msdn training

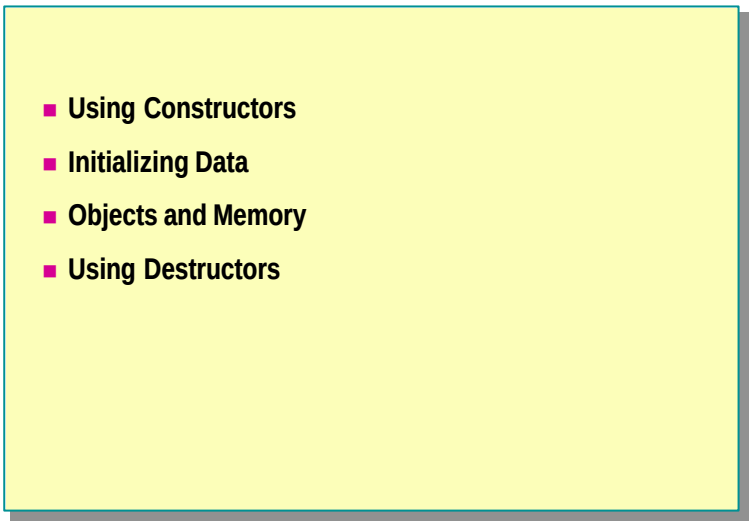
Module 9: Creating and Destroying Objects

Contents

This course is based on the prerelease Beta 1 version of Microsoft® Visual Studio .NET. Content in the final release of the course may be different from the content included in this prerelease version. All labs in the course are to be completed with the Beta 1 version of Visual Studio .NET.

Microsoft

Overview



In this module, you will learn what happens when an object is created, how to use constructors to initialize objects, and how to use destructors to destroy objects. You will also learn what happens when an object is destroyed and how garbage collection reclaims memory.

After completing this module, you will be able to:

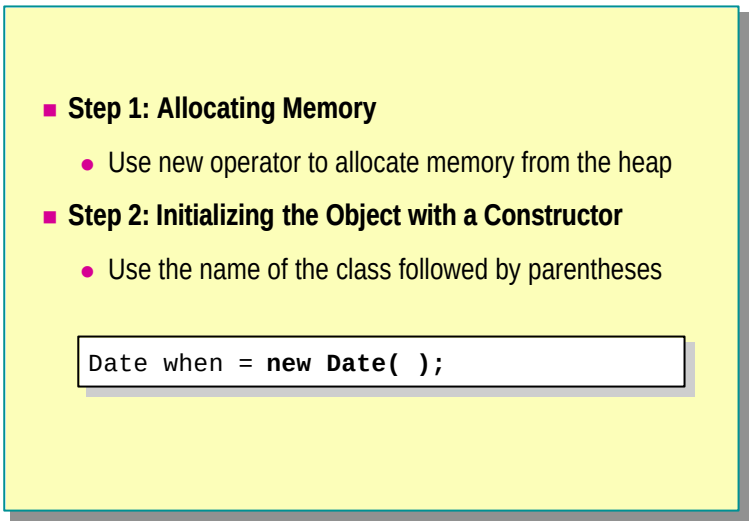
- Use constructors to initialize objects.
- Create overloaded constructors that can accept varying parameters.
- Describe the lifetime of an object and what happens when it is destroyed.
- Create destructors and use **Finalize** methods.

Using Constructors

- Creating Objects
- Using the Default Constructor
- Overriding the Default Constructor
- Overloading Constructors

Constructors are special methods that you use to initialize objects when you create them. Even if you do not write a constructor yourself, a default constructor is provided for you whenever you create an object from a reference type. In this section, you will learn how to use constructors to control what happens when an object is created.

Creating Objects



The process of creating an object in C# involves two steps:

1. Use the **new** keyword to acquire and allocate memory for the object.
2. Write a constructor to turn the memory acquired by **new** into an object.

Even though there are two steps in this process, you must perform both steps in one line of code. For example, if **Date** is the name of a class, use the following syntax to allocate memory and initialize the object **when**.

```
Date when = new Date( );
```

Step 1: Allocating Memory

The first step in creating an object is to allocate memory for the object. All objects are created by using the **new** operator. There are no exceptions to this rule. You can do this explicitly in your code, or the compiler will do it for you.

In the following table, you can see examples of code and what they represent.

Code example	Represents
string s = "Hello";	string s = **new** string("Hello");
int[ ] array = {1,2,3,4};	int[ ] array = **new** int[4]{1,2,3,4};

How new Affects Performance

Generally, there are only two functions of **new** that affect performance:

- A Boolean test

  The heap is a contiguous block of memory of known size. A special pointer marks the current position in the heap for memory allocation purposes. All memory to one side of the position has already been allocated by **new**. All memory to the other side of the position is still available. The Boolean test simply uses the difference between this position and the end of the heap to determine how many bytes of free memory are left in the heap. It then compares this amount to the number of bytes requested by **new**.

- A pointer increment

  If there are enough free bytes left in the heap, the special pointer is incremented by the number of bytes requested, thus marking the memory as allocated. The address of the allocated block is then returned.

This makes the dynamic allocation of heap memory essentially as fast as the dynamic allocation of stack memory.

Note Strictly speaking, this is only true if there is only one variable. If there are multiple variables, the stack-based variables will be allocated all at once, but the heap variables will require multiple allocations.

Step 2: Initializing the Object with a Constructor

The second step in creating an object is to write a constructor. A constructor turns the memory allocated by **new** into an object. There are two types of constructors: instance constructors and static constructors. Instance constructors are constructors that initialize objects. Static constructors are constructors that initialize classes.

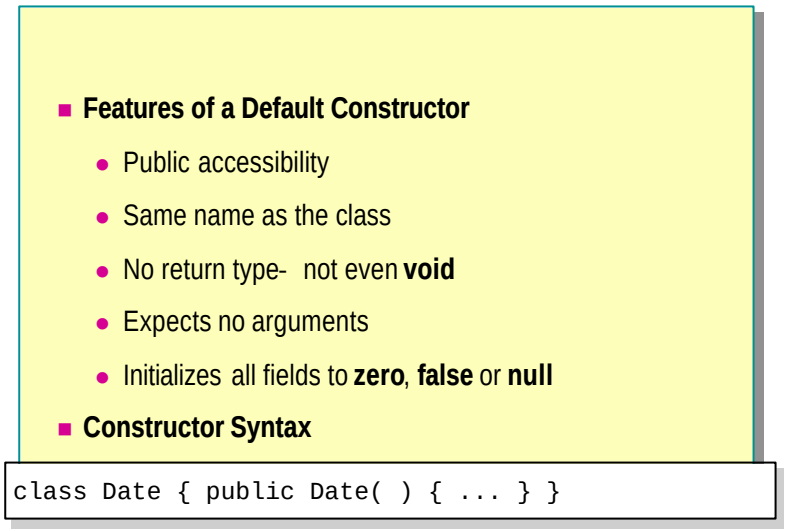
How new and Instance Constructors Collaborate

It is important to realize how closely **new** and instance constructors collaborate to create objects. The only purpose of **new** is to acquire raw uninitialized memory. The only purpose of an instance constructor is to initialize the memory and convert it into an object that is ready to use. Specifically, **new** is not involved with initialization in any way, and instance constructors are not involved in acquiring memory in any way.

Although **new** and instance constructors perform separate tasks, as a programmer you cannot use them separately. This is one way for C# to help guarantee that memory is always definitely set to a valid value before it is read. (This is called *definite assignment*.)

Note to C++ Programmers In C++, you can allocate memory and not initialize it (by directly calling operator **new**). You can also initialize memory allocated previously (by using placement **new**). This separation is not possible in C#.

Using the Default Constructor



When you create an object, the C# compiler provides a default constructor if you do not write one yourself. Consider the following example:

```
class Date
{
    private int ccyy, mm, dd;
}

class Test
{
    static void Main( )
    {
        Date when = new Date( );
        ...
    }
}
```

The statement inside **Test.Main** creates a **Date** object called **when** by using **new** (which allocates memory from the heap) and by calling a special method that has the same name as the class (the instance constructor). However, the **Date** class does not declare an instance constructor. (It does not declare any methods at all.) By default, the compiler automatically generates a default instance constructor.

Features of a Default Constructor

Conceptually, the instance constructor that the compiler generates for the **Date** class looks like the following example:

```
class Date
{
    public Date( )
    {
        ccyy = 0;
        mm = 0;
        dd = 0;
    }
    private int ccyy, mm, dd;
}
```

The constructor has the following features:

- Same name as the class name

  By definition, an instance constructor is a method that has the same name as its class. This is a natural and intuitive definition and matches the syntax that you have already seen. Following is an example:

  ```
  Date when = new Date( );
  ```

- No return type

  This is the second defining characteristic of a constructor. A constructor never has a return type—not even **void**.

- No arguments required

  It is possible to declare constructors that take arguments. However, the default constructor generated by the compiler expects no arguments.

- All fields initialized to zero

  This is important. The compiler-generated default constructor implicitly initializes all non-static fields as follows:

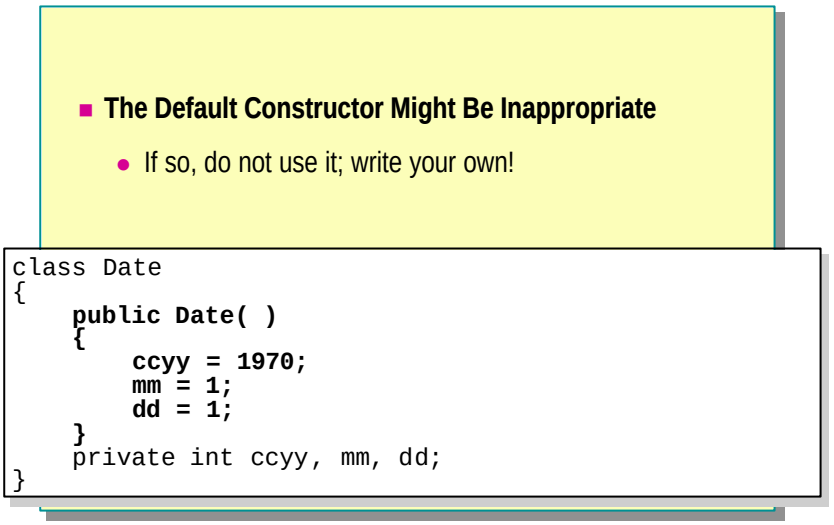
  - Numeric fields (such as int, double, and decimal) are initialized to zero.
  - Fields of type bool are initialized to **false**.
  - Reference types (covered in an earlier module) are initialized to **null**.
  - Fields of type struct are initialized to contain zero values in all their elements.

- Public accessibility

  This allows new instances of the object to be created.

Note In Module 10, "Inheritance in C#," in Course 2124A, *Introduction to C# Programming for the Microsoft .NET Platform (Prerelease)*, you will learn about abstract classes. The compiler-generated default constructor for an abstract class has protected access.

Overriding the Default Constructor



Sometimes it is not appropriate for you to use the compiler-generated default constructor. In these cases, you can write your own constructor that contains only the code to initialize fields to non-zero values. Any fields that you do not initialize in your constructor will retain their default initialization of zero.

What If the Default Constructor Is Inappropriate?

There are several cases in which the compiler-generated default constructor may be inappropriate:

- Public access is sometimes inappropriate.

  The Factory Method pattern uses a non-public constructor. (The Factory Method pattern is discussed in *Design Patterns: Elements of Reusable Object-Oriented Software*, by E. Gamma, R. Helm, R. Johnson, and J. Vlissides. It is covered in a later module.)

  Procedural functions (such as **Cos** and **Sin**) often use private constructors.

  The Singleton pattern typically uses a private constructor. (The Singleton pattern is also covered in *Design Patterns: Elements of Reusable Object-Oriented Software* and in a later topic in this section.)

- Zero initialization is sometimes inappropriate.

  Consider the compiler-generated default constructor for the following **Date** class:

  ```
  class Date
  {
      private int ccyy, mm, dd;
  }
  ```

  The default constructor will initialize the year field (*ccyy*) to zero, the month field (*mm*) to zero, and the day field (*dd*) to zero. This might not be appropriate if you want the date to default to a different value.

- Invisible code is hard to maintain

  You cannot see the default constructor code. This can occasionally be a problem. For example, you cannot single-step through invisible code when debugging. Additionally, if you choose to use the default initialization to zero, how will developers who need to maintain the code know that this choice was deliberate?

Writing Your Own Default Constructor

If the compiler-generated default constructor is inappropriate, you must write your own default constructor. The C# language helps you to do this.

You can write a constructor that only contains the code to initialize fields to non-zero values. All fields that are not initialized in your constructor retain their default initialization to zero. The following code provides an example:

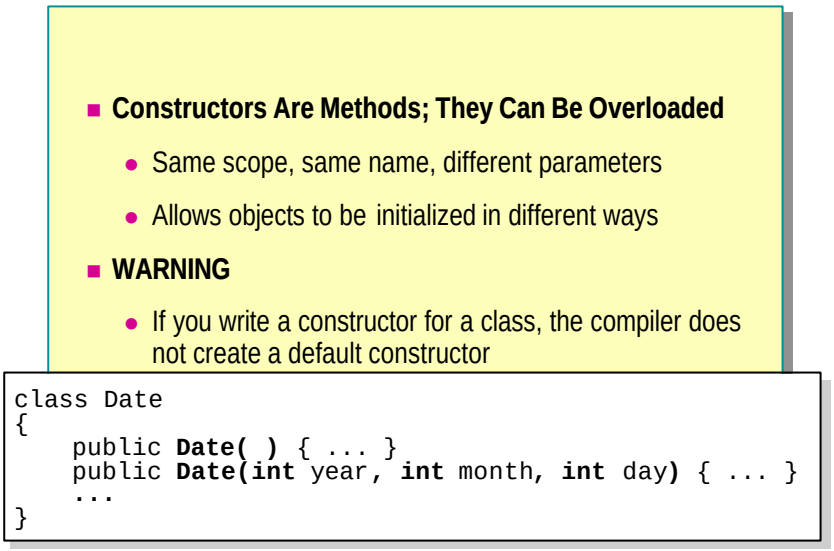
```
class DefaultInit
{
    public int a, b;
    public DefaultInit( )
    {
        a = 42;
        // b retains default initialization to zero
    }
}
class Test
{
    static void Main( )
    {
        DefaultInit di = new DefaultInit( );
        Console.WriteLine(di.a); // Writes 42
        Console.WriteLine(di.b); // Writes zero
    }
}
```

You should be wary of doing more than simple initializations in your own constructors. You must consider potential failure: the only sensible way you can signal an initialization failure in a constructor is by throwing an exception.

Note The same is also true for operators. Operators are discussed in Module 12, "Operators, Delegates, and Events," in Course 2124A, *Introduction to C# Programming for the Microsoft .NET Platform (Prerelease)*.

When initialization succeeds, you have an object that you can use. If initialization fails, you do not have an object.

Overloading Constructors



Constructors are special kinds of methods. Just as you can overload methods, you can overload constructors.

What Is Overloading?

Overloading is the technical term for declaring two or more methods in the same scope with the same name. The following code provides an example:

```
class Overload
{
    public void Method( ) { ... }
    public void Method(int x) { ... }
}
class Use
{
    static void Main( )
    {
        Overload o = new Overload( );
        o.Method( );
        o.Method(42);
    }
}
```

In this code example, two methods called **Method** are declared in the scope of the **Overload** class, and both are called in **Use.Main**. There is no ambiguity, because the number and types of the arguments determine which method is called.

Initializing an Object in More Than One Way

The ability to initialize an object in different ways was one of the primary motivations for allowing overloading. Constructors are special kinds of methods, and they can be overloaded exactly like methods. This means you can define different ways to initialize an object. The following code provides an example:

```
class Overload
{
    private int data;
    public Overload( ) { this.data = -1; }
    public Overload(int x) { this.data = x; }
}

class Use
{
    static void Main( )
    {
        Overload o1 = new Overload( );
        Overload o2 = new Overload(42);
        ...
    }
}
```

Object **o1** is created by using the constructor that takes no arguments, and the private instance variable *data* is set to –1. Object **o2** is created by using the constructor that takes a single integer, and the instance variable *data* is set to 42.

Initializing Fields to Non-Default Values

You will find many cases in which fields cannot be sensibly initialized to zero. In these cases, you can write your own constructor that requires one or more parameters that are then used to initialize the fields. For example, consider the following **Date** class:

```
class Date
{
    public Date(int year, int month, int day)
    {
        ccyy = year;
        mm = month;
        dd = day;
    }
    private int ccyy, mm, dd;
}
```

One problem with this constructor is that it is easy to get the order of the arguments wrong. For example:

```
Date birthday = new Date(23, 11, 1968); // Error
```

The code should read `new Date(1968,11,23)`. This error will not be detected as a compile-time error because all three arguments are integers. One way you could fix this would be to use the Whole Value pattern. You could turn *Year*, *Month*, and *Day* into structs rather than int values, as follows:

```
struct Year
{
    public readonly int value;
    public Year(int value) { this.value = value; }
}

struct Month // Or as an enum
{
    public readonly int value;
    public Month(int value) { this.value = value; }
}
struct Day
{
    public readonly int value;
    public Day(int value) { this.value = value; }
}
class Date
{
    public Date(Year y, Month m, Day d)
    {
        ccyy = y.value;
        mm = m.value;
        dd = d.value;
    }
    private int ccyy, mm, dd;
}
```

Tip Using structs or enums rather than classes for *Day*, *Month*, and *Year* reduces the overhead when creating a **Date** object. This will be explained later in this module.

The following code shows a simple change that would not only catch argument-order errors but would also allow you to create overloaded **Date** constructors for U.K. format, U.S. format, and ISO format:

```
class Date
{
    public Date(Year y, Month m, Day d) { ... } // ISO
    public Date(Month m, Day d, Year y) { ... } // US
    public Date(Day d, Month m, Year y) { ... } // UK
    ...
    private int ccyy, mm, dd;
}
```

Overloading and the Default Constructor

If you declare a class with a constructor, the compiler does not generate the default constructor. In the following example, the **Date** class is declared with a constructor, so the expression `new Date( )` will not compile:

```
class Date
{
    public Date(Year y, Month m, Day d) { ... }
    // No other constructor
    private int ccyy, mm, dd;
}
class Fails
{
    static void Main( )
    {
        Date defaulted = new Date( ); // Compile-time error
    }
}
```

This means that if you want to be able to create **Date** objects without supplying any constructor arguments, you will need to explicitly declare an overloaded default constructor, as in the following example:

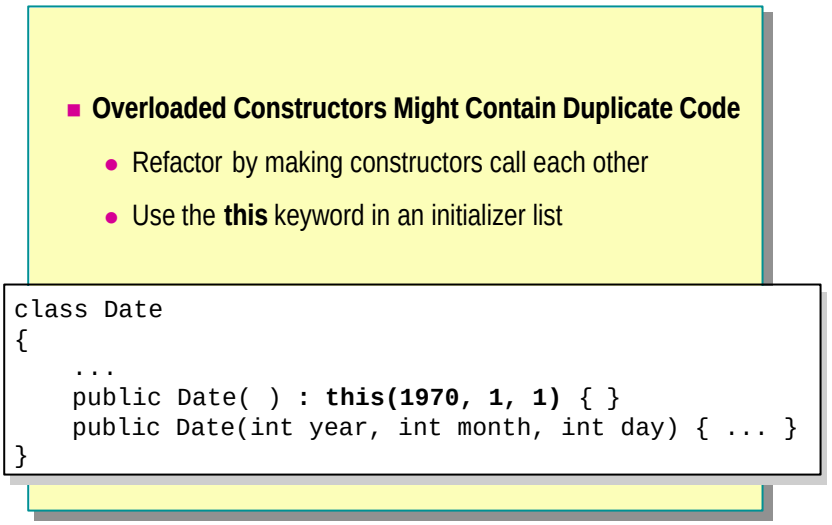
```
class Date
{
    public Date( ) { ... }
    public Date(Year y, Month m, Day d) { ... }
    ...
    private int ccyy, mm, dd;
}
class Succeeds
{
    static void Main( )
    {
        Date defaulted = new Date( ); // Okay
    }
}
```

◆ Initializing Data

- Using Initializer Lists
- Declaring Readonly Variables and Constants
- Initializing Readonly Fields
- Declaring a Constructor for a Struct
- Using Private Constructors
- Using Static Constructors

You have seen the basic elements of constructors. Constructors also have a number of additional features and uses. In this section you will learn how to initialize the data in objects by using constructors.

Using Initializer Lists



You can use special syntax called an initializer list to implement one constructor by calling an overloaded constructor.

Avoiding Duplicate Initia lizations

The following code shows an example of overloaded constructors with duplicated initialization code:

```
class Date
{
    public Date( )
    {
        ccyy = 1970;
        mm = 1;
        dd = 1;
    }
    public Date(int year, int month, int day)
    {
        ccyy = year;
        mm = month;
        dd = day;
    }
    private int ccyy, mm, dd;
}
```

Notice the duplication of *dd*, *mm*, and *ccyy* on the left side of the three initializations. This is not extensive duplication, but it is duplication nonetheless, and you should avoid it if possible. For example, suppose you decided to change the representation of a **Date** to one **long** field. You would need to rewrite every **Date** constructor.

Refactoring Duplicate Initializations

A standard way to refactor duplicate code is to extract the common code into its own method. The following code provides an example:

```
class Date
{
    public Date( )
    {
        Init(1970, 1, 1);
    }
    public Date(int year, int month, int day)
    {
        Init(day, month, year);
    }
    private void Init(int year, int month, int day)
    {
        ccyy = year;
        mm = month;
        dd = day;
    }
    private int ccyy, mm, dd;
}
```

This is better than the previous solution. Now if you changed the representation of a **Date** to one **long** field, you would only need to modify **Init**. Unfortunately, refactoring constructors in this way works some of the time but not all of the time. For example, it will not work if you try to refactor the initialization of a readonly field. (This is covered later in this module.) Object-oriented programming languages provide mechanisms to help solve this known problem. For example, in C++ you can use default values. In C# you use initializer lists.

Using an Initializer List

An initializer list allows you to write a constructor that calls another constructor in the same class. You write the initializer list between the closing parenthesis mark and the opening left brace of the constructor. An initializer list starts with a colon and is followed by the keyword **this** and then any arguments between parentheses. For example, in the following code, the default **Date** constructor (the one with no arguments) uses an initializer list to call the second **Date** constructor with three arguments: 1970, 1, and 1.

```
class Date
{
    public Date( ) : this(1970, 1, 1)
    {
    }
    public Date(int year, int month, int day)
    {
        ccyy = year;
        mm = month;
        dd = day;
    }
    private int ccyy, mm, dd;
}
```

This syntax is efficient, it always works, and if you use it you do not need to create an extra **Init** method.

Initializer List Restrictions

There are three restrictions you must observe when initializing constructors:

- You can only use initializer lists in constructors as shown in the following example:

```
class Point
{
    public Point(int x, int y) { ... }
    // Compile-time error
    public void Init( ) : this(0, 0) { }
}
```

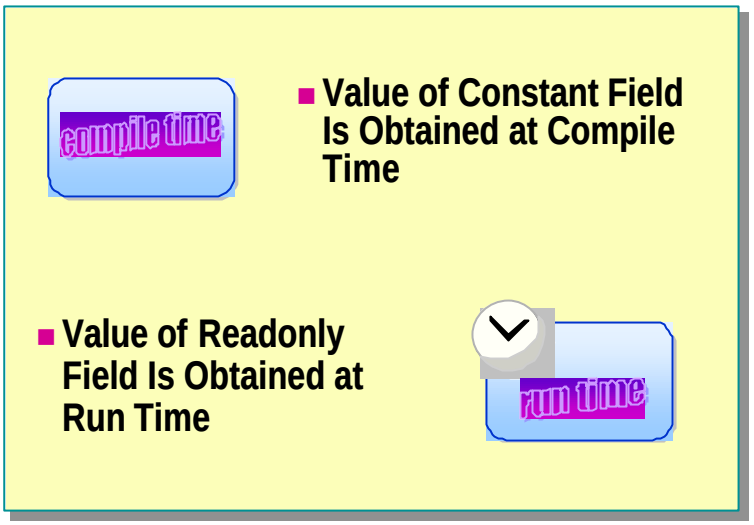
- You cannot write an initializer list that calls itself. The following code provides an example:

```
class Point
{
    // Compile-time error
    public Point(int x, int y) : this(x, y) { }
}
```

- You cannot use the **this** keyword in an expression to create a constructor argument. The following code provides an example:

```
class Point
{
    // Compile-time error
    public Point( ) : this(X(this), Y(this)) { }
    public Point(int x, int y) { ... }
    private static int X(Point p) { ... }
    private static int Y(Point p) { ... }
}
```

Declaring Readonly Variables and Constants



When using constructors, you need to know how to declare readonly variables and constants.

Using Readonly Variables

You can qualify a field as readonly in its declaration, as follows:

```
readonly int nLoopCount = 10;
```

You will get an error if you attempt to change the value at run time.

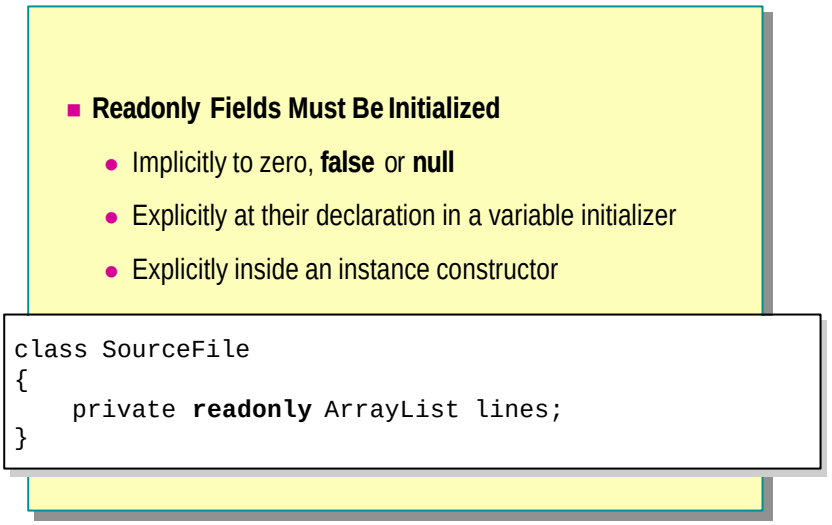
Using Constant Variables

A constant variable represents a constant value that is computed at compile time. Using constant variables, you can define variables whose values never change, as shown in the following example:

```
const int speedLimit = 55;
```

Constants can depend on other constants within the same program as long as the dependencies are not of a circular nature. The compiler automatically evaluates the constant declarations in the appropriate order.

Initializing Readonly Fields



Fields that cannot be reassigned and that must be initialized are called readonly fields. There are three ways to initialize a readonly field:

- Use the default initialization of a readonly field.
- Initialize a readonly field in a constructor.
- Initialize readonly fields by using a variable initializer.

Using the Default Initialization of a Readonly Field

The compiler-generated default constructor will initialize all fields (whether they are readonly or not) to their default value of zero, **false**, or **null**. The following code provides an example:

```
class SourceFile
{
    public readonly ArrayList lines;
}
class Test
{
    static void Main( )
    {
        SourceFile src = new SourceFile( );
        Console.WriteLine(src.lines == null); // True
    }
}
```

There is no **SourceFile** constructor, so the compiler writes a default constructor for you, which will initialize *lines* to **null**. Hence the **WriteLine** statement in the preceding example writes *"True."*

If you declare your own constructor in a class and do not explicitly initialize a readonly field, the compiler will still automatically initialize the field. Following is an example:

```
class SourceFile
{
    public SourceFile( ) { }
    public readonly ArrayList lines;
}
class Test
{
    static void Main( )
    {
        SourceFile src = new SourceFile( );
        Console.WriteLine(src.lines == null); // Still true
    }
}
```

This is not very useful. In this case, the readonly field is initialized to **null**, and it will remain **null** because you cannot reassign a readonly field.

Initializing a Readonly Field in a Constructor

You can explicitly initialize a readonly field in the body of a constructor. Following is an example:

```
class SourceFile
{
    public SourceFile( )
    {
        lines = new ArrayList( );
    }
    private readonly ArrayList lines;
}
```

The statement inside the constructor looks syntactically like an assignment to *lines*, which would not normally be allowed because *lines* is a readonly field. However, the statement compiles because the compiler recognizes that the assignment occurs inside a constructor body and so treats it as an initialization.

An advantage of initializing readonly fields like this is that you can use constructor parameters in the **new** expression. Following is an example:

```
class SourceFile
{
    public SourceFile(int suggestedSize)
    {
        lines = new ArrayList(suggestedSize);
    }
    private readonly ArrayList lines;
}
```

Initializing Readonly Fields Using a Variable Initializer

You can initialize a readonly field directly at its declaration by using a variable initializer. Following is an example:

```
class SourceFile
{
    public SourceFile( )
    {
        ...
    }
    private readonly ArrayList lines = new ArrayList( );
}
```

This is really just convenient shorthand. The compiler conceptually rewrites a variable initialization (whether it is readonly or not) into an assignment inside all constructors. For example, the preceding class will conceptually be converted into the following class:

```
class SourceFile
{
    public SourceFile( )
    {
        lines = new ArrayList( );
        ...
    }
    private readonly ArrayList lines;
}
```

Declaring a Constructor for a Struct

- **The Compiler**
  - Always generates a default constructor. Default constructors automatically initialize all fields to zero.
- **The Programmer**
  - Can declare constructors with one or more arguments. Declared constructors do not automatically initialize fields to zero.
  - Can never declare a default constructor.
  - Can never declare a protected constructor.

The syntax you use to declare a constructor is the same for a struct as it is for a class. For example, the following is a struct called **Point** that has a constructor:

```
struct Point
{
    public Point(int x, int y) { ... }
    ...
}
```

Struct Constructor Restrictions

Although the syntax for struct and class constructors is the same, there are some additional restrictions that apply to struct constructors:

- The compiler always creates a default struct constructor.
- You cannot declare a default constructor in a struct.
- You cannot declare a protected constructor in a struct.
- You must initialize all fields.

The Compiler Always Creates a Default Struct Constructor

The compiler always generates a default constructor, regardless of whether you declare constructors yourself. (This is unlike the situation with classes, in which the compiler-generated default constructor is only generated if you do not declare any constructors yourself.) The compiler generated struct constructor initializes all fie lds to zero, false, or null.

```
struct SPoint
{
    public SPoint(int x, int y) { ... }
    ...
    static void Main( )
    {
        // Okay
        SPoint p = new SPoint( );
    }
}
class CPoint
{
    public CPoint(int x, int y) { ... }
    ...
    static void Main( )
    {
        // Compile-time error
        CPoint p = new CPoint( );
    }
}
```

This means that a struct value created with

```
SPoint p = new SPoint( );
```

creates a new struct value on the stack (using **new** to create a struct does not acquire memory from the heap) and initializes the fields to zero. There is no way to change this behavior.

However, a struct value created with

```
SPoint p;
```

still creates a struct value on the stack but does not initialize any of the fields (so any field must be definitely assigned before it can be referenced). Following is an example:

```
struct SPoint
{
    public int x, y;
    ...
    static void Main( )
    {
        SPoint p1;
        Console.WriteLine(p1.x); // Compile-time error
        SPoint p2;
        p2.x = 0;
        Console.WriteLine(p2.x); // Okay
    }
}
```

Tip Ensure that any struct type that you define is valid with all fields set to zero.

You Cannot Declare a Default Constructor in a Struct

The reason for this restriction is that the compiler always creates a default constructor in a struct (as just described) so you would end up with a duplicate definition.

```
class CPoint
{
    // Okay because CPoint is a class
    public CPoint( ) { ... }
    ...
}
struct SPoint
{
    // Compile-time error because SPoint is a struct
    public SPoint( ) { ... }
    ...
}
```

You can declare a struct constructor as long as it expects at least one argument. If you declare a struct constructor it will not automatically initialize any field to a default value (unlike the compiler generated struct default constructor which will).

```
struct SPoint
{
     public SPoint(int x, int y) { ... }
    ...
}
```

You Cannot Declare a Protected Constructor in a Struct

The reason for this restriction is that you can never derive other classes or structs from a struct, and so protected access would not make sense, as shown in the following example:

```
class CPoint
{
    // Okay
    protected CPoint(int x, int y) { ... }
}
struct SPoint
{
    // Compile-time error
    protected SPoint(int x, int y) { ... }
}
```

You Must Initialize All Fields

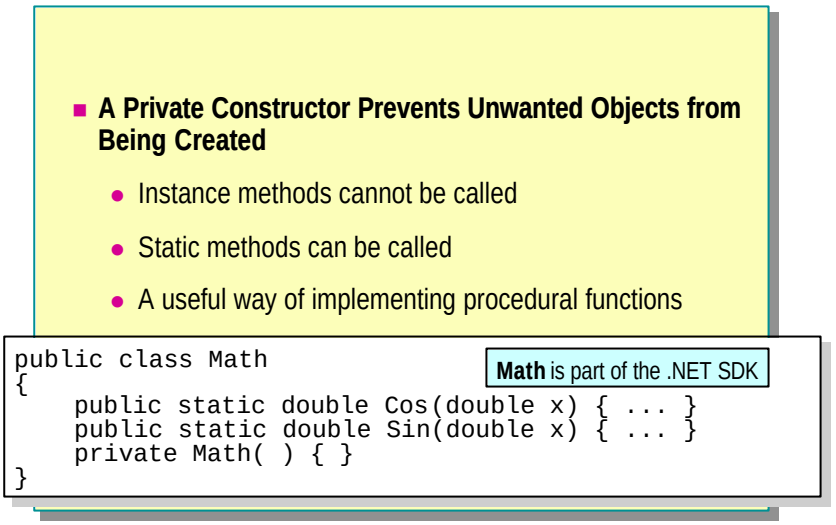
If you declare a class constructor that fails to initialize a field, the compiler will ensure that the field nevertheless retains its default zero initialization. The following code provides an example:

```
class CPoint
{
    private int x, y;
    public CPoint(int x, int y) { /*nothing*/ }
    // Okay. Compiler ensures that x and y are initialized to
    // zero.
}
```

However, if you declare a struct constructor that fails to initialize a field, the compiler will generate a compile-time error:

```
struct SPoint1 // Okay: initialized when declared
{
    private int x = 0, y = 0;
    public SPoint1(int x, int y) { }
}
struct SPoint2 // Okay: initialized in constructor
{
    private int x, y;
    public SPoint2(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
struct SPoint3 // Compile-time error
{
    private int x, y;
    public SPoint3(int x, int y) { }
}
```

Using Private Constructors



So far, you have learned how to use public constructors. C# also provides private constructors, which are useful in some applications.

Using Private Constructors for Procedural Functions

Object-oriented programming offers a powerful paradigm for structuring software in many diverse domains. However, it is not a universally applicable paradigm. For example, there is nothing object oriented about calculating the sine or cosine of a double-precision floating-point number.

Declaring Functions

The most intuitive way to calculate a sine or cosine is to use global functions defined outside an object, as follows:

```
double Cos(double x) { ... }
double Sin(double x) { ... }
```

The preceding code is not allowable in C#. Global functions are possible in procedural languages such as C and in hybrid languages such as C++, but they are not allowed in C#. In C#, functions must be declared inside a class or struct, as follows:

```
class Math
{
    public double Cos(double x) { ... }
    public double Sin(double x) { ... }
}
```

Declaring Static vs. Instance Methods

The problem with the technique in the preceding example is that, because **Cos** and **Sin** are instance methods, you are forced to create a **Math** object from which to invoke **Sin** or **Cos**, as shown in the following code:

```
class Cumbersome
{
    static void Main( )
    {
        Math m = new Math( );
      double answer;
      answer = m.Cos(42.0);
      // Or
      answer = new Math( ).Cos(42.0);
  }
}
```

However, you can easily solve this by declaring **Cos** and **Sin** as static methods, as follows:

```
class Math
{
    public static double Cos(double x) { ... }
    public static double Sin(double x) { ... }
}
class LessCumbersome
{
    static void Main( )
    {
      double answer = Math.Cos(42.0);
  }
}
```

Benefits of Static Methods

If you declare **Cos** as a static method, the syntax for using **Cos** becomes:

- Simpler

  You have only one way to call **Cos** (by means of **Math**), whereas in the previous example you had two ways (by means of **m** and by means of `new Math()`).

- Faster

  You no longer need to create a new **Math** object.

One slight problem remains. The compiler will generate a default constructor with public access, allowing you to create **Math** objects. Such objects can serve no purpose because the **Math** class contains static methods only. There are two ways you can prevent **Math** objects from being created:

- Declare **Math** as an abstract class.

  This is not a good idea. The purpose of abstract classes is to be derived from.

- Declare a private **Math** constructor.

  This is a better solution. When you declare a constructor in the **Math** class, you prevent the compiler from generating the default constructor, and if you also declare the constructor as private, you stop **Math** objects from being created. The private constructor also prevents **Math** from being used as a base class.

The Singleton Pattern

The intent of the Singleton pattern (which is discussed in *Design Patterns: Elements of Reusable Object-Oriented Software*) is to "ensure a class only has one instance, and provide a global point of access to it." The technique of declaring a class by using a private constructor and static methods is sometimes suggested as a way to implement the Singleton pattern.

Note A key aspect of the Singleton pattern is that a class has a single instance. With a private constructor and static methods, there is no instance at all. The canonical implementation of the Singleton pattern is to create a static method that gives access to the single instance, and this instance is then used to call instance methods.

Using Static Constructors

- **Purpose**
  - Called by the class loader at run time
  - Can be used to initialize static fields
  - Guaranteed to be called before instance constructor
- **Restrictions**
  - Cannot be called
  - Cannot have an access modifier
  - Must be parameterless

Just as an instance constructor guarantees that an object is in a well-defined initial state before it is used, a static constructor guarantees that a class is in a well-defined initial state before it is used.

Loading Classes at Run Time

C# is a dynamic language. When the Common Language Runtime is running a Microsoft® .NET program, it often encounters code that uses a class that has not yet been loaded. In these situations, execution is momentarily suspended, the class is dynamically loaded, and then execution continues.

Initializing Classes at Load Time

C# ensures that a class is always initialized before it is used in code in any way. This guarantee is achieved by using static constructors.

You can declare a static constructor like an instance constructor but prefix it with the keyword **static**, as follows:

```
class Example
{
    static Example( ) { ... }
}
```

After the class loader loads a class that will soon be used, but before it continues normal execution, it executes the static constructor for that class. Because of this process, you are guaranteed that classes are always initialized before they are used. The specific guarantees that the class loader provides are as follows:

- The static constructor for a class is executed before any instances of the class are created.
- The static constructor for a class is executed before any static member of the class is referenced.
- The static constructor for a class is executed before the static constructor of any of its derived classes is executed.
- The static constructor for a class never executes more than once.

Static Field Initializations and Static Constructors

The most common use for a static constructor is to initialize the static fields of a class. This is because when you initialize a static field directly at its point of declaration, the compiler conceptually converts the initialization into an assignment inside the static constructor. In other words

```
class Example
{
    private static Wibble w = new Wibble( );
}
```

is effectively converted by the compiler into

```
class Example
{
    static Example( )
    {
        w = new Wibble( );
    }
    private static Wibble w;
}
```

Static Constructor Restrictions

Understanding the following four restrictions on the syntax of static constructors will help you understand how the Common Language Runtime uses static constructors:

- You cannot call a static constructor.
- You cannot declare a static constructor with an access modifier.
- You cannot declare a static constructor with parameters.
- You cannot use the **this** keyword in a static constructor.

You Cannot Call a Static Constructor

A static constructor must be called before any instances of the class are referenced in code. If the responsibility for enforcing this rule were given to programmers rather than the .NET runtime, eventually programmers would fail to meet the responsibility. They would forget to make the call, or, perhaps worse, they would call the static constructor more than once. The .NET runtime avoids these potential problems by disallowing calls to static constructors in code. Only the .NET runtime can call a static constructor.

```
class Point
{
    static Point( ) { ... }
    static void Main( )
    {
        Point.Point( ); // Compile-time error
    }
}
```

You Cannot Declare a Static Constructor with an Access Modifier

Because you cannot call a static constructor, declaring a static constructor with an access modifier does not make sense and causes a compile-time error:

```
class Point
{
    public static Point( ) { ... } // Compile-time error
}
```

You Cannot Declare a Static Constructor with Parameters

Because you cannot call a static constructor, declaring a static constructor with parameters does not make sense and causes a compile-time error. This also means that you cannot declare overloaded static constructors. Following is an example:

```
class Point
{
    static Point(int x) { ... } // Compile-time error
}
```

You Cannot Use the this Keyword in a Static Constructor

Because a static constructor initializes the class and not object instances, it does not have an implicit **this** reference, so any attempt to use the **this** keyword results in a compile-time error:

```
class Point
{
    private int x, y;
    static Point( ) : this(0,0)   // Compile-time error
    {
        this.x = 0; // Compile-time error
        this.y = 0; // Compile-time error
    }
    ...
}
```

Lab 9.1: Creating Objects

Objectives

In this lab, you will modify the **BankAccount** class that you created in the previous labs so that it uses constructors. You will also create a new class, **BankTransaction**, and use it to store information about the transactions (deposits and withdrawals) performed on an account.

After completing this lab, you will be able to:

- Override the default constructor.
- Create overloaded constructors.
- Initialize **readonly** data.

Prerequisites

Before working on this lab, you must be able to:

- Create classes and instantiate objects.
- Define and call methods.

You should also have completed Lab 8. If you did not complete Lab 8, you can use the solution code provided.

Estimated time to complete this lab: 60 minutes

Exercise 1
Implementing Constructors

In this exercise, you will modify the **BankAccount** class that you created in the previous labs. You will remove the methods that populate the account number and account type instance variables and replace them with a series of constructors that can be used when a **BankAccount** is instantiated.

You will override the default constructor to generate an account number (by using the technique that you used earlier), set the account type to **Checking**, and set the balance to zero.

You will also create three more constructors that take different combinations of parameters:

- The first will take an **AccountType** . The constructor will generate an account number, set the balance to zero, and set the account type to the value passed in.
- The second will take a **decimal**. The constructor will generate an account number, set the account type to **Checking**, and set the balance to the value passed in.
- The third will take an **AccountType** and a **decimal** The constructor will generate an account number, set the account type to the value of the **AccountType** parameter, and set the balance to the value of the **decimal** parameter.

↙ To create the default constructor

1. Open the Constructors.sln project in the *Lab Files*\Lab09\Starter\Constructors folder.
2. In the **BankAccount** class, delete the **Populate** method.
3. Create a default constructor, as follows:
   a. The name is **BankAccount**.
   b. It is public.
   c. It takes no parameters.
   d. It has no return type.
   e. The body of the constructor should generate an account number by using the **NextNumber** method, set the account type to **AccountType.Checking**, and initialize the account balance to zero.

   The completed constructor is as follows:

```
public BankAccount( )
{
  accNo = NextNumber( );
  accType = AccountType.Checking;
  accBal = 0;
}
```

↙ To create the remaining constructors

1. Add another constructor that takes a single **AccountType** parameter called **aType**. The constructor should:
   a. Generate an account number as before.
   b. Set *accType* to **aType**.
   c. Set *accBal* to zero.
2. Define another constructor that takes a single **decimal** parameter called **aBal**. The constructor should:
   a. Generate an account number.
   b. Set *accType* to **AccountType.Checking**.
   c. Set *accBal* to **aBal**.
3. Define a final constructor that takes two parameters: an **AccountType** called **aType** and a **decimal** called **aBal**. The constructor should:
   a. Generate an account number.
   b. Set *accType* to **aType**.
   c. Set *accBal* to **aBal**.

   The completed code for all three constructors is as follows:

```
public BankAccount(AccountType aType)
{
   accNo = NextNumber( );
   accType = aType;
   accBal = 0;
}

public BankAccount(decimal aBal)
{
   accNo = NextNumber( );
   accType = AccountType.Checking;
   accBal = aBal;
}

public BankAccount(AccountType aType, decimal aBal)
{
   accNo = NextNumber( );
   accType = aType;
   accBal = aBal;
}
```

↙ To test the constructors

1. In the **Main** method of the **CreateAccount** class, define four **BankAccount** variables called *acc1*, *acc2*, *acc3*, and *acc4*.
2. Instantiate *acc1* by using the default constructor.
3. Instantiate *acc2* by using the constructor that takes only an **AccountType**. Set the type of *acc2* to **AccountType.Deposit**.
4. Instantiate *acc3* by using the constructor that takes only a **decimal** balance. Set the balance of *acc3* to 100.
5. Instantiate *acc4* by using the constructor that takes an **AccountType** and a **decimal** balance. Set the type of *acc4* to **AccountType.Deposit**, and set the balance to 500.
6. Use the **Write** method (supplied with the **CreateAccount** class) to display the contents of each account one by one. The completed code is as follows:

```
static void Main( )
{
   BankAccount acc1, acc2, acc3, acc4;

   acc1 = new BankAccount( );
   acc2 = new BankAccount(AccountType.Deposit);
   acc3 = new BankAccount(100);
   acc4 = new BankAccount(AccountType.Deposit, 500);

   Write(acc1);
   Write(acc2);
   Write(acc3);
   Write(acc4);
}
```

7. Compile the project and correct any errors. Execute it, and check that the output is as expected.

Exercise 2
Initializing readonly Data

In this exercise, you will create a new class called **BankTransaction**. It will hold information about a deposit or withdrawal transaction that is performed on an account.

Whenever the balance of an account is changed by means of the **Deposit** or **Withdraw** method, a new **BankTransaction** object will be created. The **BankTransaction** object will contain the current date and time (generated from **System.DateTime**) and the amount added (positive) or deducted (negative) from the account. Because transaction data cannot be changed once it is created, this information will be stored in two **readonly** instance variables in the **BankTransaction** object.

The constructor for **BankTransaction** will take a single decimal parameter, which it will use to populate the transaction amount instance variable. The date and time instance variable will be populated by **DateTime.Now**, a property of **System.DateTime** that returns the current date and time.

You will modify the **BankAccount** class to create transactions in the **Deposit** and **Withdraw** methods. You will store the transactions in an instance variable in the **BankAccount** class of type **System.Collections.Queue**. A queue is a data structure that holds an ordered list of objects. It provides methods for adding elements to the queue and for iterating through the queue. (Using a queue is better than using an array because a queue does not have a fixed size: it will grow automatically as more transactions are added.)

↙ To create the BankTransaction class

1. Open the Constructors.sln project in the *Lab Files*\Lab09\Starter\Constructors folder, if it is not already open.
2. Add a new class called **BankTransaction**.
3. In the **BankTransaction** class, remove the **namespace** directive together with the first opening brace ({), and the final closing brace (}). (You will learn more about namespaces in a later module.)
4. In the summary comment, add a brief description of the **BankTransaction** class. Use the description above to help you.
5. Delete the default constructor created by Visual Studio.
6. Add the following two private **readonly** instance variables:
   a. A **decimal** called *amount.*
   b. A **DateTime** variable called *when*. The **System.DateTime** structure is useful for holding dates and times, and contains a number of methods for manipulating these values.

7. Add two accessor methods, called **Amount** and **When**, that return the values of the two instance variables:

```
private readonly decimal amount;
private readonly DateTime when;
...
public decimal Amount( )
{
  return amount;
}

public DateTime When( )
{
  return when;
}
```

↙ To create the constructor

1. Define a public constructor for the **BankTransaction** class. It will take a decimal parameter called *tranAmount* that will be used to populate the *amount* instance variable.
2. In the constructor, initialize *when* with **DateTime.Now**.

> **Tip** **DateTime.Now** is a property and not a method, so you do not need to use parentheses.

The completed constructor is as follows:

```
public BankTransaction(decimal tranAmount)
{
     amount = tranAmount;
  when = DateTime.Now;
}
```

3. Compile the project and correct any errors.

↙ To create transactions

1. As described above, transactions will be created by the **BankAccount** class and stored in a queue whenever the **Deposit** or **Withdraw** method is invoked. Return to the **BankAccount** class.
2. Before the start of the **BankAccount** class, add the following **using** directive:

```
using System.Collections;
```

3. Add a private instance variable call *tranQueue* to the **BankAccount** class. Its data type should be **Queue** and it should be initialized with a new empty queue:

```
private Queue tranQueue = new Queue( );
```

4. In the **Deposit** method, before returning, create a new transaction using the deposit amount as the parameter, and append it to the queue by using the **Enqueue** method, as follows:

```
public decimal Deposit(decimal amount)
{
  accBal += amount;
  BankTransaction tran = new BankTransaction(amount);
  tranQueue.Enqueue(tran);
  return accBal;
}
```

5. In the **Withdraw** method, if there are sufficient funds, create a transaction and append it to *tranQueue* as in the **Deposit** method, as follows:

```
public bool Withdraw(decimal amount)
{
  bool sufficientFunds = accBal >= amount;
  if (sufficientFunds) {
    accBal -= amount;
    BankTransaction tran = new BankTransaction(-amount);
    tranQueue.Enqueue(tran);
  }
  return sufficientFunds;
}
```

Note For the **Withdraw** method, the value passed to the constructor of the **BankTransaction** should be the amount being withdrawn preceded by the negative sign.

↙ To test transactions

1. For testing purposes, add a public method called **Transactions** to the **BankAccount** class. Its return type should be **Queue**, and the method should return *tranQueue*. You will use this method for displaying transactions in the next step. The method will be as follows:

```
public Queue Transactions( )
{
  return tranQueue;
}
```

2. In the **CreateAccount** class, modify the **Write** method to display the details of transactions for each account. Queues implement the **IEnumerable** interface, which means that you can use the **foreach** construct to iterate through them.

3. In the body of the **foreach** loop, print out the date and time and the amount for each transaction, by using the **When** and **Amount** methods, as follows:

```
static void Write(BankAccount acc)
{
  Console.WriteLine("Account number is {0}",
acc.Number( ));
  Console.WriteLine("Account balance is {0}",
acc.Balance( ));
  Console.WriteLine("Account type is {0}", acc.Type( ));
  Console.WriteLine("Transactions:");
  foreach (BankTransaction tran in acc.Transactions( ))
  {
    Console.WriteLine("Date/Time: {0}\tAmount: {1}",
➥tran.When( ), tran.Amount( ));
  }
  Console.WriteLine( );
}
```

4. In the **Main** method, add statements to deposit and withdraw money from each of the four accounts (*acc1*, *acc2*, *acc3*, and *acc4*).

5. Compile the project and correct any errors.

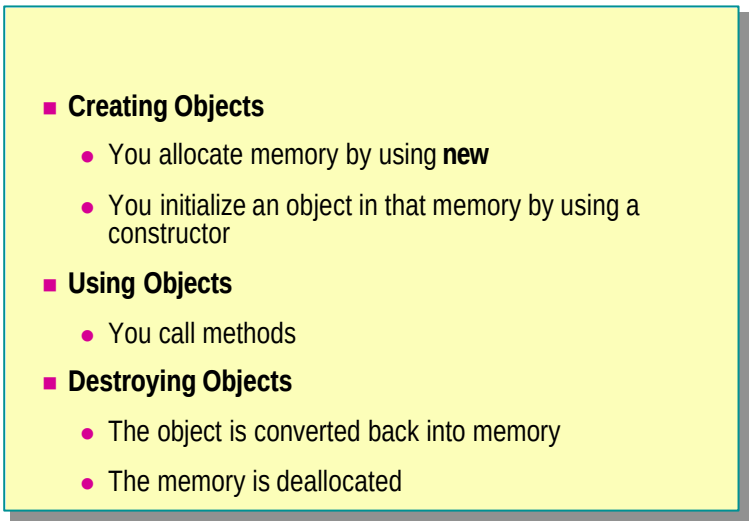
6. Execute the project. Examine the output and check whether transactions are displayed as expected.

Objects and Memory

- Object Lifetime
- Objects and Scope
- Garbage Collection

In this section, you will learn what happens when an object, as opposed to a value, goes out of scope or is destroyed and about the role of garbage collection in this process.

Object Lifetime



In C#, destroying an object is a two-step process that corresponds to and reverses the two-step object creation process.

Creating Objects

In the first section, you learned that creating a C# object for a reference type is a two-step process, as follows:

1. Use the **new** keyword to acquire and allocate memory.
2. Call a constructor to turn the raw memory acquired by **new** into an object.

Destroying Objects

Destroying a C# object is also a two-step process:

1. De-initialize the object.

   This converts the object back into raw memory. It is done by the destructor or the **Finalize** method. This is the reverse of the initialization performed by the constructor. You can control what happens in this step by writing your own destructor or finalize method.

2. The raw memory is deallocated; that is, it is given back to the memory heap.

   This is the reverse of the allocation performed by **new**. You cannot change the behavior of this step in any way.

Objects and Scope

- **The Lifetime of a Local Value Is Tied to the Scope in Which It Is Declared**
  - Short lifetime (typically)
  - Deterministic creation and destruction
- **The Lifetime of a Dynamic Object Is Not Tied to Its Scope**
  - A longer lifetime
  - A non-deterministic destruction

Unlike values such as ints and structs, which are allocated on the stack and are destroyed at the end of their scope, objects are allocated on the heap and are not destroyed at the end of their scope.

Values

The lifetime of a local value is tied to the scope in which it is declared. Local values are variables that are allocated on the stack and not through the **new** operator. This means that if you declare a variable whose type is one of the primitives (such as int), enum, or struct, you cannot use it outside the scope in which you declare it. For example, in the following code fragment, three values are declared inside a **for** statement, and so go out of scope at the end of the **for** statement:

```
struct Point { public int x, y; }
enum Season { Spring, Summer, Fall, Winter }
class Example
{
    void Method( )
    {
        for (int i = 0; i < limit; i++) {
            int x = 42;
            Point p = new Point( );
            Season s = Season.Winter;
        }
        x = 42;            // Compile-time error
        p = new Point( );  // Compile-time error
        s = Season.Winter; // Compile-time error
    }
}
```

Note In the previous example, it appears as though a **new Point** is created. However, because **Point** is a **struct**, **new** does not allocate memory from the heap. The "new" **Point** *is* created on the stack.

This means that local values have the following characteristics:

- Deterministic creation and destruction

  A local variable is created when you declare it, and is destroyed at the end of the scope in which it is declared. The start point and the end point of the value's life are deterministic; that is, they occur at known, fixed times.

- Usually very short lifetimes

  You declare a value somewhere in a method, and the value cannot exist beyond the method call. When you return a value from a method, you return a copy of the value.

Objects

The lifetime of an object is not tied to the scope in which it is created. Objects are initialized in heap memory allocated through the **new** operator. For example, in the following code, the reference variable *eg* is declared inside a **for** statement. This means that *eg* goes out of scope at the end of the **for** statement and is a local variable. However, *eg* is initialized with a **new Example()** object, and this object does not go out of scope with *eg*. Remember: a reference variable and the object it references are different things.

```
class Example
{
    void Method( )
    {
        for (int i = 0; i < limit; i++) {
            Example eg = new Example( );
            ...
        }
        // eg is out of scope
        // Does eg still exist?
        // Does the object still exist?
    }
}
```

This means that objects typically have the following characteristics:

- Non-deterministic destruction

  An object is created when you create it, but, unlike a value, it is it not destroyed at the end of the scope in which it is created. The creation of an object is deterministic, but the destruction of an object is not. You cannot control exactly when an object will be destroyed.

- Longer lifetimes

  Because the life of an object is not tied to the method that creates it, an object can exist well beyond a single method call.

Garbage Collection

- **You Cannot Destroy Objects in C#**
  - C# does not have an opposite of **new** (such as **delete**)
  - This is because an explicit delete function is a prime source of errors in other languages
- **Garbage Collection Destroys Objects for You**
  - It finds unreachable objects and destroys them for you
  - It finalizes them back to raw unused heap memory
  - It typically does this when memory becomes low

So far, you have seen that you create objects in C# in exactly the same way that you create objects in other languages, such as C++. You use the **new** keyword to allocate memory from the heap, and you call a constructor to convert that memory into an object. However, as far as the method for the destruction of objects, there is no similarity between C# and its predecessors.

You Cannot Destroy Objects in C#

In many programming languages, you can explicitly control when an object will be destroyed. For example, in C++ you can use a **delete** expression to de-initialize (or finalize) the object (turn it back into raw memory) and then return the memory to the heap. In C#, there is no way to explicitly destroy objects. In many ways, this restriction is a useful one because programmers often misuse the ability to explicitly destroy objects by:

- Forgetting to destroy objects.

  If you had the responsibility for writing the code that destroyed an object, you might sometimes forget to write the code. This can happen in C++ code, and this is a problematic bug that causes the user's computer to get slower as the program uses more memory. This is known as *memory leak*. Often the only way to reclaim the lost memory is to shut down and then restart the offending program.

- Attempting to destroy the same object more than once.

  You might sometimes accidentally attempt to destroy the same object more than once. This can happen in C++ code, and it is a serious bug with undefined consequences. The problem is that when you destroy the object the first time, the memory is reclaimed and can be used to create a new object, probably of a completely different class. When you then attempt to destroy the object the second time, the memory refers to a completely different object!

- Destroying an active object.

  You might sometimes destroy an object that was still being referred to in another part of the program. This is also a serious bug known as the *dangling pointer problem*, and it also has undefined consequences.

Garbage Collection Destroys Objects for You

In C#, you cannot destroy an object explicitly in code. Instead, C# has a *garbage collection*, which destroys objects for you. Garbage collection is completely automatic. It ensures that:

- Objects are destroyed.

  However, garbage collection does not specify exactly when the object will be destroyed.

- Objects are destroyed only once.

  This means that you cannot get the undefined behavior of double deletion that is possible in C++. This is important because it helps to ensure that a C# program always behaves in a well-defined way.

- Only unreachable objects are destroyed.

  Garbage collection ensures that an object is never destroyed if another object holds a reference to it. Garbage collection only destroys an object when no other object holds a reference to it. The ability of one object to reach another object through a reference variable is called *reachability*. Only unreachable objects are destroyed. It is the function of garbage collection to follow all of the object references to determine which objects are reachable and hence, by a process of elimination, to find the remaining unreachable objects. This can be a time-consuming operation, so garbage collection only collects garbage to reclaim unused memory when memory becomes low.

Note You can also invoke garbage collection explicitly in your code, but it is not recommended. Let the .NET runtime manage memory for you.

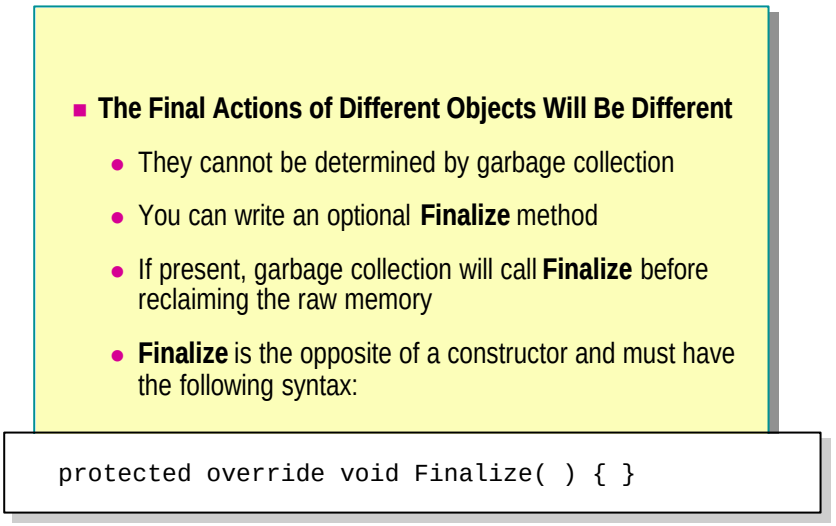
◆ Using Destructors

- The Finalize Method
- Writing Destructors
- Destructors and the Finalize Method
- Warnings About Destructor Timing
- GC.SuppressFinalize( )
- Using the Disposal Design Pattern
- Using IDisposable

A destructor is a special method that you use to de-initialize an object. In this section, you will learn how to use destructors and the **Finalize** method to control the destruction of object.

Note This course is based on the Beta 1 version of Microsoft Visual Studio.NET. In Beta 2 and subsequent versions of Visual Studio.NET, destructors will always be executed, even if it is only at the end of the program. This feature is not available in Beta 1.

The Finalize Method



You have already seen that destroying an object is a two-step process. In the first step, the object is converted back into raw memory. In the second step, the raw memory is returned to the heap to be recycled. Garbage collection completely automates the second step of this process for you.

However, the actions required to finalize a specific object back into raw memory to clean it up will depend on the specific object. This means that garbage collection cannot automate the first step for you. If there are any specific statements that you want an object to execute as it is picked up by garbage collection and just before its memory is reclaimed, you need to write these statements yourself in a method called **Finalize**.

Finalization

When garbage collection is destroying an unreachable object, it will check whether the class of the object has its own **Finalize** method. If the class has a **Finalize** method, it will call the method before recycling the memory back to the heap. The statements you write in the **Finalize** method will be specific to the class, but the signature of the **Finalize** method must take a particular form:

- No arguments required

  Remember, you do not call **Finalize**; garbage collection does.

- **void** return type

  The purpose of **Finalize** is not to return a result but to perform an action. You might think it reasonable for **Finalize** to return a bool to indicate whether the object was successfully finalized. The problem with this approach is that it would not really help. When does garbage collection call **Finalize**, and what would it return any value to?

- Use **override** modifier

  All classes inherit from the **Object** class. The **Object** class has a virtual method called **Finalize**. Your **Finalize** must override **Object**.**Finalize**. This will become clearer after you have learned about inheritance and polymorphism.

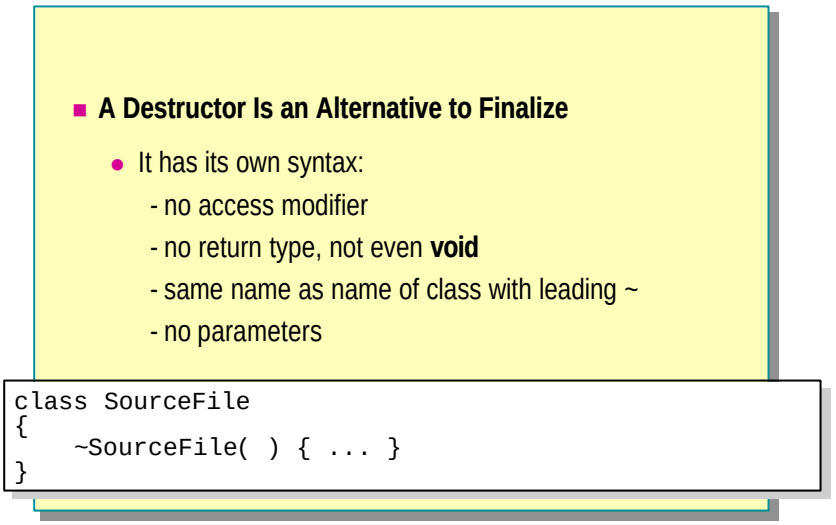
- Protected access

  Finalize in the **Object** base class has protected access, and when you override a method you are not allowed to change that method's access. This will become clearer after you have completed Module 10, "Inheritance in C#."

The following code shows an example of the **SourceFile** class with an embedded **StreamReader** whose **Finalize** method closes the **StreamReader**.

```
class SourceFile
{
    public SourceFile(string name)
    {
        File src = new File(name);
        reader = src.OpenText( );
    }
    ...
    protected override void Finalize( )
    {
        reader.Close( );
    }
    ...
    private StreamReader reader;
}
```

Writing Destructors



You can write a destructor as an alternative to the **Finalize** method. The relationship between **Finalize** and the destructor is extremely close and is explained in detail in the next topic. The **Finalize** method and destructors share the following features:

- No access modifier

  You do not call the destructor; garbage collection does.

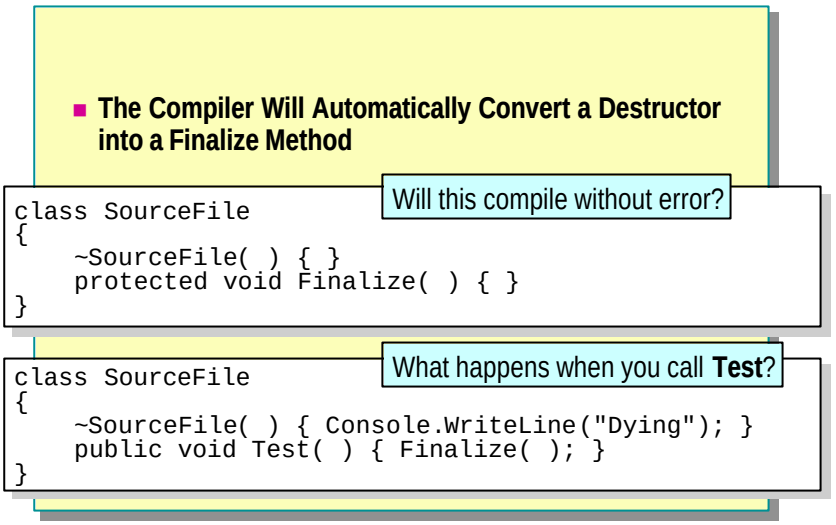
- No return type

  The purpose of the destructor is not to return a value but to perform the required clean-up actions.

- No parameters can be passed

  Again, you do not call the destructor, so you cannot pass it any arguments. Note that this means that the destructor cannot be overloaded.

Destructors and the Finalize Method



When you write a destructor for a class, the compiler will automatically convert that destructor into a **Finalize** method for that class. A **Finalize** method generated from a destructor and a **Finalize** method that you have written yourself are almost identical. In particular, garbage collection treats them the same.

One important difference between them is that a destructor will be converted into a **Finalize** method that automatically calls **Finalize** on its base class.

Question 1

Examine the following code. Will it compile without error?

```
class SourceFile
{
    ~SourceFile( ) { }
    protected void Finalize( ) { }
}
```

The code example will generate an error when compiled. The destructor is converted into a **Finalize** method that has no arguments. This means that after the compiler conversion has taken place, there will be two methods called **Finalize** that expect no arguments. This is not allowed and will cause the compiler to generate a “duplicate definition” diagnostic message.

Question 2

Examine the following code. What will happen when you call **Test**?

```
class SourceFile
{
    ~SourceFile( )
    {
        ...
        Console.WriteLine("Dying");
    }
    public void Test( )
    {
        Finalize( );
    }
}
```

To answer this question, remember that the compiler will convert the destructor into a **Finalize** method. In other words, the above example will become the following:

```
class SourceFile
{
    protected void override Finalize( )
    {
        ...
        Console.WriteLine("Dying");
    }
    public void Test( )
    {
        Finalize( );
    }
}
```

This means that when you call **Test**, the **Console.WriteLine** statement inside the destructor will be executed, writing "Dying" to the console.

Important This second question also shows that you can explicitly call the **Finalize** method on an object. But remember, garbage collection will also call the **Finalize** method on the object when the object is garbage collected, leading to the same object being finalized more than once! The solution to this multiple finalization problem is covered later in this section.

Note You cannot declare destructors or **Finalize** methods in structs.

Warnings About Destructor Timing

- **You Cannot Rely on Destructors Being Called**
  - Garbage collection is only called if memory becomes low
  - What if memory never becomes low?
- **The Order of Destruction Is Undefined**
  - Not necessarily the reverse order of construction

You have seen that in C# garbage collection is responsible for destroying objects when they are unreachable. This is unlike other languages such as C++, in which the programmer is responsible for explicitly destroying objects. Shifting the responsibility for destroying objects away from the programmer is a good thing, but it you cannot control exactly when a C# object is destroyed. This is sometimes referred to as *non-deterministic finalization*.

You Cannot Rely on Destructors Being Called

When garbage collection is called upon to destroy some objects, it must find the objects that are unreachable, call their **Finalize** methods (if they have them), and then recycle their memory back to the heap. This is a complicated process (the details of which are beyond the scope of this course), and it takes a fair amount of time. Consequently, garbage collection does not run unless it needs to (and when it does it runs in its own thread).

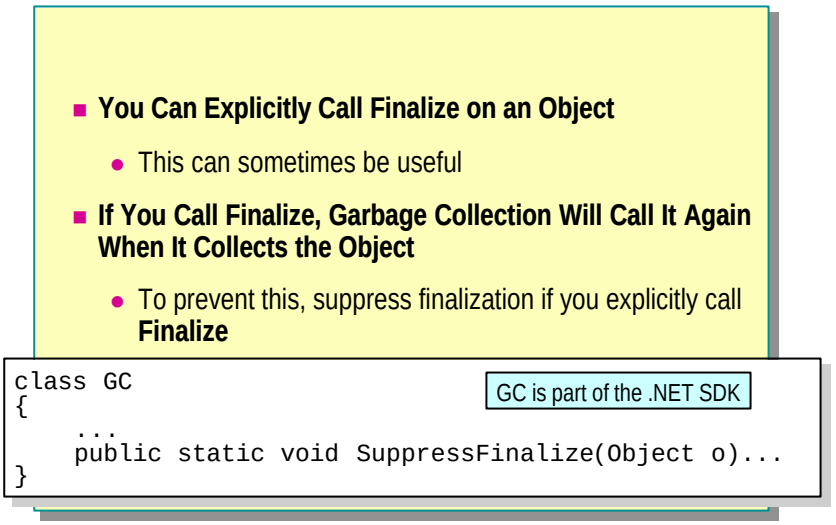
The one time when garbage collection must run is when the heap runs out of memory. But this means that if your program starts, runs, and then shuts down without getting close to using the entire heap, your **Finalize** methods may never get called, and if they do, it will only be when the program shuts down. In many cases, this is perfectly acceptable. However, there are situations in which you must ensure that your **Finalize** methods are called at known points in time. You will learn how to deal with these situations later in this section.

The Order of Destruction Is Undefined

In languages like C++, you can explicitly control when objects are created and when objects are destroyed. In C#, you can control the order in which you create objects but you cannot control the order in which they are destroyed. This is because you do not destroy the objects at all—garbage collection does.

In C#, the order of the creation of objects does not determine the order of the destruction of those objects. They can be destroyed in any order, and many other objects might be destroyed in between. However, in practice this is rarely a problem because garbage collection guarantees that an object will never be destroyed if it is reachable. If one object holds a reference to a second object, the second object is reachable from the first object. This means that the second object will never be destroyed before the first object.

GC.SuppressFinalize( )



You can explicitly call the **Finalize** method, but this creates a potential problem. If an object has a **Finalize** method, garbage collection will see it and will also call it when it destroys the object. The following code provides an example:

```
class DoubleFinalization
{
    ~DoubleFinalization( )
    {
        ...
    }
    public void Dispose( )
    {
        Finalize( );
    }
    ...
}
```

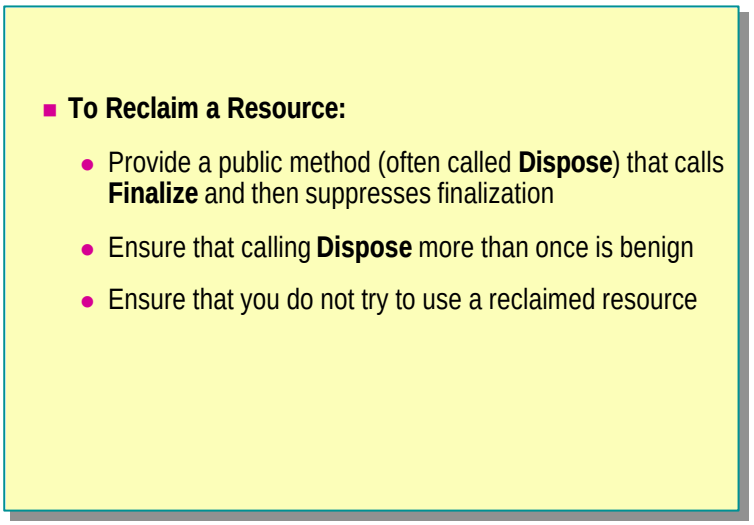
The problem with this example is that if you call **Dispose**, it will call **Finalize** (generated from the destructor). Then, when the object is garbage collected, **Finalize** will be called again.

To avoid duplicate finalization, you can call the **SuppressFinalize** method of the **GC** class and pass in the object that already had its **Finalize** method called. The following code provides an example:

```
class SingleFinalization
{
    ~SingleFinalization( )
    {
        ...
    }
    public void Dispose( )
    {
        Finalize( );
        GC.SuppressFinalize(this);
    }
    ...
}
```

Note There are several more problems related to this technique. These problems are explored in the next topic.

Using the Disposal Design Pattern



If you need to reclaim a resource and you cannot wait for garbage collection to call **Finalize** implicitly for you, you can provide a public method that calls **Finalize**.

Memory Is Not the Only Resource

Memory is the most common resource that your programs use, and you can rely on garbage collection to reclaim unreachable memory when the heap becomes low. However, memory is not the only resource. Other fairly common resources that your program might use include file handles and mutex locks. Often these other kinds of resources are in much more limited supply than memory, or need to be released quickly.

The Disposal Method Design Pattern

In these situations, you cannot rely on garbage collection to perform the release by means of a **Finalize** method, because, as you have seen, you cannot know when garbage collection will call **Finalize**. Instead, you should write a public method that releases the resource, and then make sure to call this method at the right point in the code. These methods are called Disposal Methods. (This is a well-known pattern, but it is not in *Design Patterns: Elements of Reusable Object-Oriented Software*.) In C#, there are three major points that you need to remember when implementing a Disposal Method:

- Remember to call **SuppressFinalize**.
- Ensure that the Disposal Method can be called repeatedly.
- Avoid using a released resource.

Calling SuppressFinalize

The following code shows how to call **SuppressFinalize**:

```
class Example
{
    ...
    ~Example( )
    {
        rare.Dispose( );
    }
    public void Dispose( )
    {
        Finalize( );
        GC.SuppressFinalize(this);
    }
    ...
    private Resource rare = new Resource( );
}
```

Calling the Disposal Method Multiple Times

Remember, the Disposal Method is public, so it can be called repeatedly. The easiest way to make sure multiple calls are possible is with a simple bool field. The following code provides an example:

```
class Example
{
    ...
    ~Example( )
    {
        disposed = true;
        rare.Dispose( );
    }
    public void Dispose( )
    {
        if (!disposed) {
            Finalize( );
            GC.SuppressFinalize(this);
        }
    }
    ...
    private Resource rare = new Resource( );
    private bool disposed = false;
}
```

Avoiding the Use of Released Resources

The easiest way to do avoid using released resources is to reset the reference to **null** in **Finalize** and check for **null** in each method, as follows:

```
class Example
{
    ...
    ~Example( )
    {
        rare.Dispose( );
        rare = null;
        disposed = true;
    }

    public void Dispose( )
    {
        if (!disposed) {
            Finalize( );
            GC.SuppressFinalize(this);
        }
    }

    public void Use( )
    {
        if (!disposed) {
            Wibble w = rare.Stuff( );
            ...
        } else {
            throw new DisposedException( );
        }
    }

    private Resource rare = new Resource( );
    private bool disposed = false;
}
```

Using IDisposable

- **If Any Method Throws an Exception**
  - Subsequent statements may not be executed
  - Resources may not be released by your code
- **Use try-finally Blocks Carefully**
- **A Proposed Modification to C# Handles This By:**
  - Extending the **using** statement to indicate resource use
  - Providing the **IDisposable** interface that declares a **Dispose** method, for use by resource classes

When writing disposal code, it is important to be aware of the some of the common programming errors. For example, there is a dispose method trap that is quite common. Look at the following code, and decide whether **reader.Close** (which is a Disposal Method that reclaims a scarce file handle) is called.

```
class SourceFile
{
    public SourceFile(string name)
    {
        File src = new File(name);
        contents = new char[(int)src.Length];
        StreamReader reader = src.OpenText( );
        reader.ReadBlock(contents, 0, contents.Length);
        reader.Close( );
    }
    ...
    private char[ ] contents;
}
```

The answer is that **reader.Close** is not guaranteed to be called. The problem is that if a statement before the call to **Close** throws an exception, the flow of control will bypass the call to **Close**. One way you can solve this problem is by using a **finally** block, as follows:

```
class SourceFile
{
    public SourceFile(string name)
    {
        StreamReader reader = null;
        try {
            File src = new File(name);
            contents = new char[(int)src.Length];
            reader = src.OpenText( );
            reader.ReadBlock(contents, 0, contents.Length);
        }
        finally {
            if (reader != null) {
                reader.Close( );
            }
        }
    }
    ...
    private char[ ] contents;
}
```

This solution works, but it is not completely satisfactory because:

- You must reorder the declaration of the resource reference.
- You must remember to initialize the reference to **null**.
- You must remember to ensure that the reference is not **null** in the **finally** block.
- It quickly becomes unwieldy if there is more than one resource to dispose of.

Proposed Modifications to C#

A proposed amendment to C# provides a solution that avoids all of these problems. It uses an extension of the **using** statement (part of the C# language) with the **IDisposable** interface (part of the C# .NET Framework SDK) to implement resource classes.

These new enhancements are not available in Beta 1 and therefore are not covered in this course.

Lab 9.2: Destroying Objects

Objectives

In this lab, you will learn how to use finalizers to perform processing before garbage collection destroys an object.

After completing this lab, you will be able to:

- Create a destructor.
- Make requests of garbage collection.
- Use the Disposal design pattern.

Prerequisites

Before working on this lab, you must be able to:

- Create classes and instantiate objects.
- Define and call methods.
- Define and use constructors.
- Use the **StreamWriter** class to write text to a file.

You should also have completed Lab 9.1. If you did not complete Lab 9.1, you can use the solution code provided.

Estimated time to complete this lab: 15 minutes

Exercise 1
Creating a Destructor

In this exercise, you will create a finalizer for the **BankTransaction** class. The finalizer will allow **BankTransaction** to persist its data to the end of the file Transactions.dat in the current directory. The data will be written out in human-readable form.

↙ To save transactions

1. Open the Finalizers.sln project in the *Lab Files*\Lab09\Starter\Finalizers folder.
2. Add the following **using** directive to the start of the BankTransaction.cs file:

```
using System.IO;
```

3. In the **BankTransaction** class, add a destructor, as follows:
   a. It should be called **~BankTransaction**.
   b. It should not have an access modifier.
   c. It should not take any parameters.
   d. It should not return a value.
4. In the body of the destructor, add statements to:
   a. Create a new **StreamWriter** variable that opens the Transactions.dat file in the current directory in append mode (that is, it writes data to the end of the file if it already exists.) You can achieve this by using the **File.AppendText** method. For information about this method, search for "File.AppendText" in the .NET Framework SDK Help documents.
   b. Write the contents of the transaction to this file. (Format so that it is readable.)
   c. Close the file:

```
~BankTransaction( )
{
  StreamWriter swFile =
➥File.AppendText("Transactions.Dat");
  swFile.WriteLine("Date/Time: {0}\tAmount {1}", when,
➥amount);
  swFile.Close( );
}
```

5. Compile your program and correct any errors.

↙ To test the destructor

1. Review the CreateAccount.cs test harness. The test harness:
   a. Creates an account (*acc1*).
   b. Deposits money to and withdraws money from *acc1*.
   c. Prints the contents of *acc1* and its transactions.

   When the **Main** method finishes, what will happen to the account *acc1* and the transactions?
2. Compile and execute the program. Verify that the information displayed is as expected. Is the Transactions.dat file created as expected? (It should be in the bin\debug folder in the project folder.) If not, why not?

Note You will find that the Transactions.dat file is not created because garbage collection never needs to collect garbage in such a small program, so the destructor is never executed. For a bank, this is not a good situation because the bank is probably not allowed to lose records of transactions. You will fix this problem in Exercise 2 by using the Disposal pattern.

Exercise 2
Using the Disposal Design Pattern

In this exercise, you will use the Disposal design pattern to ensure that a **BankTransaction**'s data is saved on demand rather than when garbage collection destroys the **BankTransaction**. You will also need to inform garbage collection that the **BankTransaction** has already been disposed of and suppress any attempt by garbage collection to destroy it again later.

You will add a **Dispose** method to the **BankAccount** and **BankTransaction** classes. The **Dispose** method in **BankAccount** will iterate through all of the transactions in its transaction queue, and call **Dispose** for each transaction.

↙ To make BankTransaction suitable for finalizing

1. In the **BankTransaction** class, add a public **void** method called **Dispose** that only calls **Finalize**:

```
public void Dispose( )
{
  Finalize( );
}
```

2. Add to the end of the destructor a call to `GC.SuppressFinalize(this)`.

   Calling **Finalize** will invoke the destructor. You need to ensure that garbage collection does not call the destructor again after you have used it.

↙ To create a Dispose method for the BankAccount class

1. In the **BankAccount** class, add a **using System** directive.
2. Add a private **bool** instance variable called *dead*. Initialize it to **false**.
3. Add a public method called **Dispose**. It should take no parameters and have a **void** return type.
4. In the **Dispose** method, add statements to:
   a. Examine the value of *dead*. If it is **true**, return from the method and do nothing else.
   b. If *dead* is **false**, iterate through all of the **BankTransaction** objects in *tranQueue* and call **Dispose** for each one. Use a **foreach** statement, as you did in Lab 9.1.
   c. Call `GC.SuppressFinalize(this)` to prevent garbage collection from destroying the account again.
   d. Set *dead* to **true**.

   The completed code should be as follows:

```
public void Dispose( )
{
   if (dead) return;
   foreach(BankTransaction tran in tranQueue)
   {
       tran.Dispose( );
   }
   GC.SuppressFinalize(this);
   dead = true;
}
```

5. Compile the project and correct any errors.

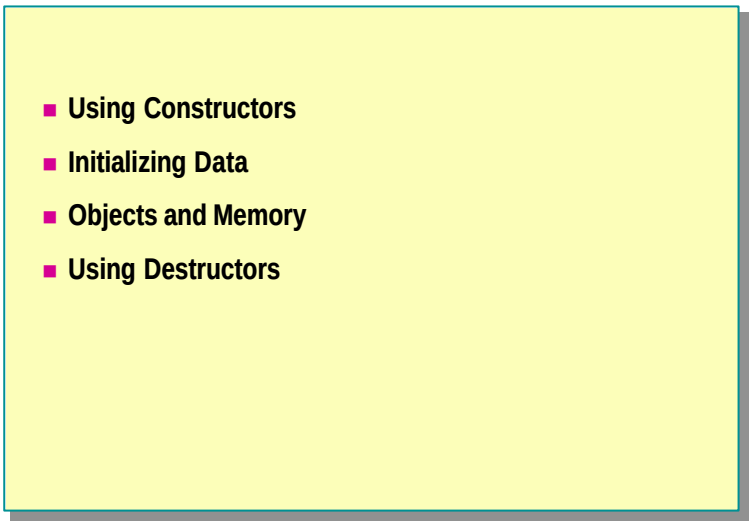
↙ To test the destructor

1. Open the CreateAccount.cs test harness.
2. Add a statement to the end of **Main** that calls the **Dispose** method of *acc1*, to ensure that it is saved correctly, as follows:

```
acc1.Dispose( );
```

3. Compile the project and correct any errors.
4. Run the program. The same output as before should be displayed on the screen. However, this time the Transactions.dat file should also be created.

Review



1. Declare a class called **Date** with a public constructor that expects three **int** parameters called *year*, *month*, and *day*.

2. Will the compiler generate a default constructor for the **Date** class that you declared in question 1? What if **Date** were a **struct** with the same three-**int** constructor?

3. Which method does garbage collection call on the object just before it recycles the object's memory back to the heap? Declare a class called **SourceFile** that contains this method.

4. What is wrong with the following code fragment?

```
class Example
{
    ~Example( ) { }
    protected void override Finalize( ) { }
}
```