guidelines, it is possible to implement both association and aggregation relationships. Both are implemented in the same way, and the only difference is whether the particular relationship being implemented needs to be bidirectional or not. Composition however introduces a new problem. As discussed in Section 4.2.2, the lifetimes of the part and whole object are usually tightly coupled under composition. For C#, the only way the lifetime between a part and whole object

[1] This is what happens when you add a control to a Form in C# (or VB.Net). In particular the control, such as a TextBox, is created and added to the Form's property 'Controls' (a collection). When a control is added, the Form provides its own address to the control which can be retrieved from the Control's property 'Parent', representing the Form the control appears on.

are truly linked is where the part object is a value type. Recall from Section 2.6 that value types in C# are either simple data types, which are usually discussed as attributes, or user-defined structures, which are not covered in this unit. Thus for our purposes composition relationships are unable to be properly implemented[2].

4.3. Dynamic Memory Management

Until now most of our programs have consisted of only one class (containing the Main method) and a few objects. Now that we have introduced relationships, and will begin to create many more objects, it is now an appropriate time to consider what happens when we create and destroy objects. Consider the following two variables being created in a method:

```
int accountNumber = 12345;              // from a simple type

Account account = new Account(54321);   // from a class
```

Both of these statements involve the creation of an object, an integer and an Account object, however even if these statements appear next to each other the objects will be stored in different locations in memory.

A simple memory map for a process is shown in see "Regions of memory in a process"[3]. The memory of a process is roughly divided into three regions: the text region (also sometimes called code region), the heap (also called data region), and the stack. The text region contains the code of your program, and hence is sometimes called the code region, written in machine code. The stack is used for the temporary storage of data relevant only to a method and/or the current statement, and also contains the information found in the call stack (previously mentioned as a debugger term in Section 2.10). Finally, the heap is a region of memory that is used for providing memory to a process on an as-needed basis at run-time, i.e., dynamically. Of these three memory regions, the text region is read only (code never changes), but the heap and call stack are read/write.

[2] It is possible to approximate the composition relationship by making sure that the memory address of the part object is never exposed outside the whole, however this relies on all future programmers maintaining the encapsulation which is not a reliable solution.

[3] This is an over-simplified model and does not consider operating system impacts, multi-threading, and so on. The study of how the memory of a process works, how it is constructed, and how it relates/maps to the physical memory of a computer is the subject of study in the unit SIT222 Operating Systems Concepts.

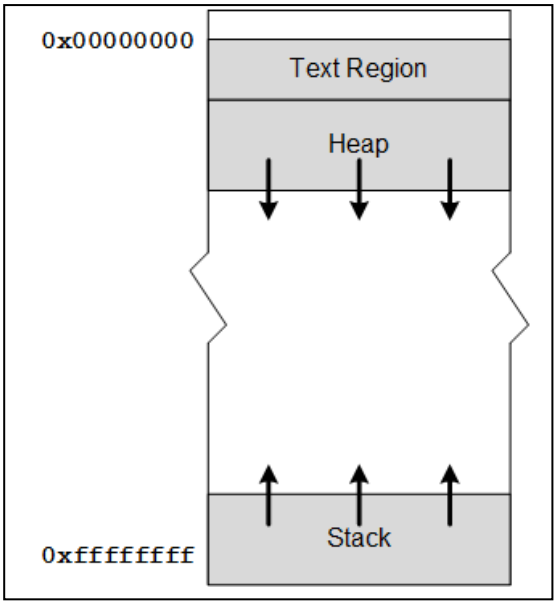

Figure 4.1: Regions of memory in a process

The first declaration shown above, the `int` variable, is known as an automatic variable. The name comes from the fact that memory is automatically allocated on the stack for the variable when the declaration statement is reached, and automatically deallocated when the variable goes out of scope.

The second declaration, the `Account` object, is allocated memory on the heap (dynamic memory allocation) which is then initialised as per the `Account` class' constructor method/s (see Section 4.4). Deallocating a dynamically allocated object is somewhat different however. Historically, most programming languages provided a keyword or a method to deallocate such objects. For example, to deallocate the account variable in C++, you would use:

```
delete account;
```

There is no such statement in C# however. Instead, deallocation of an object is achieved simply in C# by erasing all references to the object, i.e., by forgetting the object, which can be done by assigning a value of null (null effectively means 'nothing'), e.g.,

```
account = null;
```

Once an object is no longer referenced anywhere in a C# program, the memory will eventually be returned to the operating system by the garbage collector. The garbage collector is a program that is part of the Microsoft.Net framework that runs in the background. It checks to make sure that all dynamically allocated objects are still referenced by their programs. Any objects which are no longer referenced are deallocated and the memory they used is returned to the operating system for use by other processes.

The introduction of garbage collection solves one of the most common errors found in applications over the years – the memory leak[4]. A memory leak occurs where a program fails to deallocate some of its memory properly, and continues to allocate more memory as the program continues execution. Over time, progressively less memory is available to the program, which can eventually run out of memory. The garbage collector runs either when there is inadequate memory to allocate a new object or if the program is idle. The garbage collector uses a five step process[5]:

i. All objects in an application are examined to determine which objects are currently referenced and which are no longer referenced (referred to as an unreachable object);
ii. All unreachable objects are then examined to see if they require finalisation, i.e., if there is a destructor defined by their class (see Section 4.5);
iii. Unreachable objects that don't require finalisation are deallocated by compacting the heap (referenced objects are moved together to eliminate any unused gaps), and the remaining unused memory is returned to the system;
iv. The application is allowed to continue execution; and
v. Any unreachable objects that required finalisation are now finalised in parallel to the continued execution of the application.

4.4. Constructors: A Closer Look

We previously introduced constructors in Section 2.4, where we identified a constructor as a method invoked automatically when an object is created, used to prepare or initialise the object for use. Constructors are not compulsory, i.e., you do not need to write a constructor for every class you develop. However, recall the syntax used to create an object when invoking a constructor:

```
class_name variable_name = new class_name([parameter[, ...]]);
```

The parameter list indicated here matches the parameter list defined by any constructor that has been written. However consider the statement you would use to create an object from a class that does not have a constructor defined, which would be similar to the following:

```
SomeClass someObject = new SomeClass();
```

This syntax suggests that a constructor is being invoked, one without any parameters, even though we have not defined such a constructor. This is in fact what is happening – when we define a class without writing a constructor, the compiler automatically writes a constructor for our class, known as the default constructor. The default constructor has no parameters, nor any method body, i.e., it is equivalent to defining a constructor as follows:

```
class SomeClass
```

[4] Sometimes referred to more generally as a resource leak, memory being one of the many resources of a computer used by a process (disk, network, etc. being others).
[5] Sharp, J., Microsoft Visual C# 2013: Step by Step, Microsoft Press, 2013.

```
{
    public SomeClass()
    {
    }
    ...
}
```

Importantly, the default constructor is only generated if, and only if, there is no constructor defined by the programmer. Constructors written by a programmer are known as custom constructor, referring to the fact that they customise the default constructor behaviour. One of the most common forms of constructor written by programmers is one which requires no parameters, known as a parameter-less constructor[6].

Several constructors can also be defined for a class by exploiting method overloading, presented in Section 3.4.2. Consider the example code in Figure 4.2 for a simple `Account` class, such as for a bank account. Two constructors are shown, the first of which sets the account balance to a default zero value. The second constructor allows the balance to be initialised to any value specified during the object creation.

```
class Account
{
    private const decimal DEFAULT_BALANCE = 0.00M;

    public Account()
    {
        this._Balance = DEFAULT_BALANCE;
    }

    public Account(decimal openingBalance)
    {
        this._Balance = openingBalance;
    }

    private decimal _Balance;
    ...
}
```

Figure 4.2: Example of overloading custom constructors

Using these two constructors, objects can be created as follows:

```
Account firstAccount = new Account();

Account secondAccount = new Account(100.00M);
```

These two constructors shown in figure Figure 4.2 also introduce a new keyword in C#: the `this` keyword. The keyword `this` is used to refer to the instance (object) for which a method has been invoked, and can be used in any instance method (note that `this` cannot be used in static members). In the code block defining these constructors the use of the `this` keyword only clarifies that the `_Balance` variable is defined in the current class as an instance variable and is entirely optional, i.e., removing the `this` keyword makes no difference to the functionality.

[6] Note that the terms default constructor, custom constructor, and parameter-less constructor may have slightly different meanings when used in the context of other programming languages, e.g., C++.

Importantly, the code for these constructors can also be improved slightly through the introduction of what is known as an object initialiser, which allows one constructor to invoke another. To use an object initialiser, we must first reconsider the syntax for a constructor, which we expand slightly:

```
[access_modifier ]class  class_name
{
        [access_modifier ]class_name ([parameter[, ...]]) [initialiser]
        {
                method_body
        }
        ...
}
```

The object initialiser appears after the constructor's signature and has the following format:

```
: this ([parameter[, ...]])
```

You will notice in the constructors appearing in Figure 4.2 that the method bodies for the two constructors are practically identical, only changing the value assigned from a constant (the default opening balance) to a parameter (for custom opening balancing). Although very simple in this example, this is a case of duplicate code, which should always be avoided. We can eliminate this duplication by using an initialiser in the parameter-less constructor, as shown in Figure 4.3. In the updated definition, the initialiser is used to pass the default balance constant to the second constructor, which stores the zero value in the account's `_Balance` instance variable, providing equivalent functionality to the version shown in Figure 4.2.

```
class Account
{
    private const decimal DEFAULT_BALANCE = 0.00M;

    public Account() : this(DEFAULT_BALANCE)
    {
    }

    public Account(decimal openingBalance)
    {
        this._Balance = openingBalance;
    }

    private decimal _Balance;
    ...
}
```

Figure 4.3: Overloaded Custom Constructors Exploiting an Object Initialiser

4.4.1. Copying Objects with Copy Constructors

There is often a need in a program to copy an object. Consider an example where you are maintaining a collection of `Student` objects (say in a `List` collection) and want to provide the user with the ability to delete/modify students in the list. For this

purpose, you prepare a GUI window and present the list to the user, who then proceeds to modify the list. A problem now arises if the user decides they wish to abort their modifications to the list. If the original collection were used, there is no way to undo the user's modifications. However, if a copy of the collection was used, it is trivial to undo the changes by simply discarding the copy. Similarly, if the user confirmed their changes, the original could now be discarded and the (modified) copy used to replace it.

Constructors provide one mechanism through which we can create copies of objects, by defining what is known as a copy constructor. Copy constructors are used to initialise an object from the information stored in another object of the same type, i.e., to create a copy of the object. As such, copy constructors have a single parameter of the same type. The copy constructor for the above Account example could be written as follows:

```
public Account(Account source) : this(source._Balance)
{
}
```

Note that the constructor is able to access the instance variable `_Balance` in the source object. This appears to be a breech of the rules of encapsulation, given that `_Balance` is declared `private`. However this is not the case – encapsulation hides how an object works from other types of objects, not from objects that are the same type. Any code written in that class will understand and have knowledge of the encapsulated data in all instances.

It is also important to consider what would happen if you defined a class with only a copy constructor. As discussed earlier, the definition of any custom constructor, including a copy constructor, eliminates the (compiler generated) default constructor, therefore such a class would in fact be unusable as the only way to create an object of that class would be to copy another. There is no way to resolve this, thus you must be careful when defining a copy constructor that there is at least one other constructor defined. This is rarely a problem however, as classes that are complex enough to require a copy constructor will usually already have a number of constructors, and if required the simplest solution is just to define an empty parameter-less constructor, i.e.,

```
public Account()
{
}
```

4.4.2. The Shallow Copy Problem

Copying objects raises a new problem that must be solved: the shallow copy problem. The shallow copy problem occurs when an object is copied containing reference types, i.e., references to other objects, where only the memory address is copied, not the actual object stored at/referenced by that address. This can result in the situation illustrated in Figure 4.4. As shown in the figure, the `Person` object `John` has been copied from the `Person` object `Jane`, which contained references to two `Account` objects. Given

that only the addresses have been copied, `John` now shares the same `Account` objects with `Jane`.

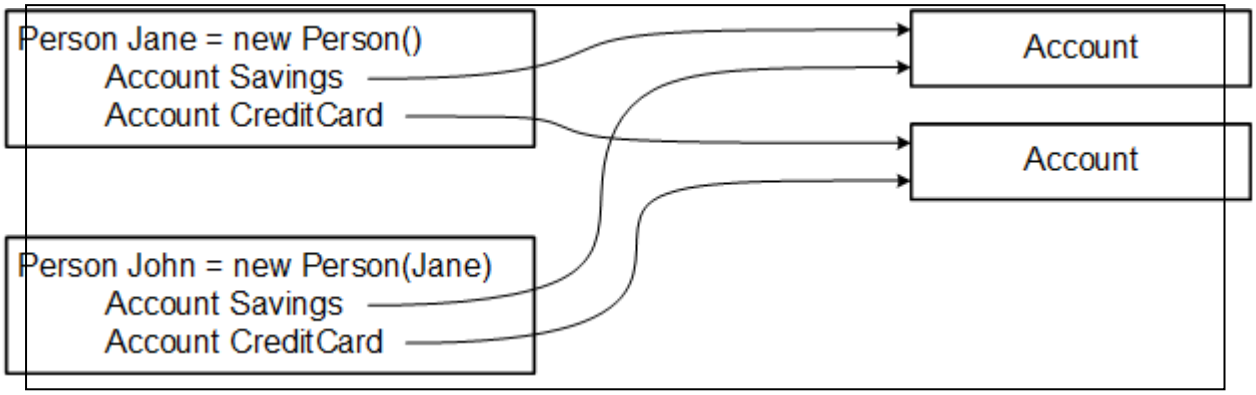

Figure 4.4: The Shallow Copy Problem

The solution to the shallow copy problem is what is to perform what is known as a deep copy. To perform a deep copy you need to also copy all objects that are referenced as well, for which copy constructors are used again, i.e., the copy constructor for the `Person` class will in turn invoke the copy constructors for the referenced `Account` objects, as follows:

```
class Person
{
    ...
    public Person(Person source)
    {
        ...
        Savings = new Account(source.Savings);
        CreditCard = new Account(source.CreditCard);
        ...
    }
    ...
}
```

Importantly, a deep copy is not always required. For example, consider developing an application for managing the sale of books in a book store. In this application, you choose to develop a class `Book` to represent an individual line of stock that is sold by the store and a class `Order` to represent for an order placed at the book store. The `Order` class stores references to the `Book` objects to track the books in the order (in a `List` collection or similar). If the store were to support repeat orders, where the same order is made regularly, or where a previous order can be used as a template/starting point for a new order, it would make sense to copy the previous `Order` object rather than to manually construct a new one. In such an example it would not make sense to copy the `Book` objects.

4.5. Destructors and the Dispose Method

In the same way that constructors provide the ability to automatically initialise an object when it is created, destructor (also called finaliser) provide the ability to auto-

matically clean up an object when it is to be deallocated. Destructors are declared with a similar syntax to constructors, only simpler:

```
[access_modifier ]class  class_name
{
        ...
        ~class_name()
        {
                method_body
        }
        ...
}
```

There are three points to note:

- The name of the method is the same as the class name, preceded by a tilde character (~);
- No access modifier can be specified for the method; and
- No parameter list can be specified for the method.

When contemplating using a destructor, it is important to consider how destructors work with respect to garbage collection, as described in Section 4.3. Firstly, the garbage collector only runs in the background which means that a destructor is not immediately invoked when the program no longer references an object – the destructor will only be invoked after the garbage collector has determined the object should be deallocated. Secondly, when an object to be deallocated has a destructor, the garbage collector doesn't immediately free the memory, instead it schedules the destructor to run then allows the program to continue execution. This means the actual deallocation of the object is delayed until at least the next time that the garbage collector runs.

Importantly, the use of garbage collection also means that there is very little need for destructors. In programming languages that do not have garbage collection, destructors are often used to free up memory that was previously allocated by the object. The garbage collector will free this memory automatically, eliminating that requirement for a destructor. However, destructors are also often used to deallocate any other resources such as files, databases, network connections, GUI components, and so on. These are generally referred to as an unmanaged resource as they involve resources outside of the application/Microsoft.Net framework, e.g., opening a file allocates resources in the operating system, and the file should be closed to free up those resources, an operation that could be performed in a destructor.

Although cleaning up unmanaged resources seems to be an appropriate use for a destructor in C#, the problem still exists that the destructor is not invoked until the garbage collector is invoked. Just like the main memory of a computer, operating system resources such as open files are a limited resource and it is possible for a process to run out of them – most operating systems limit the number of open files for a process to 64 or similar number. If the closing files were to be left to destructors for an application that was processing many files, it is possible (and likely) that the program

could unexpectedly run out of this resource, causing the operating system to terminate the program (the program would "crash").

There is no way to directly invoke a destructor, thus to solve this problem the Microsoft.Net framework incorporates the idea of a dispose method. The `Dispose()` method is a method that can be invoked manually by a programmer when they finish with an object. The `Dispose()` method is invoked in place of a destructor, so any functionality contained within the destructor should also appear in the `Dispose()` method as well. Declaring a `Dispose()` method has the following syntax:

```
[access_modifier ]class  class_name : IDisposable
{
        ...
        public void Dispose()
        {
                method_body
                GC.SuppressFinalize(this);
        }
        ...
}
```

There are four points to note with this syntax:

- `: IDisposable` appears after name of the class (this is an interface, which we examine in Section 6.5);
- The signature of the `Dispose()` method is fixed, consisting of a `public` method called `Dispose`, no parameter list, and returning `void`;
- The final line of the method invokes the garbage collector (`GC`) to tell it that this object no longer needs to be finalised, i.e., the destructor no longer needs to be invoked;
- Any objects referenced by the object should also have their `Dispose()` method invoked as well, if they have one (these would be explicitly invoked in the method_body).

Importantly, this approach relies on programmer discipline to invoke the `Dispose()` method each time a relevant object is finished with. Given that this is not necessarily reliable, you should also provide a destructor to do the same cleaning up of unmanaged resources[7] which will always be invoked (eventually) if the programmer forgets to invoke the `Dispose()` method.

4.6. Delegates

We have already seen that we can have a reference to an object:

```
SomeClass someObject = new SomeClass();
```

In the above example, the variable `someObject` is a reference to an object of type `SomeClass`. It is similarly possible to have a reference to a method, known as a

[7] As discussed previously, duplicate code can and should be avoided by using a third method to hold common code for cleaning up the unmanaged resources.
[8] The equivalent mechanism to delegates are often named differently in other programming languages, e.g., C++ has function pointers.

. The most common use of delegates in C# is for event handlers for a Windows GUI application. When an event occurs, such as a mouse click on a `Button` object, the GUI code checks to see if the programmer has stored a reference to a method in the `Button.Click` delegate. If such a reference has been stored, the method is invoked upon the button being clicked. There are four elements to using delegates that we must consider:

1. Creating a delegate data type;
2. Creating a delegate variable;
3. Assigning the method reference to the delegate variable; and
4. Invoking a method using the delegate variable.

Like all variables, delegates require a data type. Creating a delegate data type is closely related to the methods that are going to be referenced by the delegate variables that are created – only one method signature and return type can be referenced by one type of delegate. Consider the following syntax, showing the first line of a method declaration followed by the line to create a matching delegate type:

```
[access_modifier1 ][static   ]return_type  method_name (parameter_list)
{
        ...
}

[access_modifier2 ]delegate   return_type  delegate_type_name (parameter_list) ;
```

From the above syntax, it should be apparent that the *return_type* and *parameter_list* must match exactly (note that only data types must match in the parameter list, parameter names can be different). The name of the delegate data type (*delegate_type_name*) should be named in a similar manner to class names and are usually named with normal capitalisation. The keyword `delegate` appears before the *return_type* and there is an optional access modifier like for any other declaration. The two access modifiers, *access_modifier1* and *access_modifier2*, do not need to match, instead *access_modifier2* defines the visibility of the delegate data type, with the same implications as defining a data type with a class.

Now that we have the delegate data type, we can create delegate variables much the same as we declare an object reference:

```
[access_modifier ]delegate_type_name  variable_name;
```

for a member variable, or without the optional *access_modifier* for a variable local to a method.

Assigning a method reference is much the same as assigning a value to a simple data type, i.e.,

```
variable_name  =  method_name;
```

As for other data types, this can be combined with the creation of the variable:

```
[access_modifier ]delegate_type_name  variable_name  =  method_name;
```

Finally, invoking a method through a delegate is straightforward, the delegate variable is just used the same as a normal method name, passing any parameters and storing/testing the return value as appropriate, i.e.,

variable_name(*parameter_list*);

A more complex example of using delegates is shown in Figure 4.5. In this example, a delegate type is created by the name `ValueTester` for methods that take a single parameter of type `int` and return a `string`. A method is created matching this signature, `IsZero`, which tests the parameter's value and returns a `string` indicating whether or not it is equal to zero. The `Main` method then creates a delegate variable, `delegateVariable`, to which the `IsZero` method is assigned. After obtaining an integer value from the user, the `delegateVariable` delegate is then used to invoke the `IsZero` method, passing the user's input as the parameter and displaying the results on the `Console`.

```
/*************************************************************
** File: DelegateDemo.cs
** Author/s: Justin Rough
** Description:
**     A simple program demonstrating how to create a delegate
** type, create a delegate variable, assign a method reference
** to the delegate variable and invoke the method referenced
** by the delegate.
*************************************************************/
using System;

namespace DelegateDemo
{
    class DelegateDemo
    {
        private delegate string ValueTester(int value);

        static string IsZero(int value)
        {
            if (value == 0)
                return string.Format("{0} is zero", value);
            else
                return string.Format("{0} is NOT zero", value);
        }

        static void Main(string[] args)
        {
            ValueTester delegateVariable = IsZero;

            Console.Write("Enter an integer: ");
            int number = Convert.ToInt32(Console.ReadLine());

            Console.WriteLine(delegateVariable(number));
        }
    }
}
```

Figure 4.5: Example of Creating and Using Delegates

4.7. Indexers and ToString

In this section we examine two class members we haven't yet seen: indexers and the `ToString()` method. Indexers allow us to use a notation with our classes that is similar to arrays, e.g., `variable[index]`. The `ToString()` method is a method that is present in every single object in C# and the Microsoft.Net framework, and is used to provide a `string` representation of any object. By default, the `ToString()` method only returns the name of the class defining the object, so we examine how to change this to something more appropriate.

4.7.1. Indexers

Defining an indexer is a means through which we can allow instances of classes we have defined to be accessed similar to an element in an array, e.g., `variable[index]`. Unlike arrays however, the index does not need to be a numeric value, and can be any data type. Indexers use a syntax that is a mix of properties and methods:

```
[access_modifier ]return_type this[parameter[, ...]]
{
        [get
        {
                accessor_body
        }]
        [set
        {
                mutator_body
        }]
}
```

Instead of a name, the indexer uses the keyword `this`. Indexers have a parameter list just like a method declaration, however the parameters appear inside the square brackets (`[]`) rather than parenthesis (`()`), and at least one parameter must be specified. Like a property, an indexer has both a `get` and `set` code block, and also makes use of the `value` keyword in the same way as a property. Parameters can also be used from within both code blocks.

Defining an indexer is useful where you want to provide an interface to a collection of indexed data. For example, consider writing a class for a dictionary where you can store a definition:

```
myDictionary.StoreDefinition("word", "definition");
...
string retrievedDefinition = myDictionary.RetrieveDefinition("word");
```

This code could be improved with an indexer so that it reads as follows:

```
myDictionary["word"] = "definition";
...
string retrievedDefinition = myDictionary["word"];
```

by defining an indexer similar to:

```
public string this[string word]
{
    get
    {
        ...
        return definition;
    }

    set
    {
        // code to store definition (value) for word
        ...
    }
}
```

4.7.2. The ToString Method

All objects in the C# programming language include a `ToString()` method, including all value types and reference types. Even in the most simplest of classes you might define, including an empty class with no declared members, the `ToString()` method will be present and will return the name of the class defining the object. How the `ToString()` method appears in every class is discussed in Section 6.2, however for now we only need to know what the `ToString()` method is and how to provide a more appropriate implementation for it.

The intention of the `ToString()` method is to provide a string representation of an object, i.e., to provide a brief textual description of the contents/value of an object. The `ToString()` method is called automatically whenever a variable needs to be converted to a string, such as when it appears in a Console.Write or Console.WriteLine statement. Thus, it is important that `ToString()` provide an appropriate textual representation for the classes you define.

Sometimes the output of `ToString()` is straightforward, for example:

```
int value = 5;
Console.WriteLine(value.ToString());
```

The above code displays a single line of text containing the value `5`, i.e., `value.ToString()` returns the string `"5"` which is clearly representative of the data stored in the value object. However, for a custom data type such as a `Person` or `Book`, this may not be as simple. There are no rules regarding what a `ToString()` method should return, however in general it should be something relatively simple and not be broken over several lines because the text may be used in different contexts, e.g., in console output, in a GUI label or other control, or in a formatted report.

To define an appropriate implementation of the `ToString()` method for a class, the following syntax is used:

```
public override string ToString()
{
        method body returning a string containing the textual representation
}
```

The exact meaning of the `override` keyword in above syntax is examined in detail in Section 6.2.

To prepare the textual representation, we consider two alternatives: ***string concatenation*** and the `string.Format` method. String concatenation is achieved simply by adding various strings together, e.g.,

```
string result = "a" + "b" + "c" + "d" + "e";
```

Variables can be used in place of the literals in this `string` of course, and if necessary you can invoke the `ToString()` method of some other object to get a string needed for concatenation, e.g.,

```
int value = 5;
string result = "a" + "b" + value.ToString() + "d" + "e";
```

Although this is often the quickest way create a textual representation in a string, there are several disadvantages to this approach. Firstly, there is no control over the formatting of data placed in this string, i.e., no support for composite formatting. Secondly, you are relying upon the compiler to make this code efficient. Recall from Section 2.6 that strings are immutable, i.e., they cannot be modified. This means that each time you add two strings together a new string is created. Thus, in the above examples, a total of four strings were created as the operators were evaluated one after the other. In particular, the following strings were created: `"ab"` (the result of `"a" + "b"`), `"abc"` (the result of `+ "c"`), `"abcd"` (the result of `+ "d"`), and `"abcde"` (the result of `+ "e"`). The compiler will make some optimisations to your code for the final executable, e.g., "a" + "b" will be changed to "ab", however you have no control over what optimisations may or may not be made.

The most common way to address this problem is to use the `Format` method of the `string` class (a `static` method). Use of the `string.Format` method is identical to the composite formatting we have already done with the `Console.Write` and `Console.WriteLine` methods. The key difference is that rather than sending the result to the console window, the `string.Format` method returns the resulting `string`. Equivalent examples to those above are:

```
string result = string.Format("{0}{1}{2}{3}{4}", "a", "b", "c", "d", "e");
```

and

```
int value = 5;
string result = string.Format("{0}{1}{2}{3}{4}", "a", "b", value.ToString
(), "d", "e");
```

In these examples, only one string object is created. Figure 4.6 shows a simple Person class where a `ToString()` method has been provided to return the name of the person in the format of surname first, followed by the given name after a comma (,).

```
class Person
{
```

```
    public Person(string givenName, string familyName)
    {
        _GivenName = givenName;
        _FamilyName = familyName;
    }

    private string _FamilyName;
    public string FamilyName
    {
        get { return _FamilyName; }
        set { _FamilyName = value; }
    }

    private string _GivenName;
    public string GivenName
    {
        get { return _GivenName; }
        set { _GivenName = value; }
    }

    public override string ToString()
    {
        return string.Format("{0}, {1}", _FamilyName, _GivenName);
    }
}
```

Figure 4.6: Example of Defining ToString for a Person Class

Task 4.1. Object Creation and Linkage

Objective: Creating, initialising, and linking objects is a key skill that you will require and apply regularly in developing object-oriented applications. Importantly, this skill needs to become second nature and does not take a very long time, and linked with an incremental development approach will become something you achieve quickly. In this task, we explore the concepts examined in this session so far, developing the skills that are critical for successful software development using the object-oriented model. In particular, we examine the creation of objects and establishing links between those objects, controlling the initialisation of objects through constructors, and separating user-interface elements from data model elements.

This task requires you to complete an object-oriented application consisting of three classes: the `Program` class, containing `Main()`, a `Book` class representing a textbook, and an `Order` class, representing an order of textbooks. You are required to write these classes according to the specifications below.

The requirements for the `Book` class are as follows:

- Private attributes for the book authors (`_Authors`), title (`_Title`), publisher (`_Publisher`), and year (`_Year`), which are string attributes except for the year which is an integer attribute;
- Public read-only properties encapsulating each of the above attributes, i.e., a property for author, title, publisher, and year, each only containing a `get` section.
- A custom constructor that accepts four parameters (`authors`, `title`, `publisher`, and

`year`), initialising the above attributes; and
- Override the `ToString()` method to return the format: "*authors, title, publisher, year*".

The requirements for the `Order` class are as follows:

- Add a using statement to the top of the source code file: `using System.Collections.ObjectModel;`
- A private constant (`DEFAULT_CUSTOMER`) which is a string with the value `"UNKNOWN"`;
- A private attribute for the customer (`_Customer`) which is a string attribute;
- A public read-only property encapsulating the above attribute;
- A private declaration for a `List<>` of Book objects (`_Books`), i.e., declare the variable but don't create the `List<>` yet;
- A public property `Books`, defined as follows:

```
public ReadOnlyCollection<Book> Books
{
    get { return _Books.AsReadOnly(); }
}
```

- A custom constructor that takes a single string parameter representing the customer (`customer`) that initialises the matching attribute and creates the `_Books List<>`;
- A parameter-less constructor that uses an object initialiser (Section 4.4) to invoke the previously defined constructor, using the constant `DEFAULT_CUSTOMER` for the value of the `customer` parameter;
- A copy constructor that copies the contents of another `Order` object to the new object (see below for how to copy a `List<>`);
- A public method `Add()` which takes a single book parameter (`Book`) and adds that object to the `_Books` list, returning `void`; and
- Override the `ToString()` method to return the format: "Order for *customer* containing *count* books" (*count* is obtained with `_Books.Count`).

The requirements of the `Program` class are as follows:

- Create a method `DisplayOrder()` which takes a single parameter of type `Order` and returns `void`, with the following functionality:
  - Display the result of the `Order` object's `ToString()` on the screen; and
  - Use a `foreach` loop to go through the list of objects in the order's `Books` property, displaying the object's `ToString()` on the screen preceded by `"-- "` (two minus signs followed by a space).
- The Main method contains the following functionality:
  - Creates two `Book` objects, one representing the Deitel (prescribed) textbook and the other recommending the Schildt (recommended) textbook as indicated in the CloudDeakin document "Unit Textbooks" (under Administrative Information);
  - Creates three `Order` objects, each demonstrating a different type of constructor (custom, parameter-less, and copy), one containing the prescribed textbook only, one containing the recommended textbook only, and one containing both; and
  - Invokes the `DisplayOrder()` method defined above to display the contents of the three Order objects.

Copying a `List<>` (required for the `Order` copy constructor) is achieved by passing the list to be copied to the constructor during object creation, e.g., to copy an existing list `source` to a new list `destination`, you would enter:

```
List<Book> destination = new List<Book>(source);
```

A completed application should produce output similar to the following:

```
Order for UNKNOWN containing 1 books
-- Deitel, P., and Deitel, H., Visual C# 2012: How to Program, Pearson,
2014.

Order for Freddie containing 1 books
-- Schildt, H., C# 4.0: The Complete Reference, Osborne Media, 2001.

Order for UNKNOWN containing 2 books
-- Deitel, P., and Deitel, H., Visual C# 2012: How to Program, Pearson,
2014.
-- Schildt, H., C# 4.0: The Complete Reference, Osborne Media, 2001.
```

The above text represents the specification of requirements for the complete application. Whenever approaching a programming task, it is important to apply an incremental approach, i.e., identify a set of stages for development, rather than attempting to develop the entire application. If you are struggling to identify your possible stages of development, try applying the following stages in your approach:

- Create your solution, and add the `Book` and `Order` classes to the solution (don't write any code yet);
- Add the declarations for the attributes, properties, and constant to each class to get the basic structure of data in place (tip: applying the same skill at the same time, rather than constantly switching skills, always helps to speed things up!);
- Now work on the constructors, starting with the `Book` class, because that's the easiest;
- Now that all the other members are in place for our classes, implement the `ToString()` overrides;
- If you haven't already, build your solution to make sure everything is compiling, eliminating any syntax error messages before proceeding; and
- Finally, develop the `Program` class, again identifying what you think are appropriate stages. This should be a much simpler task, given that all the classes used should be fully developed.

Note how the above recommended stages resulted in a fully developed data model in the defined classes before any functionality or interface for the application was developed. This is a common strategy, and the best developers are able to breakdown a coding task quickly and effectively to ensure they maximise their productivity as much as possible. In the above example, all of the data representation classes are developed first, leaving only the functionality of the application remaining.

4.8. Namespaces

We first saw namespaces mentioned in Section 1.2 when considering our very first program in the C# programming language. Modern programming languages often provide the concept of a namespace as an important mechanism for managing the naming of data types in your programs. Whenever you create a class, enumerated data type, a delegate, etc., you are creating a new data type within your application's namespace.

A good analogy for namespaces is to consider sub-directories on a disk. You can place as many files into a directory on a disk as you like, as long as you use a different name for that file. If you try to use the same name, a conflict occurs which you must resolve either by renaming the file or replacing the existing file with that name. Over time however, this approach becomes unmanageable, so we add sub-directories to separate the files into logical groupings and make our files easier to manage.

Data type names in the program code are equivalent to files, i.e., you can have as many as you like, but must use a different name each time. This can become a problem in very large applications, where the same concept may be repeated several times. For example, consider an online web site for a bookstore. In such an application, you may create an `Order` class to define orders made by customers on the web site, but wish to define an `Order` class for automatically creating a purchase order for suppliers when stock levels fall below pre-determined levels. Although you could solve the problem by naming these classes `CustomerOrder` and `SupplierOrder`, there is likely more functionality related to customers and suppliers than just orders, and it would be ideal to separate all the data types for each area.

Namespaces are the equivalent to sub-directories. You could define two namespaces for the above example: `Customer` and `Supplier`. You could then define an `Order` class within each of these namespaces. Namespaces are declared as follows:

```
namespace   namespace_name
{
        namespace_member ...
}
```

In the above syntax, *namespace_name* is an identifier that specifies the name of the namespace (just like a directory name). The rules for naming namespaces are the same as those for naming variables (see Section 1.4), however namespaces can be nested within other namespaces by using a full stop ('.'), e.g., `Customer`, `BookWebSite.Customer`, and so on. A *namespace_member* is then the declaration of any data type (class, enumerated data type, delegate, etc.), for which several declarations can appear in the namespace. Unlike declaring a data type however[9], namespace declar-

[9] Actually, C# also includes the ability split the definition of a class across several files, referred to as partial classes, however this concept is unique to the C# programming language and has limited applications, therefore is outside the scope of this unit.

ations can be spread over several files, just requiring the same declaration with the same *namespace_name*. These are then combined together by the compiler.

Once a namespace is defined, you can then access the elements of that namespace in one of two ways. First, you can access the namespace members by giving their full name. For the example above, the two classes can be referred to directly as `Customer.Order` and `Supplier.Order`. Alternatively, you can "import" the names from a namespace, which allows the use of data types defined in a namespace without the namespace prefix. For this purpose, you place a `using` statement at the top of the program, e.g.,

```
using Supplier;
```

and can then access the data types directly, e.g.,

```
Order myOrder = new Order(); // Supplier.Order class
```

We have in fact already been using this approach for the `Console` class, which is defined in the `System` namespace, i.e.,

```
using System;
...
Console.WriteLine(...);
```

Alternatively, we could have just used:

```
System.Console.WriteLine(...);
```

4.9. Report Writing

The preparation of reports is a fairly common task when working with text based devices such as the console, but also text files, text printers, and so on. Beyond mastering composite formatting, report writing is a relatively simple skill to obtain, only requiring that you align the various fields of the report correctly and use appropriate characters to separate the fields of the report, as appropriate. There are a number of characters that can be used, such as those indicated in the following four examples:

```
#########################################################################
# Qty # Description                                   #    Cost    #
#########################################################################
#   5 # Visual C#: How to Program                     #    $134.95 #
#  50 # Database Systems                              #    $119.95 #
# 500 # HTML & XHTML: The Complete Reference          #     $72.00 #
#########################################################################

*************************************************************************
* Qty * Description                                   *    Cost    *
*************************************************************************
*   5 * Visual C*: How to Program                     *    $134.95 *
*  50 * Database Systems                              *    $119.95 *
* 500 * HTML & XHTML: The Complete Reference          *     $72.00 *
*************************************************************************

+-----+-------------------------------------------------+------------+
```

```
| Qty | Description                                    |    Cost    |
+-----+------------------------------------------------+------------+
|   5 | Visual C#: How to Program                      |    $134.95 |
|  50 | Database Systems                               |    $119.95 |
| 500 | HTML & XHTML: The Complete Reference           |     $72.00 |
+-----+------------------------------------------------+------------+

Qty | Description                                    |    Cost
================================================================
  5 | Visual C#: How to Program                      |    $134.95
 50 | Database Systems                               |    $119.95
500 | HTML & XHTML: The Complete Reference           |     $72.00
```

The last two formats use a combination of the plus (+), minus (-), pipe (|), and equals symbols (=) to form the lines between rows and columns.

Beyond the basic formatting indicated above, the only problem that needs to be considered is what to do when the data to be displayed is too long. We have already seen in Section 2.9 how to specify a minimum field width using composite formatting, but unfortunately there is no way to specify a maximum field width. Instead, we can modify (a copy of) our data to fit into the field width. The following are several ways the text `"Data is too long"` could be displayed in a field 10 characters wide:

- "`Data is too long`" – no change, which although the report will look broken/-jagged it guarantees that no data is missing/lost
- "`Data is to`" – text is truncated to the appropriate length;
- "`Data is >>`" – text is truncated, but '>>' is used to indicate missing data; and
- "`**********`" – the text is replaced by a sequence of characters that clearly indicate data is missing[10].

Note that there is no specific correct or incorrect way to do report writing, however in an industry/commercial setting the report format will often be designed by the organisation/staff who commission a software development project, and the approach used may be determined by organisational standards.

To help with writing reports, there are two useful features of the `string` class:

- A string can be truncated by using the Substring method which takes two parameters, the starting index and the length, e.g., to truncate a string to 10 characters:
  ```
  someString = someString.Substring(0, 10);
  ```
- To create a string containing some number of the same character, use one of the string's constructors overloads which takes two parameters, the character and how many, e.g., to create a string containing 40 asterisks ('*'):
  ```
  string asterisks = new string('*', 40);
  ```

[10] Although at first glance this may appear to be the worst solution, this approach is in fact used quite regularly. As an example, numerical values in Microsoft Excel that are too wide for a column are replaced by filling the cell with the hash symbol (#).

Task 4.2. Report Writing

Objective: Report writing is a fairly fundamental skill when developing applications that use the console or other text based devices. In this task you will complete a simple report for this purpose.

You are required to prepare one of the reports as shown above in Section 4.9 – which of these reports you produce is up to you. To do this, you must undertake the following tasks:

- Prepare a class `BookStock` with instance variables for quantity, description, and price. Encapsulate these instance variables with a property, and provide a custom constructor with appropriate parameters to initialise these attributes.
- In the `Main` method, create an array containing the three books as indicated in the above reports, then invokes a method `ProduceReport()` that takes the array as a parameter and produces the report as indicated, using loops as appropriate (this method should work correctly given an array of any size).

Hint: Remember to apply the incremental development approach examined in the previous task!