Module 2

Chapter 7

Creating and managing classes and objects

Understanding classification

- *Class* is the root word of the term *classification*.
- When you design a class, you systematically arrange information and behavior into a meaningful entity.
- This arranging is an act of classification and is something that everyone does—not just programmers.

  For example, all cars share common behaviours (they can be steered, stopped, accelerated, and so on) and common attributes (they have a steering wheel, an engine, and so on). People use the word *car* to mean an object that shares these common behaviors and attributes. As long as everyone agrees on what a word means, this system works well and you can express complex but precise ideas in a concise form.
- Without classification, it's hard to imagine how people could think or communicate at all.

The purpose of encapsulation

- ***Encapsulation*** is an important principle when defining classes.
- The idea is that a program that uses a class should not have to worry how that class actually works internally; the program simply creates an instance of a class and calls the methods of that class.
- Encapsulation actually has two purposes:
  - To combine methods and data within a class; in other words, to support classification
  - To control the accessibility of the methods and data; in other words, to control the use of the class

Defining and using a class

In C#, you use the ***class*** keyword to define a new class. The data and methods of the class occur in the body of the class between a pair of braces. Following is a C# class called *Circle*

that contains one method (to calculate the circle's area) and one piece of data (the circle's radius):

```
class Circle
{
        int radius;
        double Area()
        {
                return Math.PI * radius * radius;
        }
}
```

The body of a class contains ordinary methods (such as *Area*) and fields (such as *radius*).

You can use the *Circle* class in a similar manner to using the other types that you have already met.
You create a variable specifying *Circle* as its type, and then you initialize the variable with some valid data. Here is an example:

```
Circle c; // Create a Circle variable
c = new Circle(); // Initialize it
```

new keyword creates a new instance of a class, more commonly called an *object*.

C# just doesn't provide the syntax for assigning literal class values to variables. You cannot write a statement such as this:

```
Circle c;
c = 42;
```

Memory for variables of class types is allocated and managed by the runtime

You can, however, directly assign an instance of a class to another variable of the same type, like this:

```
Circle c;
```

```
c = new Circle();
Circle d;
d = c;
```

Controlling accessibility

- A method or field is **private** if it is accessible only from within the class. To declare that a method or field is private, you write the keyword *private* before its declaration. As intimated previously, this is actually the default, but it is good practice to state explicitly that fields and methods are private to avoid any confusion.
- A method or field is **public** if it is accessible from both within and outside of the class. To declare that a method or field is public, you write the keyword *public* before its declaration.

Here is the *Circle* class again. This time, *Area* is declared as a public method and *radius* is declared as a private field:

```
class Circle
{
    private int radius;
    public double Area()
    {
        return Math.PI * radius * radius;
    }
}
```

Note If you are a C++ programmer, be aware that there is no colon after the *public* and *private* keywords. You must repeat the keyword for every field and method declaration.

Although *radius* is declared as a private field and is not accessible from outside the class, *radius* is accessible from within the *Circle* class. The *Area* method is inside the *Circle* class, so the body of *Area* has access to *radius*. However, the class is still of limited value because there is no way of initializing the *radius* field. To fix this, you can use a constructor.

Tip Unlike variables declared in a method that are not initialized by default, the fields in a class are automatically initialized to *0*, *false*, or *null*, depending on their type. However, it is still good practice to provide an explicit means of initializing fields.

Naming and accessibility

The following recommendations are reasonably common and relate to the naming conventions for fields and methods based on the accessibility of class members; however, C# does not enforce these rules:

- Identifiers that are *public* should start with a capital letter. For example, *Area* starts with *A* (not *a*) because it's *public*. This system is known as the *PascalCase* naming scheme (because it was first used in the Pascal language).
- Identifiers that are not *public* (which include local variables) should start with a lowercase letter. For example, *radius* starts with *r* (not *R*) because it's *private*. This system is known as the *camelCase* naming scheme.

There's only one exception to this rule: class names should start with a capital letter, and constructors must match the name of their class exactly; therefore, a *private* constructor must start with a capital letter.

Important Don't declare two *public* class members whose names differ only in case. If you do, developers using other languages that are not case sensitive (such as Microsoft Visual Basic) might not be able to integrate your class into their solutions.

Working with constructors

- When you use the *new* keyword to create an object, the runtime needs to construct that object by using the definition of the class.
- The runtime must grab a piece of memory from the operating system, fill it with the fields defined by the class, and then invoke a constructor to perform any initialization required.

- A ***constructor*** is a special method that runs automatically when you create an instance of a class.

- It has the same name as the class, and it can take parameters, but it cannot return a value (not even *void*).

- Every class must have a constructor. If you don't write one, the compiler automatically generates a default constructor for you. (However, the compiler-generated default constructor doesn't actually do anything.)

- You can write your own default constructor quite easily. Just add a public method that does not return a value and give it the same name as the class.

The following example shows the *Circle* class with a default constructor that initializes the *radius* field to 0:

```
class Circle
{
        private int radius;
        public Circle() // default constructor
        {
                radius = 0;
        }
        public double Area()
        {
                return Math.PI * radius * radius;
        }
}
```

Note In C# parlance, the *default* constructor is a constructor that does not take any parameters. It does not matter whether the compiler generates it or you write it, it is still the default constructor. You can also write nondefault constructors (constructors that *do* take parameters), as you will see in the upcoming section "Overloading constructors."

In this example, the constructor is marked as *public*. If this keyword is omitted, the constructor will be private (just like any other methods and fields). If the constructor is private, it cannot be used outside the class, which prevents you from being able to create *Circle* objects from methods that are not part of the *Circle* class. You might therefore think that private constructors are not that valuable. They do have their uses, but they are beyond the scope of the current discussion. Having added a public constructor, you can now use the *Circle* class and exercise its *Area* method. Notice how you use dot notation to invoke the *Area* method on a *Circle* object:

```
Circle c;
c = new Circle();
double areaOfCircle = c.Area();
```

Overloading constructors

- Constructor Overloading is a technique to create multiple constructors with a different set of parameters and the different number of parameters.
- It allows us to use a class in a different manner.
- The same class may behave different type based on constructors overloading.

Add another constructor to the *Circle* class, with a parameter that specifies the radius to use, like this:

```
class Circle
{
    private int radius;
    public Circle() // default constructor
    {
        radius = 0;
    }
    public Circle(int initialRadius) // overloaded constructor
    {
        radius = initialRadius;
    }
    public double Area()
    {
        return Math.PI * radius * radius;
    }
}
```

Note: The order of the constructors in a class is immaterial; you can define constructors in whatever order with which you feel most comfortable.

You can then use this constructor when creating a new Circle object, such as in the following:

```
Circle c;
c = new Circle(45);
```

When you build the application, the compiler works out which constructor it should call based on the parameters that you specify to the new operator. In this example, you passed an *int*, so the compiler generates code that invokes the constructor that takes an *int* parameter.

- **You should be aware of an important feature of the C# language: If you write your own constructor for a class, the compiler does not generate a default constructor.**
- **Therefore, if you've written your own constructor that accepts one or more parameters and you also want a default constructor, you'll have to write the default constructor yourself.**

Understanding static methods and data

C# includes static keyword just like other programming languages such as C++, Java, etc. The static keyword can be applied on classes, variables, methods, properties, operators, events and constructors. However, it cannot be used with indexers, destructors or types other than classes.

The static modifier makes an item non-instantiable, it means the static item cannot be instantiated. If the static modifier is applied to a class then that class cannot be instantiated using the new keyword. If the static modifier is applied to a variable, method or property of class then they can be accessed without creating an object of the class, just use className.propertyName, className.methodName.

In C#, all methods must be declared within a class. However, if you declare a method or a field as static, you can call the method or access the field by using the name of the class. No instance is required. This is how the Sqrt method of the Math class is declared:

```
class Math
{
    public static double Sqrt(double d)
    {
        ...
    }
```

```
...
}
```

A static method does not depend on an instance of the class, and it cannot access any instance fields or instance methods defined in the class; it can use only fields and other methods that are marked as static.

Creating a shared field

Defining a field as static makes it possible for you to create a single instance of a field that is shared among all objects created from a single class. (Nonstatic fields are local to each instance of an object.)

In the following example, the static field *NumCircles* in the Circle class is incremented by the *Circle* constructor every time a new *Circle* object is created:

```
class Circle
{
private int radius;
public static int NumCircles = 0;
public Circle() // default constructor
{
        radius = 0;
        NumCircles++;
}
public Circle(int initialRadius) // overloaded constructor
{
        radius = initialRadius;
        NumCircles++;
}
}
```

All Circle objects share the same instance of the *NumCircles* field, so the statement *NumCircles*++; increments the same data every time a new instance is created. Notice that you cannot prefix *NumCircles* with the *this* keyword, as *NumCircles* does not belong to a specific object. You can access the *NumCircles* field from outside of the class by specifying the *Circle* class rather than a *Circle* object, such as in the following example:

Console.WriteLine("Number of Circle objects: {0}", Circle.NumCircles);

Creating a *static* field by using the *const* keyword

- By prefixing the field with the *const* keyword, you can declare that a field is static but that its value can never change.
- The keyword *const* is short for constant.
- A *const* field does not use the static keyword in its declaration but is nevertheless static.
- You can declare a field as *const* only when the field is a numeric type (such as *int* or *double*), a *string*, or an enumeration For example, here's how the Math class declares PI as a *const* field:

```
class Math
{
        ...
        public const double PI = 3.14159265358979323846;
}
```

Understanding static classes

- Another feature of the C# language is the ability to declare a class as static.
- A static class can contain only static members. (All objects that you create by using the class share a single copy of these members.)
- The purpose of a static class is purely to act as a holder of utility methods and fields.
- A static class cannot contain any instance data or methods. If you need to perform any initialization, a static class can have a default constructor as long as it is also declared as static. Any other types of constructor are illegal and will be reported as such by the compiler.
- If you were defining your own version of the Math class, one containing only static members, it could look like this:

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

```
    public static double Sqrt(double x) {...}
    ...
}
```

Anonymous classes

- An anonymous class is a class that does not have a name.
- You create an anonymous class simply by using the new keyword and a pair of braces defining the fields and values that you want the class to contain, like this:

```
myAnonymousObject = new { Name = "John", Age = 47 };
```

This class contains two public fields called Name (initialized to the string "John") and Age (initialized to the integer 47). The compiler infers the types of the fields from the types of the data you specify to initialize them.

- When you define an anonymous class, the compiler generates its own name for the class, but it won't tell you what it is.
- However, this is not a problem if you declare *myAnonymousObject* as an implicitly typed variable by using the var keyword, like this:

```
var myAnonymousObject = new { Name = "John", Age = 47 };
```

Remember that the *var* keyword causes the compiler to create a variable of the same type as the expression used to initialize it. In this case, the type of the expression is whatever name the compiler happens to generate for the anonymous class.

You can access the fields in the object by using the familiar dot notation, as is demonstrated here:

```
Console.WriteLine("Name:{0} Age: {1}", myAnonymousObject.Name,
myAnonymousObject.Age};
```

You can even create other instances of the same anonymous class but with different values, such as in the following:

```
var anotherAnonymousObject = new { Name = "Diana", Age = 46 };
```

The C# compiler uses the names, types, number, and order of the fields to determine whether two instances of an anonymous class have the same type. In this case, the variables

myAnonymousObject and *anotherAnonymousObject* have the same number of fields, with the same name and type, in the same order, so both variables are instances of the same anonymous class. This means that you can perform assignment statements such as this:

anotherAnonymousObject = myAnonymousObject;

Chapter 8

Understanding values and references

Copying value type variables and classes

- Most of the primitive types built into C#, such as *int, float, double,* and *char* (but not string) are collectively called value types.
- These types have a fixed size, and when you declare a variable as a value type, the compiler generates code that allocates a block of memory big enough to hold a corresponding value.
- For example, declaring an *int* variable causes the compiler to allocate 4 bytes of memory (32 bits). A statement that assigns a value (such as 42) to the *int* causes the value to be copied into this block of memory.
- Class types such as *Circle* are handled differently.
- When you declare a *Circle* variable, the compiler does not generate code that allocates a block of memory big enough to hold a *Circle*; all it does is allot a small piece of memory that can potentially hold the address of (or a reference to) another block of memory containing a *Circle*. (An address specifies the location of an item in memory.)
- The memory for the actual *Circle* object is allocated only when the *new* keyword is used to create the object.
- **A class is an example of a reference type**. **Reference types hold references to blocks of memory.**

Consider the situation in which you declare a variable named *i* as an *int* and assign it the value 42. If you declare another variable called *copyi* as an *int* and then assign i to *copyi*, *copyi* will hold the same value as *i* (42). However, even though *copyi* and *i* happen to hold the same value, there are two blocks of memory containing the value 42: one block for *i* and the other block for *copyi*. If you modify the value of *i*, the value of *copyi* does not change. Let's see this in code:

int i = 42; // declare and initialize i

int copyi = i; */* copyi contains a copy of the data in i: i and copyi both contain the value 42 */*

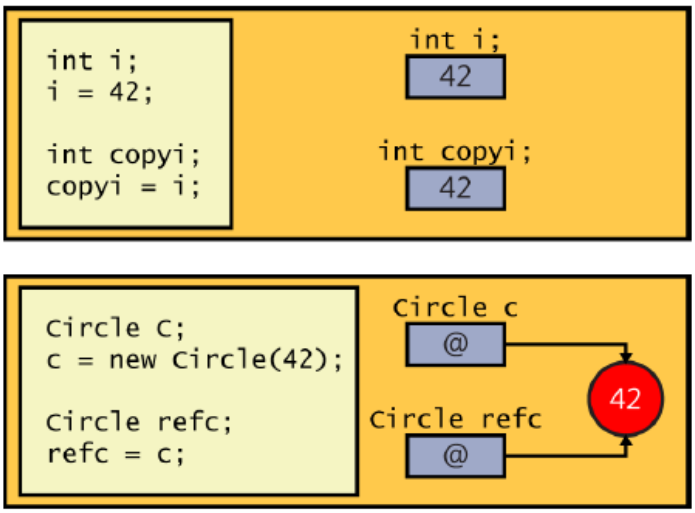
i++; */* incrementing i has no effect on copyi; i now contains 43, but copyi still contains 42 */*

The effect of declaring a variable c as a class type, such as *Circle*, is very different. When you declare c as a *Circle*, *c* can refer to a *Circle* object; the actual value held by *c* is the address of a *Circle* object in memory. If you declare an additional variable named *refc* (also as a Circle) and you assign c to *refc*, *refc* will have a copy of the same address as *c*; in other words, there is only one *Circle* object, and both *refc* and *c* now refer to it. Here's the example in code:

Circle c = new Circle(42);

Circle refc = c;

The following graphic illustrates both examples. The at sign (@) in the Circle objects represents a reference holding an address in memory:



Understanding null values and nullable types

To initialize a reference variable such as a class, you can create a new instance of the class and assign the reference variable to the new object, like this:

Circle c = new Circle(42);

This is all very well, but what if you don't actually want to create a new object? Perhaps the purpose of the variable is simply to store a reference to an existing object at some later point

in your program. In the following code example, the Circle variable copy is initialized, but later it is assigned a reference to another instance of the Circle class:

```
Circle c = new Circle(42);
Circle copy = new Circle(99); // Some random value, for initializing copy
...
copy = c; // copy and c refer to the same object
```

After assigning c to copy, what happens to the original Circle object with a radius of 99 that you used to initialize copy? Nothing refers to it anymore. **In this situation, the runtime can reclaim the memory by performing an operation known as garbage collection. The important thing to understand for now is that garbage collection is a potentially time-consuming operation; you should not create objects that are never used, because doing so is a waste of time and resources.**

You could argue that if a variable is going to be assigned a reference to another object at some point in a program, there is no point to initializing it. But, this is poor programming practice which can lead to problems in your code. For example, you will inevitably find yourself in the situation in which you want to refer a variable to an object only if that variable does not already contain a reference, as shown in the following code example:

```
Circle c = new Circle(42);
Circle copy; // Uninitialized !!!
...
if (copy == // only assign to copy if it is uninitialized, but what goes here?)
{
        copy = c; // copy and c refer to the same object
        ...
}
```

The purpose of the *if* statement is to test the copy variable to see whether it is initialized, but to which value should you compare this variable? The answer is to use a special value called null. In C#, you can assign the null value to any reference variable. The null value simply means that the variable does not refer to an object in memory. You can use it like this:

```
Circle c = new Circle(42);
```

```
Circle copy = null; // Initialized
...
if (copy == null)
{
        copy = c; // copy and c refer to the same object
        ...
}
```

Using nullable types

The null value is useful for initializing reference types. Sometimes, you need an equivalent value for value types, but null is itself a reference, and so you cannot assign it to a value type. The following statement is therefore illegal in C#:

```
int i = null; // illegal
```

However, C# defines a modifier that you can use to declare that a variable is a nullable value type.

A nullable value type behaves in a similar manner to the original value type, but you can assign the null value to it. **You use the question mark (?) to indicate that a value type is nullable,** like this:

```
int? i = null; // legal
```

You can ascertain whether a nullable variable contains null by testing it in the same way as a reference type.

```
if (i == null)
...
```

You can assign an expression of the appropriate value type directly to a nullable variable. The following examples are all legal:

```
int? i = null;
int j = 99;
i = 100; // Copy a value type constant to a nullable type
```

```
i = j; // Copy a value type variable to a nullable type
```

You should note that the converse is not true. You cannot assign a nullable variable to an ordinary value type variable. So, given the definitions of variables *i* and *j* from the preceding example, the following statement is not allowed:

```
j = i; // Illegal
```

Understanding the properties of nullable types

A nullable type exposes a pair of properties that you can use to determine whether the type actually has a non-null value, and what this value is.

- The ***HasValue*** property indicates whether a nullable type contains a value or is null.
- You can retrieve the value of a non-null nullable type by reading the **Value** property, like this:

```
int? i = null;
...
if (!i.HasValue)
{
        // If i is null, then assign it the value 99
        i = 99;
}
else
{
        // If i is not null, then display its value
        Console.WriteLine(i.Value);
}
```

This code fragment tests the nullable variable *i*, and if it does not have a value (it is null), it assigns it the value 99; otherwise, it displays the value of the variable.

In this example, using the **HasValue** property does not provide any benefit over testing for a null value directly. Additionally, reading the **Value** property is a long-winded way of reading the contents of the variable.

Note The *Value* property of a nullable type is read-only. You can use this property to read the value of a variable but not to modify it. To update a nullable variable, use an ordinary assignment statement.

Using ref and out parameters

The keywords ***ref*** and ***out*** are used to pass arguments within a method or function. Both indicate that an argument / parameter is passed by reference. By default parameters are passed to a method by value. By using these keywords (ref and out) we can pass a parameter by reference.

Ref Keyword

The *ref* keyword passes arguments by reference. It means any changes made to this argument in the method will be reflected in that variable when control returns to the calling method. When you pass an argument as a *ref* parameter, you must also prefix the argument with the *ref* keyword.

```
static void doIncrement(ref int param) // using ref
{
        param++;
}
static void Main()
{
        int arg = 42;
        doIncrement(ref arg); // using ref
        Console.WriteLine(arg); // writes 43
}
```

The *doIncrement* method receives a reference to the original argument rather than a copy, so any changes the method makes by using this reference actually change the original value. That's why the value 43 is displayed on the console.

Out Keyword

The ***out*** keyword is syntactically similar to the ***ref*** keyword. You can prefix a parameter with the out keyword so that the parameter becomes an alias for the argument. As when using ***ref***, anything you do to the parameter, you also do to the original argument.

When you pass an argument to an ***out*** parameter, you must also prefix the argument with the ***out*** keyword.

The keyword ***out*** is short for output. When you pass an ***out*** parameter to a method, the method must assign a value to it before it finishes or returns, as shown in the following example:

Because an *out* parameter must be assigned a value by the method, you're allowed to call the method without initializing its argument. For example, the following code calls *doInitialize* to initialize the variable *arg*, which is then displayed on the console:

```
static void doInitialize(out int param)
{
        param = 42;
}
static void Main()
{
        int arg; // not initialized
        doInitialize(out arg); // legal
        Console.WriteLine(arg); // writes 42
}
```

How computer memory is organized

Operating systems and language runtimes such as that used by C# frequently divide the memory used for holding data in two separate areas, each of which is managed in a distinct manner. These two areas of memory are traditionally called the stack and the heap. The stack and the heap serve different purposes, which are described here:

- Stack memory is organized like a stack of boxes piled on top of one another. When a method is called, each parameter is put in a box that is placed on top of the stack. Each local variable is likewise assigned a box, and these are placed on top of the boxes already on the stack. When a method finishes, you can think of the boxes being removed from the stack.

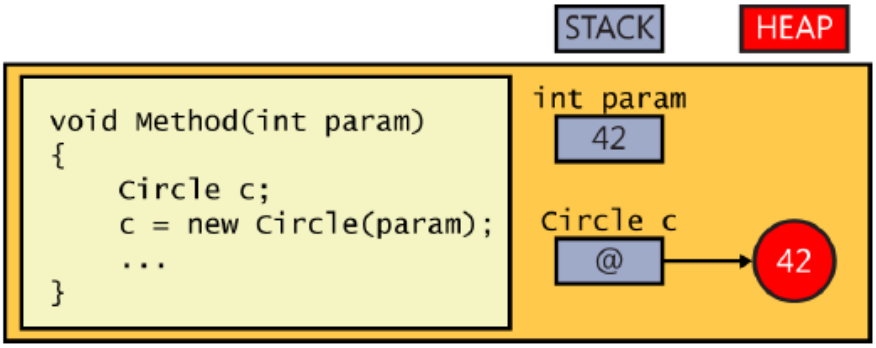
- Heap memory is like a large pile of boxes strewn around a room rather than stacked neatly on top of each other. Each box has a label indicating whether it is in use. When a new object is created, the runtime searches for an empty box and allocate it to the object. The reference to the object is stored in a local variable on the stack. The runtime keeps track of the number of references to each box. (Remember that two variables can refer to the same object.) When the last reference disappears, the runtime marks the box as not in use, and at some point in the future it will empty the box and make it available for reuse.

Using the stack and the heap

Now, let's examine what happens when the following method Method is called:

```
void Method(int param)
{
        Circle c;
        c = new Circle(param);
        ...
}
```

Suppose the argument passed into *param* is the value 42. When the method is called, a block of memory (just enough for an *int*) is allocated from the stack and initialized with the value 42. As execution moves inside the method, another block of memory big enough to hold a reference (a memory address) is also allocated from the stack but left uninitialized. (This is for the *Circle* variable, c.) Next, another piece of memory big enough for a *Circle* object is allocated from the heap. This is what the new keyword does. The *Circle* constructor runs to convert this raw heap memory to a *Circle* object. A reference to this *Circle* object is stored in the variable c. The following graphic illustrates the situation:



- Although the object is stored on the heap, the reference to the object (the variable c) is stored on the stack.
- Heap memory is not infinite. If heap memory is exhausted, the new operator will throw an ***OutOfMemoryException*** exception and the object will not be created.

When the method ends, the parameters and local variables go out of scope. The memory acquired for *c* and for *param* is automatically released back to the stack. The runtime notes that the *Circle* object is no longer referenced and at some point in the future will arrange for its memory to be reclaimed by the heap.

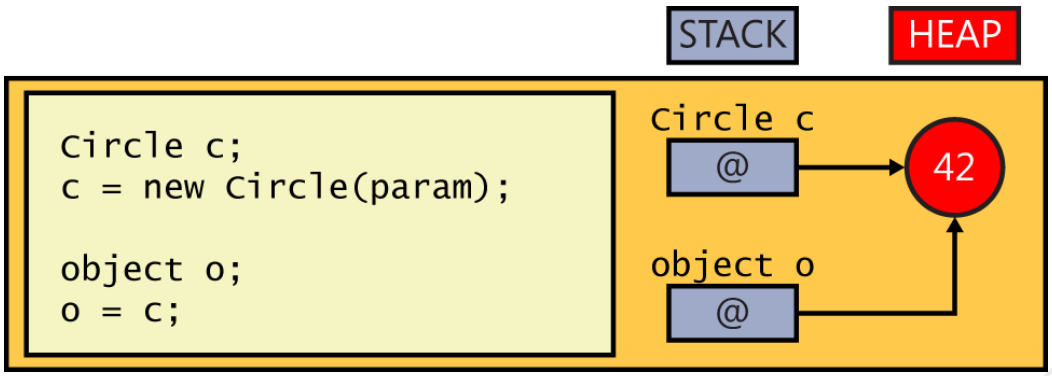
The System.Object class

All classes are specialized types of *System.Object* and that you can use *System.Object* to create a variable that can refer to any reference type. *System.Object* is such an important class that C# provides the *object* keyword as an alias for *System.Object*. In your code, you can use *object* or you can write *System.Object*—they mean exactly the same thing.

In the following example, the variables *c* and *o* both refer to the same *Circle* object. The fact that the type of *c* is Circle and the type of *o* is object (the alias for *System.Object*) in effect provides two different views of the same item in memory.

```
Circle c;
c = new Circle(42);
object o;
o = c;
```

The following diagram illustrates how the variables *c* and *o* refer to the same item on the heap.



Boxing

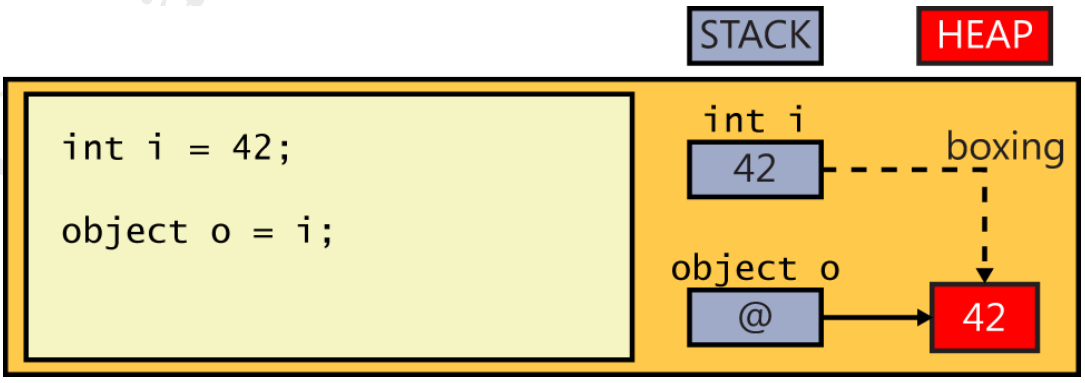
The operation of Converting a Value Type to a Reference Type is called Boxing (automatic copying of an item from the stack to the heap).

For example, the following two statements initialize the variable *i* (of type *int*, a value type) to 42 and then initialize the variable *o* (of type object, a reference type) to *i*:

int i = 42;

object o = i;

The second statement requires a little explanation to appreciate what is actually happening. Remember that *i* is a value type and that it lives on the stack. If the reference inside *o* referred directly to *i*, the reference would refer to the stack. However, all references must refer to objects on the heap; creating references to items on the stack could seriously compromise the robustness of the runtime and create a potential security flaw, so it is not allowed. Therefore, the runtime allocates a piece of memory from the heap, copies the value of integer *i* to this piece of memory, and then refers the object *o* to this copy.



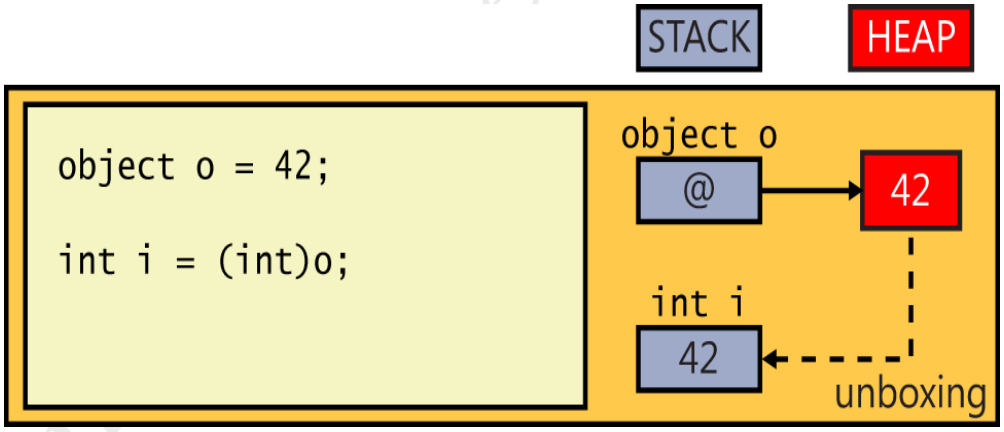
Unboxing

Converting a value of a reference type into a value of a value type is called UnBoxing.

To obtain the value of the boxed copy, you must use what is known as a ***cast***. This is an operation that checks whether it is safe to convert an item of one type to another before it actually makes the copy. You prefix the object variable with the name of the type in parentheses, as in this example:

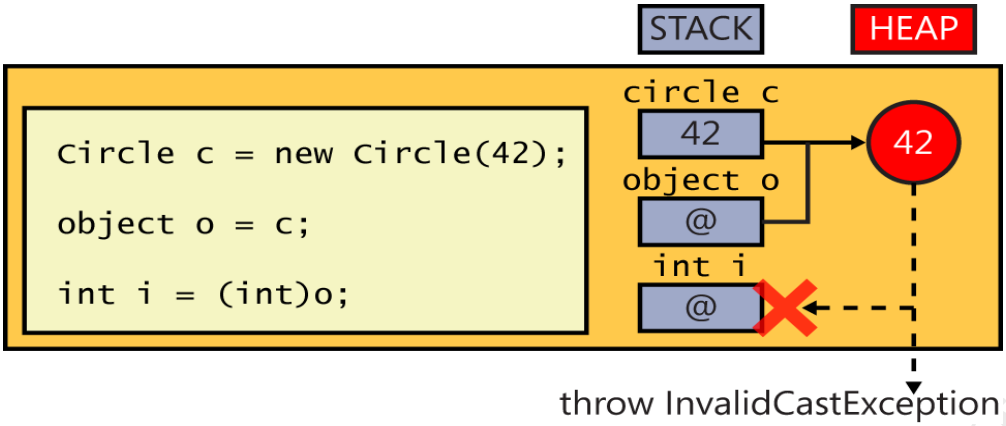
```
int i = 42;
object o = i; // boxes
i = (int)o; // unboxing
```

The compiler notices that you've specified the type *int* in the cast. Next, the compiler generates code to check what *o* actually refers to at run time. It could be absolutely anything. Just because your cast says *o* refers to an *int*, that doesn't mean it actually does. If *o* really does refer to a boxed *int* and everything matches, the cast succeeds and the compiler-generated code extracts the value from the boxed *int* and copies it to *i*.



On the other hand, if *o* does not refer to a boxed *int*, there is a type mismatch, causing the cast to fail. The compiler-generated code throws an *InvalidCastException* exception at run time. Here's an example of an unboxing cast that fails:

```
Circle c = new Circle(42);
object o = c; // doesn't box because Circle is a reference variable
int i = (int)o; // compiles okay but throws an exception at run time
```



Casting data safely (Using *is* and *as* operator)

C# provides two more very useful operators that can help you perform casting in a much more elegant manner: the *is* and *as* operators.

The *is* operator

You can use the *is* operator to verify that the type of an object is what you expect it to be, like this:

```
Circle c = new Circle();
...
object o = c;
if (o is Circle)
{
    Circle temp = (Circle)o; // This is safe; o is a Circle
    ...
}
```

The *is* operator takes two operands: a reference to an object on the left, and the name of a type on the right. If the type of the object referenced on the heap has the specified type, *is* evaluates to *true*; otherwise, *is* evaluates to *false*. The preceding code attempts to cast the reference to the *object* variable *o* only if it knows that the cast will succeed.

The *as* operator

The *as* operator fulfills a similar role to *is* but in a slightly truncated manner. You use the *as* operator like this:

```
Circle c = new Circle();
```

```
...
object o = c;
Circle temp = o as Circle;
if (temp != null)
{
        ... // Cast was successful
}
```

Like the *is* operator, the *as* operator takes an object and a type as its operands. The runtime attempts to cast the object to the specified type. If the cast is successful, the result is returned and, in this example, it is assigned to the *Circle* variable *temp*. If the cast is unsuccessful, the *as* operator evaluates to the *null* value and assigns that to *temp* instead.

Chapter 9
Creating Value Types with Enumerations and Structures

Things to Learn

- ✓ Declare an enumeration type.
- ✓ Create and use an enumeration type.
- ✓ Declare a structure type.
- ✓ Create and use a structure type.
- ✓ Explain the differences in behavior between a structure and a class.

The two fundamental types that exist in Microsoft Visual C#: ***value*** types and ***reference*** types. A ***value*** type variable holds its **value** directly on the **stack**, whereas a ***reference*** type variable holds a **reference** to an **object** on the **heap**. C# supports two kinds of ***value*** types: ***enumerations*** and ***structures***.

Working with enumerations

Suppose user wants to represent the seasons of the year in a program, he could use the integers **0**, **1**, **2**, and **3** to represent spring, summer, fall, and winter, respectively. This system would work, but it's not very intuitive. If integer value **0** in code is used, it wouldn't be obvious that a particular **0** represented spring. It also wouldn't be a very robust solution. Eg: if an ***int*** variable named ***season*** is declared, there is nothing to stop the user from assigning it any legal integer value outside of the set **0**, **1**, **2**, or **3**. C# offers a better solution - create an enumeration (sometimes called an ***Enum*** type), whose values are limited to a set of symbolic names.

Declaring an enumeration

An enumeration can be defined by using the ***enum*** keyword, followed by a set of symbols identifying the legal values that the type can have, enclosed between braces. Eg: declaring an enumeration named ***Season*** whose literal values are limited to the symbolic names ***Spring***, ***Summer***, ***Fall***, and ***Winter***:

```
enum Season { Spring, Summer, Fall, Winter }
```

Using an enumeration

After an enumeration is declared, it can be used in exactly the same way as any other type. If the name of the enumeration is ***Season***, variables, fields, and parameters of type ***Season***, can be created as shown in this example:

```
enum Season { Spring, Summer, Fall, Winter }
class Example
{
        public void Method(Season parameter) // method parameter example
        {
        Season localVariable; // local variable example
        ...
        }
        private Season currentSeason; // field example
}
```

Before reading the value of an enumeration variable, it must be assigned a value. Only value that is defined by the enumeration can be assigned to an enumeration variable:

```
Season colorful = Season.Fall;
Console.WriteLine(colorful); // writes out 'Fall'
```

Note: A **nullable** version of an enumeration variable can be created by using the '**?**' modifier, and can then assign the **null** value, as well the values defined by the enumeration, to the variable:

```
Season? colorful = null;
```

All enumeration literal names are scoped by their enumeration type. This is useful because it makes it possible for different enumerations to coincidentally contain literals with the same name. Eg: write ***Season.Fall*** rather than just ***Fall***.

Also, notice that when displaying an enumeration variable using ***Console.WriteLine***, the compiler generates code that writes out the name of the literal whose value matches the value of the variable. If needed, can also explicitly convert an enumeration variable to a string that represents its current value by using the built-in **ToString** method that all enumerations automatically contain, as demonstrated in the following example:

string name = colorful.ToString();
Console.WriteLine(name); *// also writes out 'Fall'*

Many standard operators that are used on integer variables can also be used on enumeration variables (except the **bitwise** and **shift** operators).Eg: comparing two enumeration variables of the same type for equality by using the equality operator (==), and even performing arithmetic on an enumeration variable.

Choosing enumeration literal values

Internally, an enumeration type associates an integer value with each element of the enumeration. By default, the numbering starts at 0 for the first element and goes up in steps of 1. It's possible to retrieve the underlying integer value of an enumeration variable. To do this, must cast enumeration variable to its underlying type. Unboxing instructs that casting a type converts the data from one type to another, as long as the conversion is valid and meaningful. The following code example writes out the value **2** and not the word ***Fall*** (remember, in the ***Season*** enumeration ***Spring*** is **0**, ***Summer*** **1**, ***Fall*** **2**, and ***Winter*** **3**):

enum Season { Spring, Summer, Fall, Winter }
...
Season colorful = Season.Fall;
Console.WriteLine((int)colorful); *// writes out '2'*
Can also associate a specific integer constant (such as 1) with an enumeration literal (such as Spring):
enum Season { Spring = 1, Summer, Fall, Winter }

If an enumeration literal is not explicitly given a constant integer value, the compiler gives it a value that is one greater than the value of the previous enumeration literal, except for the very first enumeration literal, to which the compiler gives the default value ***0***. In the preceding example, the underlying values of ***Spring***, ***Summer***, ***Fall***, and ***Winter*** are now ***1***, ***2***, ***3***, and ***4***.
More than one enumeration literal can be given the same underlying value. For example, in the United Kingdom, ***Fall*** is referred to as ***Autumn***, both cultures can be catered as follows:

enum Season { Spring, Summer, Fall, Autumn = Fall, Winter }

Choosing an enumeration's underlying type

When declaring an enumeration, the enumeration literals are given values of type ***int***. We can also choose to base enumeration on a different underlying integer type. For example, declare ***Season's*** underlying type to ***short*** rather than an ***int***:

enum Season : short { Spring, Summer, Fall, Winter }

The main reason for doing this is to save memory; an ***int*** occupies more memory than a short, and if user does not need the entire range of values available to an ***int***, using a smaller data type can make sense.

User can base an enumeration on any of the eight integer types: ***byte***, ***sbyte***, ***short***, ***ushort***, ***int***, ***uint***, ***long***, or ***ulong***. The values of all the enumeration literals must fit within the range of the chosen base type. For example, if you base an enumeration on the byte data type, you can have a maximum of ***256*** literals (starting at ***0***).

Working with structures

Classes define reference types that are always created on the heap. In some cases, the class can contain so little data that the overhead of managing the heap becomes disproportionate. In these cases, it is better to define the type as a structure**. A structure is a value type**. Because **structures are stored on the stack**, as long as the structure is reasonably small, the memory management overhead is often reduced.

Like a class, a **structure can have its own fields, methods, and constructors**.

Common structure types

In C#, the primitive numeric types **int**, **long**, and **float** are aliases for the structures **System.Int32**, **System.Int64**, and **System.Single**, respectively. These structures have **fields** and **methods**, and we can actually call methods on variables and literals of these types. For example, all of these structures provide a **ToString** method that can convert a numeric value to its string representation. The following statements are all legal in C#:

```
int i = 55;
Console.WriteLine(i.ToString());
Console.WriteLine(55.ToString());
```

```
float f = 98.765F;
Console.WriteLine(f.ToString());
Console.WriteLine(98.765F.ToString());
```

We don't see this use of the **ToString** method often, because the **Console.WriteLine** method calls it automatically when it is needed. It is more common to use some of the static methods exposed by these structures. Eg: the static **int.Parse** method is used to convert a string to its corresponding integer value. What is actually being invoked is the **Parse** method of the **Int32** structure:

```
string s = "42";
int i = int.Parse(s); // exactly the same as Int32.Parse
```

These structures also include some useful static fields. Eg: **Int32.MaxValue** is the maximum value that an **int** can hold, and **Int32.MinValue** is the smallest value that you can store in an **int**.
The following table shows the primitive types in C# and their equivalent types in the **Microsoft .NET Framework**. Notice that the string and object types are classes (reference types) rather than structures.

Declaring a structure

To declare structure type, use the **struct** keyword followed by the name of the type, followed by the body of the structure, between opening and closing braces. Syntactically, the process is similar to declaring a class. For example, here is a structure named **Time** that contains three **public int** fields named **hours**, **minutes**, and **seconds**:

```
struct Time
{
    public int hours, minutes, seconds;
}
```

As with classes, making the fields of a structure **public** is not advisable in most cases; there is no way to control the values held in **public** fields. For example, anyone could set the value of **minutes** or **seconds** to a value greater than **60**. A better idea is to make the fields **private** and

provide your structure with constructors and methods to initialize and manipulate these fields, as shown in this example:

```
struct Time
{
        private int hours, minutes, seconds;
        ...
        public Time(int hh, int mm, int ss)
        {
                this.hours = hh % 24;
                this.minutes = mm % 60;
                this.seconds = ss % 60;
        }
        public int Hours()
        {
                return this.hours;
        }
}
```

Note :

By default, we cannot use many of the common operators on our own structure types. For example, we cannot use operators such as the **equality** operator (==) and the **inequality** operator (**!=**) on our own structure type variables. However, we can use the built-in **Equals()** method exposed by all structures to compare them, and we can also explicitly declare and implement operators for our own structure types.

When we copy a value type variable, we get two copies of the value. In contrast, when we copy a reference type variable, we get two references to the same object. In summary, use structures for small data values for which it's just as or nearly as efficient to copy the value as it would be to copy an address. Use classes for more complex data that is too big to copy efficiently.

Understanding structure and class differences

A structure and a class are syntactically similar, but there are a few important differences. Let's look at some of these variances:

- We can't declare a default constructor (a constructor with no parameters) for a structure. The following example would compile if Time were a class, but because Time is a structure, it does not:

```
struct Time
{
public Time() { ... } // compile-time error
...
}
```

The reason we can't declare our own default constructor for a structure is that the compiler always generates one. In a class, the compiler generates the default constructor only if we don't write a constructor ourself. The compiler-generated default constructor for a structure always sets the fields to **0**, false, or null—just as for a class. Therefore, we should ensure that a structure value created by the **default** constructor behaves logically and makes sense with these default values.

We can initialize fields to different values by providing a non-default constructor. However, when we do this, our non-default constructor must explicitly initialize all fields in our structure; the default initialization no longer occurs. If we fail to do this, we'll get a compile-time error. For example, although the following example would compile and silently initialize seconds to **0** if **Time** were a class, because **Time** is a structure, it fails to compile:

```
struct Time
{
    private int hours, minutes, seconds;
    ...
    public Time(int hh, int mm)
    {
        this.hours = hh;
        this.minutes = mm;
    } // compile-time error: seconds not initialized
}
```

- In a structure, we cannot initialize instance **fields** at their point of declaration. The following example would compile if **Time** were a class, but because **Time** is a structure, it causes a compile-time error:

```
struct Time
{
        private int hours = 0; // compile-time error
        private int minutes;
        private int seconds;
        ...
}
```

The following table summarizes the main differences between a structure and a class.

Structure	Class
A structure is a value type.	A class is a reference type.
Structure instances are called values and live on the stack.	Class instances are called objects and live on the heap.
We cannot declare a default constructor	We can declare a default constructor
If you declare your own constructor, the compiler still generate the default constructor	If you declare your own constructor, the compiler will not generate the default constructor
If you don't initialize a field in your own constructor, the compiler will not automatically initialize it for you	If you don't initialize a field in your own constructor, the compiler will automatically initialize it for you
Not possible to initialize instance fields at their point of declaration	Possible to initialize instance fields at their point of declaration

Declaring structure variables

After we have defined a structure type, we can use it in exactly the same way as any other type. For example, if we have defined the **Time** structure, we can create variables, fields, and parameters of type **Time**, as shown in this example:

```
struct Time
{
        private int hours, minutes, seconds;
        ...
}
class Example
{
        private Time currentTime;
        public voidMethod(Time parameter)
        {
                Time localVariable;
                ...
        }
}
```

As with enumerations, we can create a **nullable** version of a structure variable by using the '**?**' modifier. We can then assign the null value to the variable:

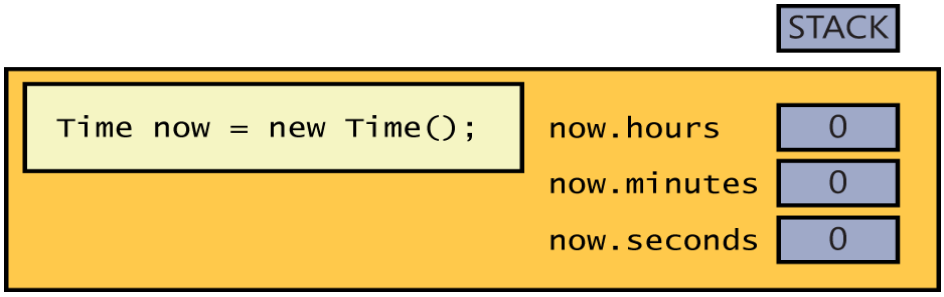
```
Time? currentTime = null;
```

Understanding structure initialization

Earlier we saw how we can initialize the fields in a structure by using a constructor. If we call a constructor, the various rules described earlier guarantee that all the fields in the structure will be initialized:

```
Time now = new Time();
```

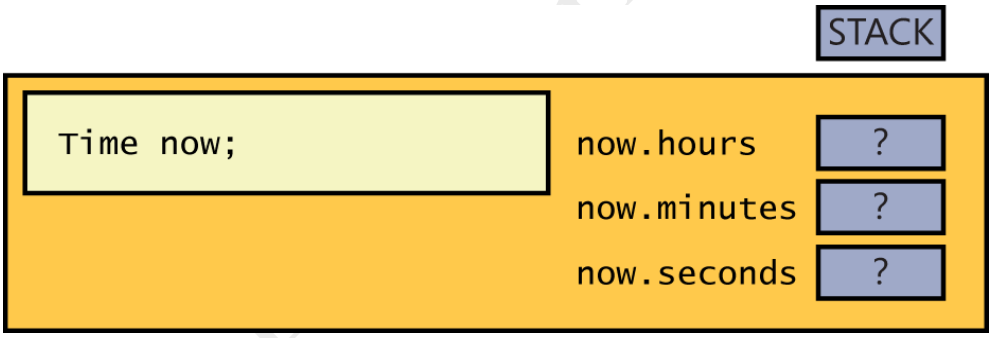
The following graphic depicts the state of the fields in this structure:



However, because structures are value types, we can also create structure variables without calling a constructor, as shown in the following example:

Time now;

This time, the variable is created but its fields are left in their **uninitialized** state. The following graphic depicts the state of the fields in the now variable. Any attempt to access the values in these fields will result in a **compiler error**:



Note that in both cases, the **Time** variable is created on the **stack**.

If we've written our own structure constructor, we can also use that to initialize a structure variable. A structure constructor must always explicitly initialize all its fields. For example:

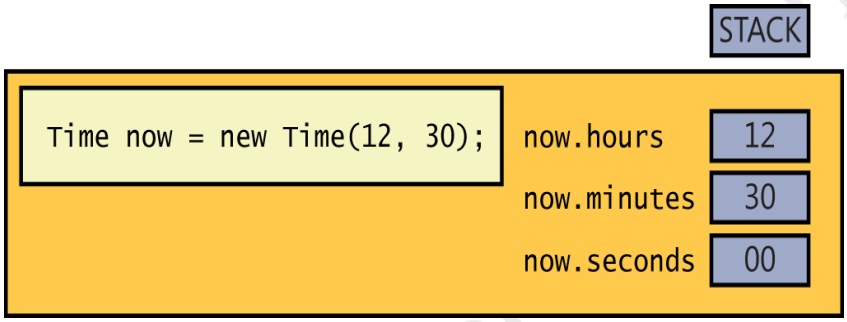
```
struct Time
{
        private int hours, minutes, seconds;
        ...
        public Time(int hh, int mm)
        {
                hours = hh;
```

```
            minutes = mm;
            seconds = 0;
        }
    }
```

The following example initializes now by calling a user-defined constructor:

```
Time now = new Time(12, 30);
```

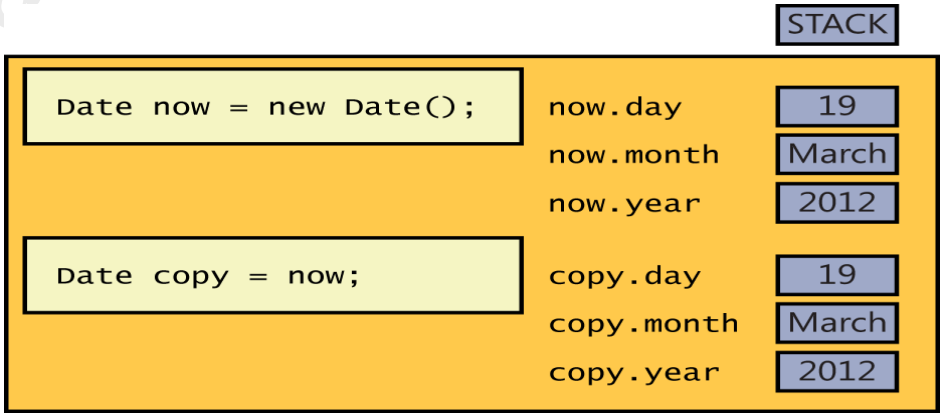
The following graphic shows the effect of this example:



Copying structure variables

We're allowed to initialize or assign one structure variable to another structure variable, but only if the structure variable on the right side is completely initialized (that is, if all its fields are populated with valid data rather than undefined values). The following example compiles because now is fully initialized. The graphic shows the results of performing such an assignment.

```
Date now = new Date();
Date copy = now;
```



The following example fails to compile because now is not initialized:

```
Date now;
Date copy = now; // compile-time error: now has not been assigned
```

When we copy a structure variable, each field on the left side is set directly from the corresponding field on the right side. This copying is done as a fast, single operation that copies the contents of the entire structure and that never throws an exception. Compare this behavior with the equivalent action if **Time** were a class, in which case both variables (now and copy) would end up referencing the same object on the heap.

Chapter 10

Using arrays

Declaring and creating an array

- An ***array*** is an unordered sequence of items.
- All the items in an array have the same type, unlike the fields in a structure or class, which can have different types.
- The items in an array live in a contiguous block of memory and are accessed by using an index, unlike fields in a structure or class, which are accessed by name.

Declaring array variables

- Declare an array variable by specifying the name of the element type, followed by a pair of square brackets, followed by the variable name.
- The square brackets signify that the variable is an array.
- For example, to declare an array of *int* variables named *pins* (for holding a set of personal identification numbers) you can write the following:

```
int[] pins; // Personal Identification Numbers
```

Arrays are not restricted to primitive types as array elements. You can also create arrays of structures, enumerations, and classes. For example, you can create an array of *Date* structures like this:

```
Date[] dates;
```

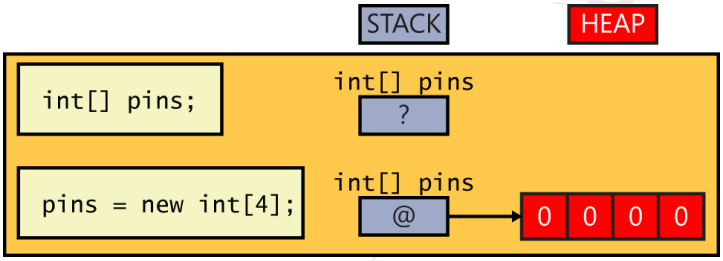
Creating an array instance

- Arrays are **reference types**, regardless of the type of their elements.
- This means that an array variable *refers* to a contiguous block of memory holding the array elements on the heap, just as a class variable refers to an object on the **heap**.
- Even if the array contains a value type such as *int*, the memory will still be allocated on the heap; this is the one case where value types are not allocated memory on the stack.

- Remember that when you declare a class variable, memory is not allocated for the object until you create the instance by using ***new***.
- Arrays follow the same pattern: when you declare an array variable, you do not declare its size and no memory are allocated (other than to hold the reference on the stack).
- **The array is given memory only when the instance is created**, and this is also the point at which you specify the size of the array.
- **To create an array instance, you use the *new* keyword followed by the element type, followed by the size of the array you're creating between square brackets**.
- Creating an array also initializes its elements by using the now familiar default values (*0*, *null*, or *false*, depending on whether the type is numeric, a reference, or a Boolean, respectively). For example, to create and initialize a new array of four integers for the *pins* variable declared earlier, you write this:

pins = new int[4];

The following graphic illustrates what happens when you declare an array, and later when you create an instance of the array:



- Because the memory for the array instance is allocated dynamically, the size of the array does not have to be a constant; it can be calculated at run time, as shown in this example:

int size = int.Parse(Console.ReadLine());
int[] pins = new int[size];

- **You can also create an array whose size is 0**. This might sound bizarre, but it's useful for situations in which the size of the array is determined dynamically and

could even be 0. An array of size 0 is not a ***null*** array; it is an array containing zero elements.

Populating and using an array

- When you create an array instance, all the elements of the array are initialized to a default value depending on their type. For example, all numeric values default to *0*, objects are initialized to *null*, ***DateTime*** values are set to the date and time *"01/01/0001 00:00:00"*, and strings are initialized to *null*.
- You can modify this behavior and initialize the elements of an array to specific values if you prefer.
- This is by providing a comma-separated list of values between a pair of braces. For example, to initialize *pins* to an array of four ***int*** variables whose values are *9*, *3*, *7*, and *2*, you write this:

```
int[] pins = new int[4]{ 9, 3, 7, 2 };
```

The values between the braces do not have to be constants; they can be values calculated at run time, as shown in the following example, which populates the ***pins*** array with four random numbers:

```
Random r = new Random();
int[] pins = new int[4]{ r.Next() % 10, r.Next() % 10,
r.Next() % 10, r.Next() % 10 };
```

The number of values between the braces must exactly match the size of the array instance being created:

```
int[] pins = new int[3]{ 9, 3, 7, 2 }; // compile-time error
int[] pins = new int[4]{ 9, 3, 7 }; // compile-time error
int[] pins = new int[4]{ 9, 3, 7, 2 }; // OK
```

When you're initializing an array variable in this way, you can actually omit the *new* expression and the size of the array. In this case, the compiler calculates the size from the number of initializers and generates code to create the array, such as in the following example:

int[] pins = { 9, 3, 7, 2 };

If you create an array of structures or objects, you can initialize each structure in the array by calling the structure or class constructor, as shown in this example:

Time[] schedule = { new Time(12,30), new Time(5,30) };

Creating an implicitly typed array

- The element type when you declare an array must match the type of elements that you attempt to store in the array. For example, if you declare *pins* to be an array of *int*, as shown in the preceding examples, you cannot store a *double*, *string*, *struct*, or anything that is not an *int* in this array.
- If you specify a list of initializers when declaring an array, you can let the C# compiler infer the actual type of the elements in the array for you, like this:

var names = new[]{"John", "Diana", "James", "Francesca"};

- In this example, the C# compiler determines that the ***names*** variable is an array of strings.
- It is worth pointing out a couple of syntactic quirks in this declaration.
    - First, you omit the square brackets from the type; the ***names*** variable in this example is declared simply as ***var***, not ***var[].***
    - Second, you must specify the ***new*** operator and square brackets before the initializer list.
- If you use this syntax, you must ensure that all the initializers have the same type. This next example causes the compile-time error "No best type found for implicitly typed array":

var bad = new[]{"John", "Diana", 99, 100};

- However, in some cases, the compiler will convert elements to a different type, if doing so makes sense. In the following code, the ***numbers*** array is an array of ***double*** because the constants *3.5* and *99.999* are both ***double***, and the C# compiler can convert the integer values *1* and *2* to ***double*** values:

  var numbers = new[]{1, 2, 3.5, 99.999};

- Implicitly typed arrays are most useful when you are working with anonymous types. The following code creates an array of anonymous objects, each containing two fields specifying the name and age of the members of my family:

  var names = new[] { new { Name = "John", Age = 47 },
  new { Name = "Diana", Age = 46 },
  new { Name = "James", Age = 20 },
  new { Name = "Francesca", Age = 18 } };

- The fields in the anonymous types must be the same for each element of the array.

Accessing an individual array element

- To access an individual array element, you must provide an **index** indicating which element you require.
- Array indexes are zero-based; thus, the initial element of an array lives at **index 0** and not index 1.
- An index value of *1* accesses the second element. For example, you can read the contents of element 2 of the *pins* array into an ***int*** variable by using the following code:

  int myPin;
  myPin = pins[2];

  Similarly, you can change the contents of an array by assigning a value to an indexed element:

  myPin = 1645;

```
pins[2] = myPin;
```

All array element access is bounds-checked. If you specify an index that is less than 0 or greater than or equal to the length of the array, the compiler throws an ***IndexOutOfRangeException*** exception, as in this example:

```
try
{
        int[] pins = { 9, 3, 7, 2 };
        Console.WriteLine(pins[4]); // error, the 4th and last element is at index 3
}
catch (IndexOutOfRangeException ex)
{
        ...
}
```

Iterating through an array

All arrays are actually instances of the *System.Array* class in the Microsoft .NET Framework, and this class defines a number of useful properties and methods.

Length: It is property to discover how many elements an array contains and iterate through all the elements of an array by using a ***for*** statement. The following sample code writes the array element values of the *pins* array to the console:

```
int[] pins = { 9, 3, 7, 2 };
for (int index = 0; index < pins.Length; index++)
{
int pin = pins[index];
Console.WriteLine(pin);
}
```

foreach

- C# provides the ***foreach*** statement with which you can iterate through the elements of an array without worrying about these issues. For example, here's the preceding *for* statement rewritten as an equivalent ***foreach*** statement:

```
int[] pins = { 9, 3, 7, 2 };
foreach (int pin in pins)
{
Console.WriteLine(pin);
}
```

- The ***foreach*** statement declares an **iteration variable** (in this example, ***int pin***) that automatically acquires the value of each element in the array.
- The type of this variable must match the type of the elements in the array.
- The ***foreach*** statement is the preferred way to iterate through an array; it expresses the intention of the code directly, and all of the ***for*** loop scaffolding drops away.
- **However, in a few cases, you'll find that you have to revert to a *for* statement:**
  - A ***foreach*** statement always iterates through the entire array. If you want to iterate through only a known portion of an array (for example, the first half) or bypass certain elements (for example, every third element), it's easier to use a *for* statement.
  - A ***foreach*** statement always iterates from index 0 through index ***Length* – 1**. If you want to iterate backward or in some other sequence, it's easier to use a ***for*** statement.
  - If the body of the loop needs to know the index of the element rather than just the value of the element, you'll have to use a *for* statement.
  - If you need to modify the elements of the array, you'll have to use a ***for*** statement. This is because the iteration variable of the ***foreach*** statement is a read-only copy of each element of the array.

You can declare the iteration variable as a ***var*** and let the C# compiler work out the type of the variable from the type of the elements in the array. This is especially useful if you don't actually know the type of the elements in the array, such as when the array contains

anonymous objects. The following example demonstrates how you can iterate through the array of family members shown earlier:

```
var names = new[] { new { Name = "John", Age = 47 },
                    new { Name = "Diana", Age = 46 },
                    new { Name = "James", Age = 20 },
foreach (var familyMember in names)
{
        Console.WriteLine("Name: {0}, Age: {1}", familyMember.Name,
        familyMember.Age);
}
```

Passing arrays as parameters and return values for a method

- The syntax for passing an array as a parameter is much the same as declaring an array. For example, the code sample that follows defines a method called *ProcessData* that takes an array of integers as a parameter. The body of the method iterates through the array and performs some unspecified processing on each element:

```
public void ProcessData(int[] data)
{
        foreach (int i in data)
        {
                ...
        }
}
```

- It is important to remember that **arrays are reference objects**, so **if you modify the contents of an array passed as a parameter inside a method such as** *ProcessData***, the modification is visible through all references to the array, including the original argument passed as the parameter**. To return an array from a method, you specify the type of the array as the return type. In the method, you create and populate the array.
- The following example prompts the user for the size of an array, followed by the data for each element. The array created by the method is passed back as the return value:

```
public int[] ReadData()
{
        Console.WriteLine("How many elements?");
        string reply = Console.ReadLine();
        int numElements = int.Parse(reply);
        int[] data = new int[numElements];
        for (int i = 0; i < numElements; i++)
        {
                Console.WriteLine("Enter data for element {0}", i);
                reply = Console.ReadLine();
                int elementData = int.Parse(reply);
                data[i] = elementData;
        }
        return data;
}
```

You can call the *ReadData* method like this:

```
int[] data = ReadData();
```

Copying arrays

- Arrays are reference types (remember that an array is an instance of the ***System.Array*** class). **An array variable contains a reference to an array instance**. This means that when you copy an array variable, you actually end up with two references to the same array instance, as demonstrated in the following example:

```
int[] pins = { 9, 3, 7, 2 };
int[] alias = pins; // alias and pins refer to the same array instance
```

In this example, if you modify the value at *pins[1]*, the change will also be visible by reading *alias[1]*.

If you want to make a copy of the array instance (the data on the heap) that an array variable refers to, you have to do two things. First, you create a new array instance of the same type and the same length as the array you are copying. Second, you copy the data element by element from the original array to the new array, as in this example:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
for (int i = 0; i < pins.Length; i++)
{
copy[i] = pins[i];
}
```

Note that this code uses the ***Length*** property of the original array to specify the size of the new array.

- *System.Array* class provides some useful methods that you can employ to copy an array rather than writing your own code. For example, the ***CopyTo*** method copies the contents of one array into another array given a specified starting index. The following example copies all the elements from the *pins* array to the *copy* array starting at element zero:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
pins.CopyTo(copy, 0);
```

Another way to copy the values is to use the *System.Array* static method named ***Copy*.** As with ***CopyTo***, you must initialize the target array before calling ***Copy***:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
Array.Copy(pins, copy, copy.Length);
```

Yet another alternative is to use the *System.Array* instance method named ***Clone*.** You can call this method to create an entire array and copy it in one action:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = (int[])pins.Clone();
```

Using multidimensional arrays

The arrays shown so far have contained a single dimension, and you can think of them as simple lists of values. You can create arrays with more than one dimension. For example, to create a two-dimensional array, you specify an array that requires two integer indexes. The following code creates a two dimensional array of 24 integers called *items*. If it helps, you can think of the array as a table with the first dimension specifying a number of rows, and the second specifying a number of columns.

```
int[,] items = new int[4, 6];
```

To access an element in the array, you provide two index values to specify the "**cell**" holding the element. (A cell is the intersection of a row and a column.) The following code shows some examples using the *items* array:

```
items[2, 3] = 99; // set the element at cell(2,3) to 99
items[2, 4] = items [2,3]; // copy the element in cell(2, 3) to cell(2, 4)
items[2, 4]++; // increment the integer value at cell(2, 4)
```

There is no limit on the number of dimensions that you can specify for an array. The next code example creates and uses an array called *cube* that contains three dimensions. Notice that you must specify three indexes to access each element in the array.

```
int[, ,] cube = new int[5, 5, 5];
cube[1, 2, 1] = 101;
cube[1, 2, 2] = cube[1, 2, 1] * 3;
```

At this point, it is worth giving a word of caution about creating arrays with more than three dimensions. Specifically, arrays can consume a lot of memory. The *cube* array contains 125 elements (5 * 5 * 5). A four-dimensional array for which each dimension has a size of 5 contains 625 elements. **If you start to create arrays with three or more dimensions, you**

can soon run out of memory. Therefore, you should always be prepared to catch and handle ***OutOfMemoryException*** exceptions when you use multidimensional arrays.

Creating jagged arrays

In C#, ordinary multidimensional arrays are sometimes referred to as *rectangular* arrays. Each dimension has a regular shape. For example, in the following tabular two-dimensional *items* array, every row has a column containing 40 elements, and there are 160 elements in total:

```
int[,] items = new int[4, 40];
```

As mentioned in the previous section, multidimensional arrays can consume a lot of memory. If the application uses only some of the data in each column, allocating memory for unused elements is a waste. In this scenario, you can use a ***jagged array***, for which each column has a different length, like this:

```
int[][] items = new int[4][];
int[] columnForRow0 = new int[3];
int[] columnForRow1 = new int[10];
int[] columnForRow2 = new int[40];
int[] columnForRow3 = new int[25];
items[0] = columnForRow0;
items[1] = columnForRow1;
items[2] = columnForRow2;
items[3] = columnForRow3;
...
```

In this example, the application requires only 3 elements in the first column, 10 elements in the second column, 40 elements in the third column, and 25 elements in the final column. This code illustrates an array of arrays—rather than *items* being a two-dimensional array, it has only a single dimension, but the elements in that dimension are themselves arrays. Furthermore, the total size of the *items* array is 78 elements rather than 160; no space is allocated for elements that the application is not going to use.

It is worth highlighting some of the syntax in this example. The following declaration specifies that *items* is an array of arrays of *int*.

int[][] items;

The following statement initializes *items* to hold four elements, each of which is an array of indeterminate length:

items = new int[4][];

The arrays *columnForRow0* to c*olumnForRow3* are all single-dimensional *int* arrays, initialized to hold the required amount of data for each column. Finally, each column array is assigned to the appropriate elements in the *items* array, like this:

items[0] = columnForRow0;

Recall that arrays are reference objects, so this statement simply adds a reference to *columnFor-Row0* to the first element in the *items* array; it does not actually copy any data. You can populate data in this column either by assigning a value to an indexed element in *columnForRow0* or by referencing it through the *items* array. The following statements are equivalent:

columnForRow0[1] = 99;
items[0][1] = 99;

You can extend this idea further if you want to create arrays of arrays of arrays rather than rectangular three-dimensional arrays, and so on.