```
partial
where
equals
into
remove
yield
```


With contextual keywords, ambiguity cannot arise within the context in which they are used.

Literals, Punctuators, and Operators

Literals are primitive pieces of data statically embedded into the program. The lit-

Literals are primitive pieces of data statically embedded into the program. The literals we used in our example program are `12` and `30`.

Punctuators help demarcate the structure of the program. These are the punctuators we used in our example program:

```
;  {  }
```

The semicolon is used to terminate a statement. This means that statements can wrap multiple lines:

```
Console.WriteLine
  (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

The braces are used to group multiple statements into a statement block.

An *operator* transforms and combines expressions. Most operators in C# are denoted with a symbol, such as the multiplication operator, *. We will discuss operators in more detail later in the chapter. These are the operators we used in our example program:

```
.  ()  *  =
```

The period denotes a member of something (or a decimal point with numeric liter-

The period denotes a member of something (or a decimal point with numeric literals). The parentheses are used when declaring or calling a method; empty parentheses are used when the method accepts no arguments. The equals sign is used for *assignment* (the double equals sign, ==, is used for equality comparison, as we'll see later).

Comments

C# offers two different styles of source-code documentation: *single-line comments* and *multiline comments*. A single-line comment begins with a double forward slash and continues until the end of the line. For example:

```
int x = 3;   // Comment about assigning 3 to x
```

A multiline comment begins with /* and ends with */. For example:

```
int x = 3;   /* This is a comment that
                spans two lines */
```

Comments may embed XML documentation tags, explained in "XML Documentation" on page 176 in Chapter 4.

tation" on page 176 in Chapter 4.

Type Basics

A *type* defines the blueprint for a value. A *value* is a storage location denoted by a *variable* or a *constant*. A variable represents a value that can change, whereas a

constant represents an invariant (we will visit constants later in the chapter). We created a local variable named x in our first program:

```
static void Main()
{
  int x = 12 * 30;
  Console.WriteLine (x);
}
```

All values in C# are an *instance* of a specific type. The meaning of a value, and the set of possible values a variable can have, is determined by its type. The type of x is `int`.

set of possible values a variable can have, is determined by its type. The type of x is int.

Predefined Type Examples

Predefined types are types that are specially supported by the compiler. The int type is a predefined type for representing the set of integers that fit into 32 bits of memory, from -2^{31} to $2^{31}-1$. We can perform functions such as arithmetic with instances of is a predefined type for representing the set of integers that fit into 32 bits of memory, from -2^{31} to $2^{31}-1$. We can perform functions such as arithmetic with instances of the `int` type as follows:

```
int x = 12 * 30;
```

Another predefined C# type is `string`. The `string` type represents a sequence of characters, such as ".NET" or "*http://oreilly.com*". We can work with strings by calling functions on them as follows:

```
string message = "Hello world";
string upperMessage = message.ToUpper();
Console.WriteLine (upperMessage);
```

```
// HELLO WORLD
// Hello world2010
```

```
int x = 2010;
message = message + x.ToString();
Console.WriteLine (message);
```

The predefined `bool` type has exactly two possible values: `true` and `false`. The `bool` type is commonly used to conditionally branch execution flow based with an `if` statement. For example:

```
bool simpleVar = false;
if (simpleVar)
  Console.WriteLine ("This will not print");
```

```
int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
```

```
if (lessThanAMile)
  Console.WriteLine ("This will print");
```

In C#, predefined types (also referred to as built-in types) are recognized with a C# keyword. The `System` namespace in the .NET Framework contains many important types that are not predefined by C# (e.g., `DateTime`).

Custom Type Examples

Just as we can build complex functions from simple functions, we can build complex types from primitive types. In this example, we will define a custom type named `UnitConverter`—a class that serves as a blueprint for unit conversions:

```
using System;
```

```
using System;

public class UnitConverter
{
  int ratio;                                                    // Field
  public UnitConverter (int unitRatio) {ratio = unitRatio; } // Constructor
  public int Convert   (int unit)    {return unit * ratio; } // Method
}

class Test
{
  static void Main()
  {
    UnitConverter feetToInchesConverter = new UnitConverter (12);
    UnitConverter milesToFeetConverter  = new UnitConverter (5280);

    Console.WriteLine (feetToInchesConverter.Convert(30));
    Console.WriteLine (feetToInchesConverter.Convert(100));
    Console.WriteLine (feetToInchesConverter.Convert(
                         milesToFeetConverter.Convert(1)));
  }
```

```
  }
}
```

```
// 360
// 1200
// 63360
```

Members of a type

A type contains *data members* and *function members*. The data member of `UnitConverter` is the *field* called `ratio`. The function members of `UnitConverter` are the `Convert` method and the `UnitConverter`'s *constructor*.

Symmetry of predefined types and custom types

A beautiful aspect of C# is that predefined types and custom types have few differences. The predefined `int` type serves as a blueprint for integers. It holds data—32

A beautiful aspect of C# is that predefined types and custom types have few differences. The predefined `int` type serves as a blueprint for integers. It holds data—32 bits—and provides function members that use that data, such as `ToString`. Similarly, our custom `UnitConverter` type acts as a blueprint for unit conversions. It holds data—the ratio—and provides function members to use that data.

Constructors and instantiation

Data is created by *instantiating* a type. Predefined types can be instantiated simply by using a literal. For example, the following line instantiates two integers (`12` and `30`), which are used to compute a third instance, `x`:

```
int x = 12 * 30;
```

The `new` operator is needed to create a new instance of a custom type. We created and declared an instance of the `UnitConverter` type with this statement:

```
UnitConverter feetToInchesConverter = new UnitConverter (12);
```


Immediately after the `new` operator instantiates an object, the object's *constructor* is

Immediately after the `new` operator instantiates an object, the object's *constructor* is called to perform initialization. A constructor is defined like a method, except that the method name and return type are reduced to the name of the enclosing type:

```
public class UnitConverter
{
  ...
  public UnitConverter (int unitRatio) { ratio = unitRatio; }
  ...
}
```

Instance versus static members

The data members and function members that operate on the *instance* of the type are called instance members. The `UnitConverter`'s `Convert` method and the `int`'s `ToString` method are examples of instance members. By default, members are instance members.

[redacted]

Data members and function members that don't operate on the instance of the type, but rather on the type itself, must be marked as `static`. The `Test.Main` and `Console.WriteLine` methods are static methods. The `Console` class is actually a *static class*, which means *all* its members are static. You never actually create instances of a `Console`—one console is shared across the whole application.

To contrast instance from static members, in the following code the instance field `Name` pertains to an instance of a particular `Panda`, whereas `Population` pertains to the set of all `Panda` instances:

set of all Panda instances.

```
public class Panda
{
  public string Name;
  public static int Population;
  public Panda (string n)
  {
    Name = n;
    Population = Population + 1;
  }
}
```

```
}
```

```
// Instance field
// Static field
// Constructor
// Assign the instance field
// Increment the static Population field
```

The following code creates two instances of the `Panda`, prints their names, and then prints the total population:

```
using System;
```

```
class Program
{
  static void Main()
```

```
  static void Main()
  {
    Panda p1 = new Panda ("Pan Dee");
    Panda p2 = new Panda ("Pan Dah");

Console.WriteLine (p1.Name);
Console.WriteLine (p2.Name);

// Pan Dee
// Pan Dah

Console.WriteLine (Panda.Population);

// 2
```

```
// 2
```



```
    }
}
```

The public keyword

The `public` keyword exposes members to other classes. In this example, if the `Name` field in `Panda` was not public, the `Test` class could not access it. Marking a member `public` is how a type communicates: “Here is what I want other types to see—everything else is my own private implementation details.” In object-oriented terms, we say that the public members *encapsulate* the private members of the class.

Conversions

C# can convert between instances of compatible types. A conversion always creates a new value from an existing one. Conversions can be either *implicit* or *explicit*: implicit conversions happen automatically, and explicit conversions require a *cast*. In the following example, we *implicitly* cast an `int` to a `long` type (which has twice the bitwise capacity of an `int`) and *explicitly* cast an `int` to a `short` type (which has half the capacity of an `int`):

```
int x = 12345;       // int is a 32-bit integer
long y = x;          // Implicit conversion to 64-bit integer
short z = (short)x;  // Explicit conversion to 16-bit integer
```

Implicit conversions are allowed when both of the following are true:

-
-

The compiler can guarantee they will always succeed.

No information is lost in conversion.*

Conversely, *explicit* conversions are required when one of the following is true:

Conversely, *explicit* conversions are required when one of the following is true:

-
-

The compiler cannot guarantee they will always succeed.

Information may be lost during conversion.

> The *numeric conversions* that we just saw are built into the language. C# also supports *reference conversions* and *boxing conversions* (see Chapter 3) as well as *custom conversions* (see "Operator Overloading" on page 153 in Chapter 4). The compiler doesn't enforce the aforementioned rules with custom conversions, so it's possible for badly designed types to behave otherwise.

Value Types Versus Reference Types

All C# types fall into the following categories:

All C# types fall into the following categories:

- Value types
- Reference types

* A minor caveat is that very large `long` values lose some precision when converted to `double`.

-
-

Generic type parameters

Pointer types

In this section, we'll describe value types and reference types. In "Generics" on page 101 in Chapter 3, we'll cover generic

In "Generics" on page 101 in Chapter 3, we'll cover generic type parameters, and in "Unsafe Code and Pointers" on page 170 in Chapter 4, we'll cover pointer types.

Value types comprise most built-in types (specifically, all numeric types, the `char` type, and the `bool` type) as well as custom `struct` and `enum` types.

Reference types comprise all class, array, delegate, and interface types.

[redacted]

The fundamental difference between value types and reference types is how they are handled in memory.

Value types

The content of a *value type* variable or constant is simply a value. For example, the content of the built-in value type, `int`, is 32 bits of data.

You can define a custom value type with the `struct` keyword (see Figure 2-1):

```
public struct Point { public int X, Y; }
```

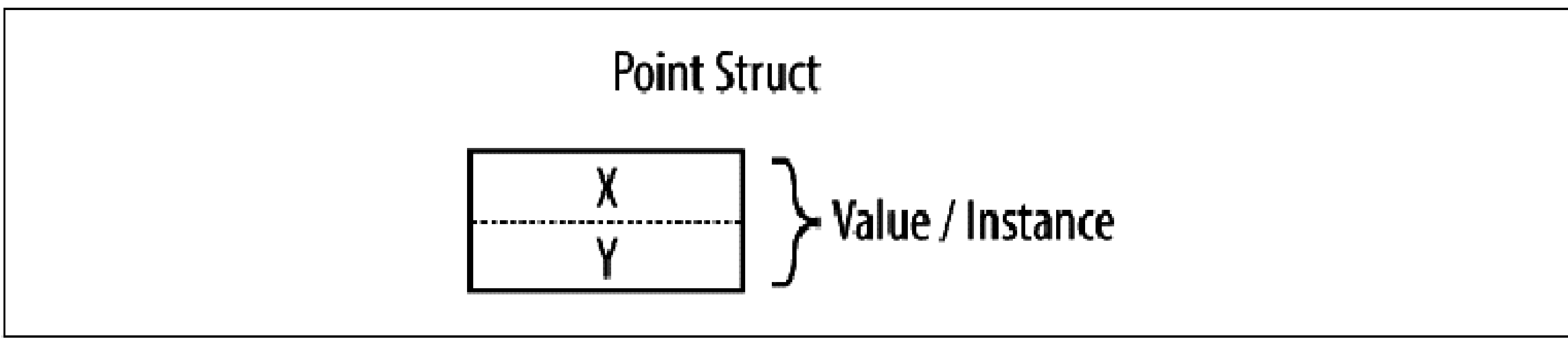


Figure 2-1. A value type instance in memory

The assignment of a value type instance always *copies* the instance. For example:

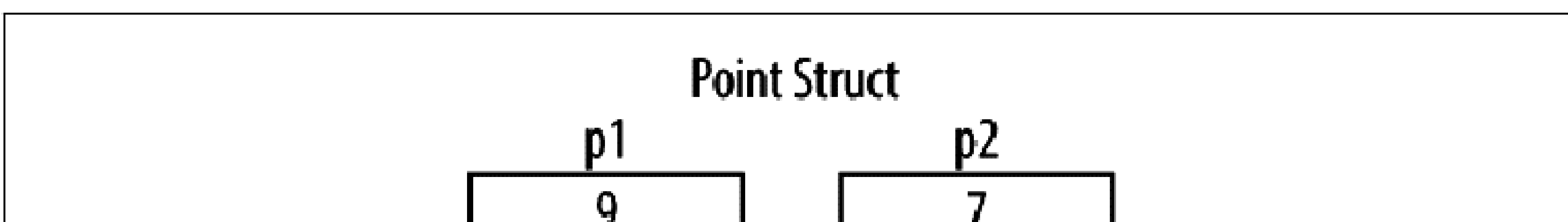
The assignment of a value type instance always *copies* the instance. For example:

```
static void Main()
{
  Point p1 = new Point();
  p1.X = 7;

  Point p2 = p1;
  Console.WriteLine (p1.X);
  Console.WriteLine (p2.X);
  p1.X = 9;
  Console.WriteLine (p1.X);
  Console.WriteLine (p2.X);
}
// Assignment causes copy
```

```
// Assignment causes copy
// 7
// 7
// Change p1.X
// 9
// 7
```

Figure 2-2 shows that p1 and p2 have independent storage.



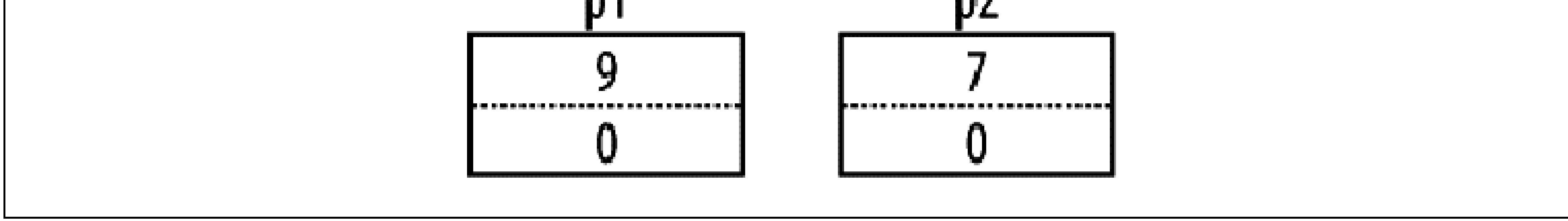
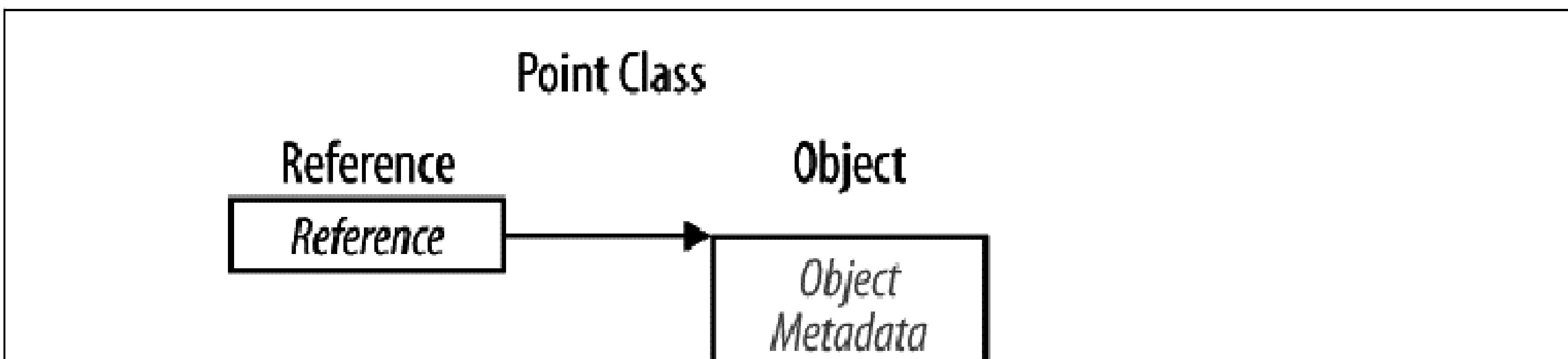


Figure 2-2. Assignment copies a value-type instance

Reference types

A reference type is more complex than a value type, having two parts: an *object* and the *reference* to that object. The content of a reference-type variable or constant is a reference to an object that contains the value. Here is the `Point` type from our previous example rewritten as a class, rather than a `struct` (shown in Figure 2-3):

```
public class Point { public int X, Y; }
```



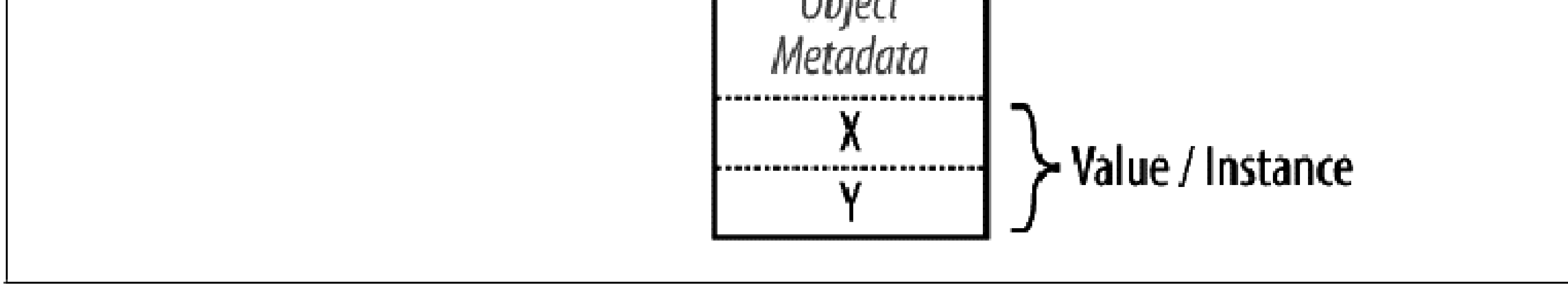


Figure 2-3. A reference-type instance in memory

Assigning a reference-type variable copies the reference, not the object instance. This allows multiple variables to refer to the same object—something not ordinarily possible with value types. If we repeat the previous example, but with `Point` now a class, an operation to `X` affects `Y`:

```
static void Main()
{
  Point p1 = new Point();
  p1.X = 7;

  Point p2 = p1;
  Console.WriteLine (p1.X);
  Console.WriteLine (p2.X);
```

```
  Console.WriteLine (p2.X);
  p1.X = 9;
  Console.WriteLine (p1.X);
  Console.WriteLine (p2.X);
}
```

```
// Copies p1 reference
// 7
// 7
// Change p1.X
// 9
// 9
```

```
// 9
```

Figure 2-4 shows that `p1` and `p2` are two references that point to the same object.

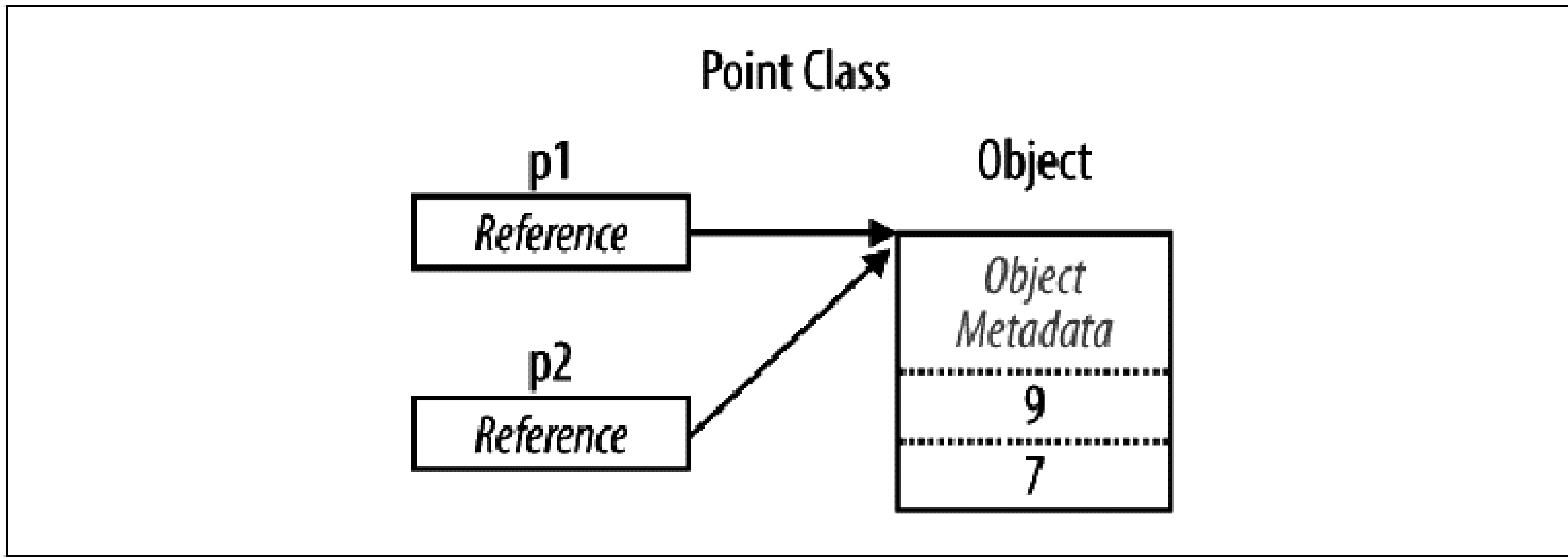


Figure 2-4. Assignment copies a reference

[redacted]

Null

A reference can be assigned the literal `null`, indicating that the reference points to no object:

```
class Point {...}
...
```

```
Point p = null;
```

```
Point p = null;
Console.WriteLine (p == null);   // True
// The following line generates a runtime error
// (a NullReferenceException is thrown):
Console.WriteLine (p.X);
```

In contrast, a value type cannot ordinarily have a null value:

```
struct Point {...}
...
```

```
Point p = null;
int x = null;
```

```
// Compile-time error
// Compile-time error
```

```
// Compile-time error
```

C# also has a construct called *nullable types* for representing value-type nulls (see "Nullable Types" on page 148 in Chapter 4).

Storage overhead

Value-type instances occupy precisely the memory required to store their fields. In this example, `Point` takes eight bytes of memory:

```
struct Point
{
  int x;  // 4 bytes
  int y;  // 4 bytes
}
```

```
  int y;  // 4 bytes
}
```


Technically, the CLR positions fields within the type at an address that's a multiple of the fields' size (up to a maximum of 8 bytes). Thus, the following actually consumes 16 bytes of memory (with the 7 bytes following the first field "wasted"):

```
struct A { byte b; long l; }
```

Reference types require separate allocations of memory for the reference and object. The object consumes as many bytes as its fields, plus additional administrative overhead. The precise overhead is intrinsically private to the implementation of the .NET runtime, but at minimum the overhead is eight bytes, used to store a key to the object's type, as well as temporary information such as its lock state for multithreading and a flag to indicate whether it has been fixed from movement by the garbage collector. Each reference to an object requires an extra 4 or 8 bytes, depending on whether the .NET runtime is running on a 32- or 64-bit platform.

depending on whether the .NET runtime is running on a 32- or 64-bit platform.

Predefined Type Taxonomy

The predefined types in C# are:

Value types

- Numeric
  - Signed integer (`sbyte`, `short`, `int`, `long`)
  - Unsigned integer (`byte`, `ushort`, `uint`, `ulong`)
  - Real number (`float`, `double`, `decimal`)
- Logical (`bool`)
- Character (`char`)

Reference types

- String (`string`)

- String (`string`)
- Object (`object`)

Predefined types in C# alias Framework types in the `System` namespace. There is only a syntactic difference between these two statements:

```
int i = 5;
System.Int32 i = 5;
```

The set of predefined *value* types excluding `decimal` are known as *primitive types* in the CLR. Primitive types are so called because they are supported directly via instructions in compiled code, and this usually translates to direct support on the underlying processor. For example:

```
                   // Underlying hexadecimal representation
int i = 7;         // 0x7
bool b = true;     // 0x1
char c = 'A';      // 0x41
float f = 0.5f;    // uses IEEE floating-point encoding
```

The `System.IntPtr` and `System.UIntPtr` types are also primitive (see Chapter 25).

The `System.IntPtr` and `System.UIntPtr` types are also primitive (see Chapter 25).

Numeric Types

C# has the predefined numeric types shown in Table 2-1.

Table 2-1. Predefined numeric types in C#

C# type	System type	Suffix	Size	Range
Integral—signed				
sbyte	SByte		8 bits	-2^7 to 2^7-1
short	Int16		16 bits	-2^{15} to $2^{15}-1$
int	Int32			

Int32

32 bits

-2^{31} to $2^{31}-1$

C# Basics

long	Int64	L	64 bits	-2^{63} to $2^{63}-1$

long	Int64	L	64 bits	-2^{63} to $2^{63}-1$
Integral—unsigned				
byte	Byte		8 bits	0 to $2^{8}-1$
ushort	UInt16		16 bits	0 to $2^{16}-1$
uint	UInt32	U	32 bits	0 to $2^{32}-1$
ulong	UInt64	UL	64 bits	0 to $2^{64}-1$
Real				
float	Single	F	32 bits	$\pm(\sim10^{-45}$ to $10^{38})$

`float`	`Single`	`F`	32 bits	$\pm(\sim10^{-45}$ to $10^{38})$
`double`	`Double`	`D`	64 bits	$\pm(\sim10^{-324}$ to $10^{308})$
`decimal`	`Decimal`	`M`	128 bits	$\pm(\sim10^{-28}$ to $10^{28})$

Of the *integral* types, `int` and `long` are first-class citizens and are favored by both C# and the runtime. The other integral types are typically used for interoperability or when space efficiency is paramount.

Of the *real* number types, `float` and `double` are called *floating-point types*† and are typically used for scientific calculations. The `decimal` type is typically used for financial calculations, where base-10-accurate arithmetic and high precision are required.

Numeric Literals

Integral literals can use decimal or hexadecimal notation; hexadecimal is denoted with the `0x` prefix. For example:

```
int x = 127;
```

```
int x = 127;
long y = 0x7F;
```

Real literals can use decimal and/or exponential notation. For example:

```
double d = 1.5;
double million = 1E06;
```

† Technically, `decimal` is a floating-point type too, although it's not referred to as such in the C# language specification.

Numeric literal type inference

By default, the compiler *infers* a numeric literal to be either `double` or an integral type:

•

-
-

If the literal contains a decimal point or the exponential symbol (`E`), it is a `double`.

Otherwise, the literal's type is the first type in this list that can fit the literal's value: `int`, `uint`, `long`, and `ulong`.

For example:

```
Console.WriteLine (        1.0.GetType());
Console.WriteLine (       1E06.GetType());
Console.WriteLine (          1.GetType());
Console.WriteLine ( 0xF0000000.GetType());
```

Numeric suffixes

```
// Double
// Double
```

```
// Double
// Int32
// UInt32
```

(double)
(double)
(int)
(uint)

Numeric suffixes explicitly define the type of a literal. Suffixes can be either lower- or uppercase, and are as follows:

Category	C# type	Notes	Example
F	float		float f = 1.0F;

F	float		`float f = 1.0F;`
D	double		`double d = 1D;`
M	decimal		`decimal d = 1.0M;`
U	uint or ulong	Combinable with L	`uint i = 1U;`
L	long or ulong	Combinable with U	`ulong i = 1UL;`

The suffixes U and L are rarely necessary, because the uint, long, and ulong types can nearly always be either *inferred* or *implicitly converted* from int:

The suffixes `U` and `L` are rarely necessary, because the `uint`, `long`, and `ulong` types can nearly always be either *inferred* or *implicitly converted* from `int`:

```
long i = 5;     // Implicit lossless conversion from int literal to long
```

The `D` suffix is technically redundant, in that all literals with a decimal point are inferred to be `double`. And you can always add a decimal point to a numeric literal:

```
double x = 4.0;
```

The `F` and `M` suffixes are the most useful and should always be applied when specifying `float` or `decimal` literals. Without the `F` suffix, the following line would not compile, because 4.5 would be inferred to be of type `double`, which has no implicit conversion to `float`:

```
float f = 4.5F;
```

The same principle is true for a decimal literal:

```
decimal d = -1.23M;     // Will not compile without the M suffix.
```

We describe the semantics of numeric conversions in detail in the following section.

We describe the semantics of numeric conversions in detail in the following section.

Numeric Conversions

Integral to integral conversions

Integral conversions are *implicit* when the destination type can represent every possible value of the source type. Otherwise, an *explicit* conversion is required. For example:

```
int x = 12345;       // int is a 32-bit integral
long y = x;          // Implicit conversion to 64-bit integral
short z = (short)x;  // Explicit conversion to 16-bit integral
```

Floating-point to floating-point conversions

A `float` can be implicitly converted to a `double`, since a `double` can represent every possible value of a `float`. The reverse conversion must be explicit.

Floating-point to integral conversions

All integral types may be implicitly converted to all floating-point numbers:

All integral types may be implicitly converted to all floating-point numbers:

```
int i = 1;
float f = i;
```

The reverse conversion must be explicit:

```
int i2 = (int)f;
```

When you cast from a floating-point number to an integral, any fractional portion is truncated; no rounding is performed. The static class `System.Convert` provides methods that round while converting between various numeric types (see Chapter 6).

Implicitly converting a large integral type to a floating-point type preserves *magnitude* but may occasionally lose *precision*. This is because floating-point types always have more magnitude than integral types, but may have less precision. Rewriting our example with a larger number demonstrates this:

our example with a larger number demonstrates this:

```
int i1 = 100000001;
float f = i1;
int i2 = (int)f;
```

```
// Magnitude preserved, precision lost
// 100000000
```

Decimal conversions

All integral types can be implicitly converted to the decimal type, since a decimal can represent every possible C# integral value. All other numeric conversions to and from a decimal type must be explicit.

Arithmetic Operators

Arithmetic Operators

The arithmetic operators (+, -, *, /, %) are defined for all numeric types except the 8- and 16-bit integral types:


```
+    Addition
-    Subtraction
*    Multiplication
/    Division
%    Remainder after division
```

Increment and Decrement Operators

The increment and decrement operators (++, --) increment and decrement numeric types by 1. The operator can either precede or follow the variable, depending on whether you want the variable to be updated *before* or *after* the expression is evaltypes by 1. The operator can either precede or follow the variable, depending on whether you want the variable to be updated *before* or *after* the expression is evaluated. For example:

```
int x = 0;
Console.WriteLine (x++);   // Outputs 0; x is now 1
Console.WriteLine (++x);   // Outputs 2; x is now 2
Console.WriteLine (--x);   // Outputs 1; x is now 1
```

Specialized Integral Operations

Integral division

Division operations on integral types always truncate remainders. Dividing by a variable whose value is zero generates a runtime error (a `DivideByZeroException`):

```
int a = 2 / 3;      // 0
```

```
int b = 0;
int c = 5 / b;
```

```
int c = 5 / b;
```

```
// throws DivisionByZeroException
```

Dividing by the *literal* 0 generates a compile-time error.

Integral overflow

At runtime, arithmetic operations on integral types can overflow. By default, this happens silently—no exception is thrown. Although the C# specification is agnostic as to the result of an overflow, the CLR always causes wraparound behavior. For example, decrementing the minimum possible `int` value results in the maximum possible `int` value:

```
int a = int.MinValue;
a--;
Console.WriteLine (a == int.MaxValue); // True
```

Integral arithmetic overflow check operators

Integral arithmetic overflow check operators

The `checked` operator tells the runtime to generate an `OverflowException` rather than failing silently when an integral expression or statement exceeds the arithmetic limits of that type. The `checked` operator affects expressions with the `++`, `--`, `+`, `-` (binary and unary), `*`, `/`, and explicit conversion operators between integral types.

`checked` can be used around either an expression or a statement block. For example:

```
int a = 1000000;
int b = 1000000;
```



```
int c = checked (a * b);
checked
{
```

```
{
    ...
    c = a * b;
    ...
}
```

```
// Checks just the expression.
```

```
// Checks all expressions
// in statement block.
```

You can make arithmetic overflow checking the default for all expressions in a program by compiling with the `/checked+` command-line switch (in Visual Studio, go to Advanced Build Settings). If you then need to disable overflow checking just for specific expressions or statements, you can do so with the `unchecked` operator. For example, the following code will not throw exceptions—even if compiled with `/checked+`:

with /checked+:

C# Basics

```
int x = int.MaxValue;
int y = unchecked (x + 1);
unchecked { int z = x + 1; }
```

Overflow checking for constant expressions

Regardless of the /checked compiler switch, expressions evaluated at compile time are always overflow-checked—unless you apply the unchecked operator:

```
int x = int.MaxValue + 1;               // Compile-time error
int y = unchecked (int.MaxValue + 1);   // No errors
```

Bitwise operators

C# supports the following bitwise operators:

Operator	Meaning	Sample expression	Result
~	Complement	~0xfU	0xfffffff0U
&	And	0xf0 & 0x33	0x30
|	Or	0xf0 | 0x33	0xf3
^	Exclusive Or	0xff00 ^ 0x0ff0	0xf0f0

^	Exclusive Or	`0xff00 ^ 0x0ff0`	`0xf0f0`
<<	Shift left	`0x20 << 2`	`0x80`
>>	Shift right	`0x20 >> 1`	`0x10`

8- and 16-Bit Integrals

The 8- and 16-bit integral types are `byte`, `sbyte`, `short`, and `ushort`. These types lack their own arithmetic operators, so C# implicitly converts them to larger types as required. This can cause a compile-time error when trying to assign the result back to a small integral type:

```
short x = 1, y = 1;
short z = x + y;

// Compile-time error
```

In this case, `x` and `y` are implicitly converted to `int` so that the addition can be performed. This means the result is also an `int`, which cannot be implicitly cast back to a `short` (because it could cause loss of data). To make this compile, we must add an explicit cast:

```
short z = (short) (x + y);   // OK
```

Special Float and Double Values

Unlike integral types, floating-point types have values that certain operations treat specially. These special values are NaN (Not a Number), +∞, –∞, and –0. The `float` and `double` classes have constants for `NaN`, +∞, and –∞, as well as other values (`MaxValue`, `MinValue`, and `Epsilon`). For example:

```
Console.WriteLine (double.NegativeInfinity);   // -Infinity
```

The constants that represent special values for `double` and `float` are as follows:

Special value	Double constant	Float constant
NaN	double.NaN	float.NaN
+∞	double.PositiveInfinity	float.PositiveInfinity
−∞	double.NegativeInfinity	float.NegativeInfinity
−0	-0.0	-0.0f

Dividing a nonzero number by zero results in an infinite value. For example:

```
Console.WriteLine ( 1.0 /  0.0);                  //  Infinity
Console.WriteLine (-1.0 /  0.0);                  // -Infinity
Console.WriteLine ( 1.0 / -0.0);                  // -Infinity
Console.WriteLine (-1.0 / -0.0);                  //  Infinity
```

Dividing zero by zero, or subtracting infinity from infinity, results in a NaN. For example:

```
Console.WriteLine ( 0.0 /  0.0);                 //  NaN
Console.WriteLine ((1.0 /  0.0) - (1.0 / 0.0));  //  NaN
```

When using ==, a NaN value is never equal to another value, even another NaN

When using ==, a NaN value is never equal to another value, even another NaN value:

```
Console.WriteLine (0.0 / 0.0 == double.NaN);
```

```
// False
```

To test whether a value is NaN, you must use the `float.IsNaN` or `double.IsNaN` method:

```
Console.WriteLine (double.IsNaN (0.0 / 0.0));
```

```
// True
```

When using `object.Equals`, however, two NaN values are equal:

```
Console.WriteLine (object.Equals (0.0 / 0.0, double.NaN));
```

```
// True
```

```
// True
```


NaNs are sometimes useful in representing special values. In WPF, **`double.NaN`** represents a measurement whose value is "Automatic." Another way to represent such a value is with a nullable type (Chapter 4); another is with a custom struct that wraps a numeric type and adds an additional field (Chapter 3).

`float` and `double` follow the specification of the IEEE 754 format types, supported natively by almost all processors. You can find detailed information on the behavior of these types at *http://www.ieee.org*.

double Versus decimal

[redacted]

`double` is useful for scientific computations (such as computing spatial coordinates). `decimal` is useful for financial computations and values that are “man-made” rather than the result of real-world measurements. Here’s a summary of the differences:

Category	double	decimal
Internal representation	Base 2	Base 10
Precision	15–16 significant figures	28–29 significant figures
Range	$\pm(\sim 10^{-324}$ to $\sim 10^{308})$	$\pm(\sim 10^{-28}$ to $\sim 10^{28})$
Special values	+0, −0, +∞, −∞, and NaN	None

Range	$\pm(\sim10^{-324}$ to $\sim10^{308})$	$\pm(\sim10^{-28}$ to $\sim10^{28})$
Special values	+0, −0, +∞, −∞, and NaN	None
Speed	Native to processor	Non-native to processor (about 10 times slower than double)

Real Number Rounding Errors

float and double 1internally represent numbers in base 2. For this reason, only numbers expressible in base 2 are represented precisely. Practically, this means most literals with a fractional component (which are in base 10) will not be represented precisely. For example:

```
float tenth = 0.1f;                       // Not quite 0.1
float one   = 1f;
Console.WriteLine (one - tenth * 10f);    // -1.490116E-08
```

This is why float and double are bad for financial calculations. In contrast, decimal works in base 10 and so can precisely represent numbers expressible in base 10 (as well as its factors, base 2 and base 5). Since real literals are in base 10, decimal can precisely represent numbers such as 0.1. However, neither double nor decimal can precisely represent a fractional number whose base 10 representation is decimal can precisely represent numbers such as 0.1. However, neither double nor decimal can precisely represent a fractional number whose base 10 representation is recurring:

```
decimal m = 1M / 6M;               // 0.1666666666666666666666666667M
double  d = 1.0 / 6.0;             // 0.16666666666666666
```

This leads to accumulated rounding errors:

```
decimal notQuiteWholeM = m+m+m+m+m+m;  // 1.0000000000000000000000000002M
double  notQuiteWholeD = d+d+d+d+d+d;  // 0.99999999999999989
```


which breaks equality and comparison operations:

```
Console.WriteLine (notQuiteWholeM == 1M);   // False
Console.WriteLine (notQuiteWholeD < 1.0);   // True
```

Boolean Type and Operators

C#'s bool type (aliasing the System.Boolean type) is a logical value that can be as-

C#'s `bool` type (aliasing the `System.Boolean` type) is a logical value that can be assigned the literal `true` or `false`.

Although a Boolean value requires only one bit of storage, the runtime will use one byte of memory, since this is the minimum chunk that the runtime and processor can efficiently work with. To avoid space inefficiency in the case of arrays, the Framework provides a `BitArray` class in the `System.Collections` namespace that is designed to use just one bit per Boolean value.

Bool Conversions

No conversions can be made from the `bool` type to numeric types or vice versa.

Equality and Comparison Operators

`==` and `!=` test for equality and inequality of any type, but always return a `bool` value.‡ Value types typically have a very simple notion of equality:

```
int x = 1;
```

```
int x = 1;
int y = 2;
int z = 1;
Console.WriteLine (x == y);
Console.WriteLine (x == z);
```

```
// False
// True
```

For reference types, equality, by default, is based on *reference*, as opposed to the actual *value* of the underlying object (more on this in Chapter 6):

```
public class Dude
{
```

```
{
  public string Name;
  public Dude (string n) { Name = n; }
}
```

```
...
Dude d1 = new Dude ("John");
Dude d2 = new Dude ("John");
Console.WriteLine (d1 == d2);
Dude d3 = d1;
Console.WriteLine (d1 == d3);
```

```
// False
```

```
// False
// True
```

The equality and comparison operators, `==`, `!=`, `<`, `>`, `>=`, and `<=`, work for all numeric types, but should be used with caution with real numbers (as we saw in "Real Number Rounding Errors" on page 27). The comparison operators also work on `enum`

‡ It's possible to *overload* these operators (Chapter 4) such that they return a non-`bool` type, but this is almost never done in practice.

type members, by comparing their underlying integral values. We describe this in "Enums" on page 97 in Chapter 3.

We explain the equality and comparison operators in greater detail in Chapter 4 in the sections "Operator Overloading" on page 153 and "Equality Comparison" on page 245 and in the section "Order Comparison" on page 255 in Chapter 6.

Conditional Operators

Conditional Operators

The `&&` and `||` operators test for *and* and *or* conditions. They are frequently used in conjunction with the `!` operator, which expresses *not*. In this example, the `UseUmbrella` method returns `true` if it's rainy or sunny (to protect us from the rain or the sun), as long as it's not also windy (since umbrellas are useless in the wind):

```
static bool UseUmbrella (bool rainy, bool sunny, bool windy)
{
  return !windy && (rainy || sunny);
}
```

[redacted]

The `&&` and `||` operators *short-circuit* evaluation when possible. In the preceding example, if it is windy, the expression `(rainy || sunny)` is not even evaluated. Short-circuiting is essential in allowing expressions such as the following to run without throwing a `NullReferenceException`:

```
if (sb != null && sb.Length > 0) ...
```

The `&` and `|` operators also test for *and* and *or* conditions:

```
return !windy & (rainy | sunny);
```

The difference is that they *do not short-circuit*. For this reason, they are rarely used in place of conditional operators.

Unlike in C and C++, the `&` and `|` operators perform (non-short-circuiting) *boolean* comparisons when applied to `bool` expressions. The `&` and `|` operators perform *bitwise* operations only sions. The & and | operators perform *bitwise* operations only when applied to numbers.

The ternary conditional operator (simply called the *conditional operator*) has the form `q ? a : b`, where if condition `q` is true, `a` is evaluated, else `b` is evaluated. For example:

```
static int Max (int a, int b)
{
  return (a > b) ? a : b;
}
```

The conditional operator is particularly useful in LINQ queries (Chapter 8).

Strings and Characters

C#'s `char` type (aliasing the `System.Char` type) represents a Unicode character and occupies two bytes. A `char` literal is specified inside single quotes:

occupies two bytes. A `char` literal is specified inside single quotes:

```
char c = 'A';       // Simple character
```

Escape sequences express characters that cannot be expressed or interpreted literally. An escape sequence is a backslash followed by a character with a special meaning. For example:

```
char newLine = '\n';
char backSlash = '\\';
```

The escape sequence characters are shown in Table 2-2.

Table 2-2. Escape sequence characters

Char	Meaning	Value
\'	Single quote	0x0027
\"	Double quote	0x0022

\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

The \u (or \x) escape sequence lets you specify any Unicode character via its four-

The \u (or \x) escape sequence lets you specify any Unicode character via its four-digit hexadecimal code:

```
char copyrightSymbol = '\u00A9';
char omegaSymbol     = '\u03A9';
char newLine         = '\u000A';
```

Char Conversions

An implicit conversion from a char to a numeric type works for the numeric types that can accommodate an unsigned short. For other numeric types, an explicit conversion is required.

String Type

C#'s string type (aliasing the System.String type, covered in depth in Chapter 6) represents an immutable sequence of Unicode characters. A string literal is specified inside double quotes:

```
string a = "Heat";
```

`string` is a reference type, rather than a value type. Its equality operators, however, follow value-type semantics:

```
string a = "test";
string b = "test";
Console.Write (a == b);  // True
```

The escape sequences that are valid for `char` literals also work inside strings:

```
string a = "Here's a tab:\t";
```

The cost of this is that whenever you need a literal backslash, you must write it twice:

```
string a1 = "\\\\server\\fileshare\\helloworld.cs";
```

To avoid this problem, C# allows *verbatim* string literals. A verbatim string literal is prefixed with @ and does not support escape sequences. The following verbatim string is identical to the preceding one:

```
string a2 = @ "\\server\fileshare\helloworld.cs";
```

A verbatim string literal can also span multiple lines:

```
string escaped  = "First Line\r\nSecond Line";
string verbatim = @"First Line
```

```
string verbatim = @"First Line
Second Line";
```

```
// Assuming your IDE uses CR-LF line separators:
Console.WriteLine (escaped == verbatim);  // True
```

You can include the double-quote character in a verbatim literal by writing it twice:

```
string xml = @"<customer id=""123""></customer>";
```

String concatenation

The + operator concatenates two strings:

```
string s = "a" + "b";
```

The righthand operand may be a nonstring value, in which case ToString is called on that value. For example:

on that value. For example:

```
string s = "a" + 5;  // a5
```

Since `string` is immutable, using the + operator repeatedly to build up a string is inefficient: a better solution is to use the `System.Text.StringBuilder` type (described in Chapter 6).

String comparisons

`string` does not support < and > operators for comparisons. You must use the string's `CompareTo` method, described in Chapter 6.

Arrays

An array represents a fixed number of elements of a particular type. The elements in an array are always stored in a contiguous block of memory, providing highly

An array represents a fixed number of elements of a particular type. The elements in an array are always stored in a contiguous block of memory, providing highly efficient access.

An array is denoted with square brackets after the element type. For example:

```
char[] vowels = new char[5];    // Declare an array of 5 characters
```

Square brackets also *index* the array, accessing a particular element by position:

```
vowels [0] = 'a';
vowels [1] = 'e';
vowels [2] = 'i';
vowels [3] = 'o';
vowels [4] = 'u';
Console.WriteLine (vowels [1]);      // e
```

This prints "e" because array indexes start at 0. We can use a `for` loop statement to iterate through each element in the array. The `for` loop in this example cycles the integer `i` from `0` to `4`:

```
for (int i = 0; i < vowels.Length; i++)
  Console.Write (vowels [i]);            // aeiou
```

The `Length` property of an array returns the number of elements in the array. Once

The `Length` property of an array returns the number of elements in the array. Once an array has been created, its length cannot be changed. The `System.Collection` namespace and subnamespaces provide higher-level data structures, such as dynamically sized arrays and dictionaries.

An *array initialization expression* specifies each element of an array. For example:

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

or simply:

```
char[] vowels = {'a','e','i','o','u'};
```

All arrays inherit from the `System.Array` class, providing common services for all arrays. These members include methods to get and set elements regardless of the array type, and are described in "The Array Class" on page 273 in Chapter 7.

Default Element Initialization

Creating an array always preinitializes the elements with default values. The default value for a type is the result of a bitwise zeroing of memory. For example, consider

Creating an array always preinitializes the elements with default values. The default value for a type is the result of a bitwise zeroing of memory. For example, consider creating an array of integers. Since int is a value type, this allocates 1,000 integers in one contiguous block of memory. The default value for each element will be 0:

```
int[] a = new int[1000];
Console.Write (a[123]);            // 0
```


Value types versus reference types

Whether an array element type is a value type or a reference type has important performance implications. When the element type is a value type, each element value is allocated as part of the array. For example:

```
public struct Point { public int X, Y; }
...
```

```
Point[] a = new Point[1000];
int x = a[500].X;
```

```
int x = a[500].X;
```

```
// 0
```

Had `Point` been a class, creating the array would have merely allocated 1,000 null references:


```
public class Point { public int X, Y; }
```

```
public class Point { public int X, Y; }
...
Point[] a = new Point[1000];
int x = a[500].X;
// Runtime error, NullReferenceException
```

To avoid this error, we must explicitly instantiate 1,000 `Points` after instantiating the array:

```
Point[] a = new Point[1000];
for (int i = 0; i < a.Length; i++) // Iterate i from 0 to 999
   a[i] = new Point();             // Set array element i with new point
```

An array *itself* is always a reference type object, regardless of the element type. For instance, the following is legal:

```
int[] a = null;
```

```
int[] a = null;
```

Multidimensional Arrays

Multidimensional arrays come in two varieties: *rectangular* and *jagged*. Rectangular arrays represent an *n*-dimensional block of memory, and jagged arrays are arrays of arrays.

Rectangular arrays

Rectangular arrays are declared using commas to separate each dimension. The following declares a rectangular two-dimensional array, where the dimensions are 3 by 3:

```
int [,] matrix = new int [3, 3];
```

The `GetLength` method of an array returns the length for a given dimension (starting at 0):

```
for (int i = 0; i < matrix.GetLength(0); i++)
```

```
for (int i = 0; i < matrix.GetLength(0); i++)
  for (int j = 0; j < matrix.GetLength(1); j++)
    matrix [i, j] = i * 3 + j;
```

A rectangular array can be initialized as follows (each element in this example is initialized to be identical to the previous example):


```
int[,] matrix = new int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
```

```
};
```

Jagged arrays

Jagged arrays are declared using successive square brackets to represent each dimension. Here is an example of declaring a jagged two-dimensional array, where the outermost dimension is 3:

```
int [][] matrix = new int [3][];
```

The inner dimensions aren't specified in the declaration. Unlike a rectangular array, each inner array can be an arbitrary length. Each inner array is implicitly initialized to null rather than an empty array. Each inner array must be created manually:

```
for (int i = 0; i < matrix.Length; i++)
{
  matrix[i] = new int [3];                    // Create inner array
  for (int j = 0; j < matrix[i].Length; j++)
    matrix[i][j] = i * 3 + j;
}
```

A jagged array can be initialized as follows (each element in this example is initialized

A jagged array can be initialized as follows (each element in this example is initialized to be identical to the previous example):

```
int[][] matrix = new int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8}
};
```

Simplified Array Initialization Expressions

There are two ways to shorten array initialization expressions. The first is to omit the new operator and type qualifications:

```
char[] vowels = {'a','e','i','o','u'};
```

```
int[,] rectangularMatrix =
{
  {0,1,2},
```

```
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
int[][] jaggedMatrix =
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8}
};
```

The second approach is to use the **var** keyword, which tells the compiler to implicitly type a local variable:

```
var i = 3;              // i is implicitly of type int
var s = "sausage";      // s is implicitly of type string
```

```
// Therefore:
var rectMatrix = new int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
var jaggedMat = new int[][]
```

```
};
var jaggedMat = new int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8}
};

// rectMatrix is implicitly of type int[,]
// jaggedMat is implicitly of type int[][]
```

Implicit typing can be taken one stage further with single-dimensional arrays. You can omit the type qualifier after the new keyword and have the compiler *infer* the array type:

```
var vowels = new[] {'a','e','i','o','u'};   // Compiler infers char[]
```

The elements must all be implicitly convertible to a single type in order for implicit array typing to work. For example:

```
var x = new[] {1,10000000000};   // all convertible to long
```

Bounds Checking

Bounds Checking

All array indexing is bounds-checked by the runtime. If you use an invalid index, an `IndexOutOfRangeException` is thrown:

```
int[] arr = new int[3];
arr[3] = 1;                    // IndexOutOfRangeException thrown
```

As with Java, array bounds checking is necessary for type safety and simplifies debugging.

Generally, the performance hit from bounds checking is minor, and the JIT (Just-in-Time) compiler can perform optimizations, such as determining in advance whether all indexes will be safe before entering a loop, thus avoiding a check on each iteration. In addition, C# provides "unsafe" code that can explicitly bypass bounds checking (see the section "Unsafe Code and Pointers" on page 170 in Chapter 4).

Variables and Parameters

A variable represents a storage location that has a modifiable value. A variable can be a *local variable*, *parameter* (*value*, *ref*, or *out*), *field* (*instance* or *static*), or *array element*.

The Stack and the Heap

The stack and the heap are the places where variables and constants reside. Each has very different lifetime semantics.

Stack

The stack is a block of memory for storing local variables and parameters. The stack logically grows and shrinks as a function is entered and exited. Consider the following method (to avoid distraction, input argument checking is ignored):

logically grows and shrinks as a function is entered and exited. Consider the following method (to avoid distraction, input argument checking is ignored):

```
static int Factorial (int x)
{
  if (x == 0) return 1;
  return x * Factorial (x-1);
}
```

This method is recursive, meaning that it calls itself. Each time the method is entered, a new `int` is allocated on the stack, and each time the method exits, the `int` is deallocated.

Heap

The heap is a block of memory in which *objects* (i.e., reference-type instances) reside. Whenever a new object is created, it is allocated on the heap, and a reference to that object is returned. During a program's execution, the heap starts filling up as new objects are created. The runtime has a garbage collector that periodically deallocates objects from the heap, so your computer does not run out of memory. An object is eligible for deallocation as soon as nothing references it.

In the following example, we start by creating a `StringBuilder` object referenced by the variable `ref1`, and then write out its content. That `StringBuilder` object is then immediately eligible for garbage collection, because nothing subsequently uses it.

Then, we create another `StringBuilder` referenced by variable `ref2`, and copy that reference to `ref3`. Even though `ref2` is not used after that point, `ref3` keeps the same `StringBuilder` object alive—ensuring that it doesn't become eligible for collection until we've finished using `ref3`.

```
using System;
using System.Text;

class Test
{
  static void Main()
  {
    StringBuilder ref1 = new StringBuilder ("object1");
```

```
Console.WriteLine (ref1);
// The StringBuilder referenced by ref1 is now eligible for GC.

StringBuilder ref2 = new StringBuilder ("object2");
StringBuilder ref3 = ref2;
// The StringBuilder referenced by ref2 is NOT yet eligible for GC.

    Console.WriteLine (ref3);
  }
}

// object2
```

Value-type instances (and object references) live wherever the variable was declared. If the instance was declared as a field within an object, or as an array element, that

Value-type instances (and object references) live wherever the variable was declared. If the instance was declared as a field within an object, or as an array element, that instance lives on the heap.

You can't explicitly delete objects in C#, as you can in C++. An unreferenced object is eventually collected by the garbage collector.

collector.

The heap also stores static fields and constants. Unlike objects allocated on the heap (which can get garbage-collected), these live until the application domain is torn down.

Definite Assignment

C# enforces a definite assignment policy. In practice, this means that outside of an `unsafe` context, it's impossible to access uninitialized memory. Definite assignment has three implications:

- Local variables must be assigned a value before they can be read.
- Function arguments must be supplied when a method is called.
- All other variables (such as fields and array elements) are automatically initialized by the runtime.

All other variables (such as fields and array elements) are automatically initialized by the runtime.

For example, the following code results in a compile-time error:

```
static void Main()
{
  int x;
  Console.WriteLine (x);        // Compile-time error
}
```

Fields and array elements are automatically initialized with the default values for their type. The following code outputs 0, because array elements are implicitly assigned to their default values:

```
static void Main()
{
  int[] ints = new int[2];
  Console.WriteLine (ints[0]);    // 0
}
```


The following code outputs 0, because fields are implicitly assigned to a default

The following code outputs 0, because fields are implicitly assigned to a default value:

```
class Test
{
  static int x;
  static void Main() { Console.WriteLine (x); }
}
```

```
// 0
```

Default Values

All type instances have a default value. The default value for the predefined types is the result of a bitwise zeroing of memory:

Type	Default value
All reference types	`null`
All numeric and enum types	`0`
`char` type	`'\0'`
`bool` type	`false`

You can obtain the default value for any type using the `default` keyword (in practice, this is useful with generics, which we'll cover in Chapter 3):

```
decimal d = default (decimal);
```

The default value in a custom value type (i.e., `struct`) is the same as the default value for each field defined by the custom type.

Parameters

Parameters

A method has a sequence of parameters. Parameters define the set of arguments that must be provided for that method. In this example, the method `Foo` has a single parameter named `p`, of type `int`:

```
static void Foo (int p)
{
  p = p + 1;                // Increment p by 1
  Console.WriteLine(p);     // Write p to screen
}
static void Main() { Foo (8); }
```

You can control how parameters are passed with the `ref` and `out` modifiers:

Parameter modifier	Passed by	Variable must be definitely assigned
None	Value	Going *in*
`ref`	Reference	Going *in*
`out`	Reference	Going *out*