Glossary

abstract Something that is abstract does not have a "proper" existence as such. When writing programs, we use the word to mean "an idealized description of something." In the case of component design, an abstract class contains descriptions of things that need to be present, but it does not say how they are to be realized. In C# terms, a class is abstract if it is marked as such, or if it contains one or more methods that are marked as abstract.

You can't make an instance of an abstract class, but you can use it as the basis of, or the template for, a concrete one. For example, you might decide that you need many different kinds of sprite in the BreadAndCheese game: bat sprite, ball sprite, target sprite, and so on. We don't know how each particular sprite will work inside, but we do know those behaviors that it must have to make it into a sprite.

We can therefore create an abstract `Sprite` class that serves as the basis of all the concrete ones. Each "real" sprite class is created by extending the parent, abstract one. This means that it is a member of the sprite family (that is, it can be treated as a sprite) but it works in its own way.

Algorithm An algorithm is a description of steps to solve a problem. You can think of it as a recipe if you like. It gives a sequence of steps to be followed and decisions to be taken. A good "getting wet avoidance" algorithm would be "If it is raining, take an umbrella."

Analog An analog value is one which can never be held exactly. Some data items, for example the number of attacking spaceships in a space shooter, or whether or not a button has been pressed, can be represented precisely in a program. Other values, for example the physical position of a thumbstick on the gamepad, cannot be held exactly since there are in theory an infinite number of positions available. Programs often need to manipulate analog values, and they do this by storing them within a particular range and to a particular precision. The analog thumbsticks on a gamepad are represented by a floating point number in an XNA program which has a particular number of decimal places. The `float` and `decimal` types in C# are used to represent analog values.

App Hub If you want to run your Microsoft XNA programs on an Xbox 360, you must be a member of the App Hub. Members of the club pay a membership fee, and their Xbox Live account is extended to include XNA game development. App Hub members can also distribute applications to Windows Phone owners via the Windows Phone Marketplace.

Arithmetic The adjective *arithmetic* is applied to operators that perform some form of calculation on their operands and generate a numeric result. The * (multiply) operator can be used as an arithmetic operator to multiply values together.

Array An array holds a large number of items in a single variable. A one-dimensional array holds a number of values in a single row. You use a subscript to indicate which box in the row you want to use. Consider the following, which creates an array to hold 10 integer high scores and sets all the elements to 0:

```
int[] scores = new int[10];
for (int i = 0; i < 10; i = i + 1)
{
    scores[i] = 0;
}
```

The `int[] scores` part of the code tells the compiler that you want to create an array variable. You can think of this as a reference that can be made to refer to an array of integers. The array itself is created by `new int[10]`. When the program runs, a 10-element array is created; if the value 10 is replaced by a different number, an array of that size is made. Each item in the array is called an element. In the program, you identify which element you mean by putting its number in square brackets [ ] after the array name. This part is called the subscript. The size of an array can be set using an expression as well as a constant, allowing the program to create exactly the right-sized array for a given task.

Arrays can have more than one dimension; a two-dimensional array equates to a grid, with two subscripts used to specify the row and column of the desired element. A three-dimensional array equates to a pile of grids and requires three subscripts. The C# language can handle arrays with a very large number of dimensions, but it is unlikely that you'll ever need to go beyond three.

Aspect ratio This is the ratio of height to width of a display screen. The first TV sets had an aspect ratio of 3:2 (that is, their screens were was 3 units wide and 2 units high). Wide-screen displays have a ratio of 16:9 (that is, the screen is 16 units wide and 9 units high). Games must be written to accommodate the possibility that they will be used with different display formats.

Assembly An assembly is used by .NET framework to bring together program code and resources that the program might need. It is created when a project is built. There are two forms of assembly: programs that can be executed (which have the file extension .exe) and libraries (which have the file extension .dll). Only program assemblies have a `Main` method, which starts the program running.

Asset An asset is any item of content that is used as part of a game. This includes sounds and images that the game requires, as well as 3-D models and any other game information. The XNA Framework provides a Content Manager, which manages the assets in a game project.

Assignment There are two parts to an assignment: the thing you want to assign and the place you want to put it. For example, consider the following:

```
int first, second, third ;
first = 1 ;
second = 2 ;
third = second + first ;
```

The program declares three variables: `first`, `second`, and `third`, each of which is of integer type. The last three statements are the ones that actually do some work. These are assignment statements. An assignment gives a value to a specified variable that must be of a compatible type. The value that is assigned is an expression. The equals sign in the middle is there mainly to confuse you; it does not mean "equals" in the numeric sense. I like to think of it as a "gozzinta." A gozzinta takes the result on the right-hand side of the assignment and drops it into the box on the left.

Bit A bit is a single "binary digit." It is the smallest unit of data that a computer can hold and has two possible states: on (1) or off (0). Bits are combined so that values larger than 1 can be represented. Each bit that you append doubles the number of possible values.

Block A block is a number of code statements that are enclosed in curly brackets. These are the characters *{* and *}* and are also known as braces. Any block can contain any number of local variables; that is, variables that are local to that block. Here's an example:

```
{
   int localToThisBlock;
   // create a variable local to
      the block
   localToThisBlock = 99;
   // OK because the variable
      exists here
}
localToThisBlock = 100;
// will cause compilation error
```

Blocks are used as the bodies of methods and in any situation where you want to lump a number of statements together so that they can be treated as a single entity, such as in an `if` condition or loop.

Boolean Boolean arithmetic deals only with values that can be true or false. A variable of type `bool` can hold a value that is true or false. Sometimes that is all you need. An example of a `bool` variable could be one that holds the state of a network connection, like this:

```
bool networkOK;
```

This variable can be set to indicate the state of the network. The results of conditions are Boolean values, and variables of type bool can be used directly in conditions:

```
if (networkOK) sendPlayerMove();
```

The preceding statement would call `sendPlayerMove` if `networkOK` was set to `true`.

Bounds (of an array) The bounds of an array is the range of possible subscripts that can be used to access elements in the array. This ranges from 0 (the element at the base of the array) to (size-1), which is the element at the end of the array.

If your program "goes outside the bounds of the array"—that is, tries to access an element with a subscript that is not in the permitted range—then it fails with an exception.

Brace The curly bracket characters (*{* and *}*) are sometimes called braces. This is perhaps a reference to the fact that they come in pairs; that is, every open bracket must be matched by a closed bracket. Braces are used to enclose statements and create blocks.

break The `break` keyword is used in looping constructions and `switch` statements to allow program execution to exit from the construct:

```
for (int i = 0; i < 10; i++)
{
   if (i == 5) break;
}
// get here when i reaches 5
```

The loop would terminate when the value of `i` reaches 5. The `break` causes execution to transfer to the statement immediately following the loop block. The `break` keyword is used in the `switch` construct to end the execution of the `switch` statement.

Breakpoint Breakpoints are used when debugging programs. They are a way of finding out what a program is doing. Within XNA Game Studio, you can mark program statements with breakpoints. In debugging mode, a program runs until it reaches (or hits) the breakpoint, at which point it pauses and returns control to you so that you can investigate the state of the program. You can then resume execution or step through statements. Note that you can set breakpoints while your program is running, even if it is running inside an Xbox, as long as you started it using debugging mode in XNA Game Studio.

Byte A byte is the smallest unit of addressable storage in a computer. It is made up of 8 bits, meaning that it can represent any one of 256 possible values, from 0 to 255.

C# "The Programming Language of Champions," I reckon.

Call When you want to use a method, you call it. When a method is called, the sequence of execution switches to that method, starting at the first statement in its body. When the end of the method, or a `return` statement, is reached, the sequence of execution returns to the caller.

Cast A cast gives an additional instruction to the compiler to force it to convert a value in a particular way. You cast a value by putting the required type in brackets before the value. For example:

```
double d = 1.7;
int i = (int) d ;
```

Because the double type has greater range and precision than an integer, the programmer must tell the compiler explicitly that the assignment is sensible. In the previous code, the message to the compiler is "I don't care that this assignment could cause information to be lost. I, as the writer of the program, take the responsibility of making sure that the program works correctly." Casting can cause data to be lost. In the code above the fractional part of d would be truncated when it is transferred leaving the value 1 in i.

char The `char` type is used to hold a single character in a program. The character can be a letter, a digit, a punctuation character, or a nonprintable character, such as the newline character. Here's an example:

```
char ch = 'A';
```

Some characters have special "control" behaviors and do not map to printable characters on the screen. They are expressed using a sequence of characters that starts with a special escape character. Escape in this context means "Escape from the normal humdrum conventions of just meaning what you are, and let's do something special." The escape character in C# is the backslash (\). Control characters and their possible escape sequences are shown in the following table.

Character	Escape Sequence
\'	Single quote
\"	Double quote
\\	Backslash
\0	Null
\a	Alert
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

The effect of these control characters depends on the device you send them to. Some systems beep when you send the alert character to them. Some clear the screen when you send the form feed character. You can use them as follows:

```
char beep = '\a' ;
```

Note that the a must be lowercase. Within Microsoft XNA, you can use the New Line escape sequence '\n' in a string to produce a control character that will cause the `DrawString` method to take a new line.

Class A class is a collection of behaviors (methods) and data (fields). Class instances are managed by reference. Declaring a variable of the type of the class creates a reference to an instance of that class. To make an instance of a class, you have to use the `new` keyword.

Cohesion *Cohesion* is a software engineering term that refers to how "together" an object is. Objects with high cohesion are self-contained and self-reliant. They contain all the data and behaviors they need to do their job and do not require the involvement of other objects. High cohesion is generally a good thing because it means that the objects can be interchanged with others more easily and alterations to the way they work internally do not affect other objects.

Comment A comment is an item that you put into your program for the humans to read and the compiler to ignore. Comments can be given in two forms, depending on how much you want to say:

```
// This is a simple comment that
   just runs to the end of
   this line

/* This is a comment in which I'm
      going to try to express the
      creative forces that drove
      me to write this program,
      which was forged in the
      smithy of my soul and for
      which all should be
      grateful.
*/
```

The first comment begins with the characters // and finishes at the end of the line. The second kind of comment begins with the /* characters and continues until the */ characters appear. Comments are a good thing; you can use them to provide useful information to someone trying to make sense of your program (or, indeed, even to yourself).

Compiler The compiler is the part of XNA Game Studio that converts the C# program that you write into instructions to be executed on the target device. It ensures that the statements that you write have the correct C# syntax and that your code is broadly sensible. The compiler produces compilation errors if it finds problems with your source code that prevent it from being able to produce an output, such as a missing semicolon or mismatched brackets or braces. The compiler also produces warnings if it detects something in the program that indicates you might have made a mistake—for example, if a variable is created but never used, or that some part of the code would never be reached when the program runs.

Component A component is a piece of software that has a particular set of behaviors that are exposed in a particular way. It can be exchanged with another component that is configured the same way. A Microsoft XNA `Game` class can be regarded as a component in that it has `Initialize`, `Draw`, and `Update` behaviors that can be used by other classes. The XNA Framework uses these behaviors when it runs your game. In this way, the XNA Framework can treat a game as a component that it is using. Components often expose their behaviors by means of an interface.

Conditional compilation This allows a programmer to "switch off" statements in the program so that the statements are compiled only if a given symbol is defined:

```
#if debug
// debug code goes here
#endif
```

The `debug` symbol is defined at the top of the program:

```
#define debug
```

If the debug symbol is not defined, the compiled program does not contain any of the statements controlled by it.

Constructor A constructor is a method in a class or structure that gets control when a new instance of the class or structure is being created. Constructor methods often accept parameters so that an instance can be given values to set it up. The `Color` structure has a number of constructor methods that accept different numbers

of parameters, depending on how the color is to be created. You used one constructor when you created the colors for the mood light, like so:

```
Color background = new Color
(redIntensity, greenIntensity,
blueIntensity);
```

When you create your own classes or structures, you can give them constructors so that they can be initialized when they are created:

```
class Player
{
   public string Name;
   public int Score;
   public Player ( string inName,
     int inScore )
   {
       Name = inName;
       Score = inScore;
   }
}

Player p = new Player("Rob", 100);
```

Once you declare a constructor for the `Player` class, the only way that an instance of the `Player` class can be created is by calling this constructor, which must initialize the `Name` and `Score` fields, typically based on values passed to the constructor.

A constructor method has the same name as the class or structure of which it is part. Once you have added one or more constructor methods, programmers must call one of the constructors to create an instance.

Content Manager The Content Manager is the component of Microsoft XNA that manages all the assets used by a particular game. It includes the tools that prepare the content when a game project is being built and is also the component that makes the content available when the game is running. The Content Manager is component-based so that it can be extended to handle new types of assets as required.

continue The `continue` keyword is used to cause the execution of a loop to return to the "top" of the loop and perform the update behavior:

```
for (int i = 0; i < 10; i++)
{
   if (i == 5) continue;
   // will never get here with
      i holding 5
}
```

In this example, the code after the conditional statement is not executed when `i` has the value 5 because the `continue` will have been performed, causing the execution to return to the top of the loop. Note that this behavior is not the same as the `break` keyword in that it does not cause the loop to be abandoned completely.

Control characters Character variables normally represent letters, digits, or symbols that can be read from a keyboard or displayed on a screen. A control character is not visible, but it has some form of control effect; for example, it takes a new line or returns the cursor to the start of a new line. A control character is expressed in a program as an escape sequence. A list of the control characters and escape sequences that can be represented in a C# program is given in the entry for *char* in this glossary.

Control expression A control expression is used in a switch to select the case to be performed.

Coupling Two objects are said to be coupled if one of them relies on the other to perform its work. This reliance means that there is a dependency between the two objects such that if one of them (the one being relied on) changes the way it works, it is necessary to make sure that the other class is not affected. As an example, you could consider a menu screen that displays settings for the player of a game. The menu screen object must read data from the game and display it. In this respect it is "coupled" to the game. If the way the game stores its data changes, the menu screen might have to change as well. However, this dependency is only one-way. Changes to the menu screen do not affect the behavior of the game. Large amounts of coupling in a system make it hard to maintain and update because time must be spent checking dependencies and making sure that a change in one object does not break others.

Creators Club See App Hub.

Debug Faults in programs are called bugs, which perhaps is a reference to an insect that was found trapped in a piece of computer hardware by Grace Hopper, one of the world's first programmers. The body of the insect was physically stopping the program from working, and she "debugged" the program by removing it. A bug is caused by a misunderstanding of the problem, a limitation in the algorithm that is intended to solve the problem, or a mistranslation when the algorithm is converted into program code. Programs are debugged by a mixture of skill, determination, and luck. You

often have to add extra statements to find out what is going on in the program when it fails. You can also use *breakpoints* to stop a running program and investigate the state of the variables in it.

Declaration A declaration is a program statement that tells the compiler about a new variable or method in your program. The new item must be given an identifier and a type. If a method is being declared, the source code must also give the method signature (the type of the method and the identifiers and types of any parameters), as well as the method body:

```
int i;    // declare an integer
             with the identifier i

int doAdd( int first, int second)
          // declare a method
{
   return first + second;
}
```

Variables can be local to a block or members of a class or structure. Local variables must be declared in a block before they can be used. Methods are members of a particular class or structure and are declared within it.

delegate A delegate is a type-safe reference to a method. A delegate is created for a particular method signature (for example, "this method accepts two integers and returns a float"). It can then refer to a method that matches that signature. Note that the delegate instance holds two items: a reference to the instance of the class which contains the method and a reference to the method itself. The fact that a delegate is an object means that it can be passed around like any other.

Delegates are used to inform event generators (things like network sessions or timers) of the method that is to be called when the event they generate takes place.

Directive A directive is a command in the source of a program that tells the compiler to do something. The `#define` directive tells the compiler that a symbol is being defined. The `using` directive tells the compiler to look in a particular namespace for objects.

Directory A directory is a place in a file store where you can store a file. It is sometimes called a folder. Directories can contain directories so that file storage can be arranged in a hierarchy. The path to a file identifies all the directories that must be traversed to get to that file. Each directory name is separated from the next by the backslash character, as in `c:\code\program\progfile.cs`.

do – while The `do – while` construction allows a program to repeat a block of code until a controlling condition at the end becomes `false`. Note that the test is performed after the statement or block; that is, even if the test is bound to fail, the statement is performed at least once, as follows:

```
do
   statement or block
while (condition) ;
```

This form of loop can be used as an alternative to the `for` loop constructions. It is very useful in programs where you want to request something, check that it is okay, and then repeat the process if it is not. There is an alternative form where the condition is tested before the statement:

```
while (condition)
    statement or block
```

In this looping construction, the statement is not performed at all if the condition is `false` at the beginning of the `while` loop.

You don't have to use these constructions if you have no need to; it's simply provided for situations where a loop is required but there is no need for a counter as would be used in a `for` loop.

Element (of an array) An element is an individual item in an array. Each element is identified by its subscript value.

Encapsulation This is the creation of objects that encapsulate a set of behaviors and data for a particular purpose. The object performs all the functions required for that purpose and can be regarded as a "black box," with no need for outsiders to actually know how it works. An example would be an `AlienSprite` object that would perform all its drawing, updating, and initialization behavior without needing the involvement of any other classes.

Enumerated type An enumerated type is one for which you specify the set of values that it can have. Here is an example:

```
enum SeaState {
     EmptySea,
```

```
        Attacked,
        Battleship,
        Cruiser,
        Submarine
    } ;

SeaState openSea ;
openSea = SeaState.EmptySea;
```

The type `SeaState` could be used to hold the state of the sea in a battleship game. It has five possible values, which are created as shown. The variable `openSea is of type SeaState` and is set to `EmptySea` in the previous code.

Exception An exception is a way that a C# program can signal that something has gone wrong when it runs. The exception itself is an object that is created when the exception is "thrown" and can be "caught" by an exception handler. The C# language provides the `try – catch` construction, which can be used to deal with exceptions that might be thrown. Your program gets exceptions if it calls things that create a bad result. For example, if a running program tries to get the Content Manager to load a nonexistent resource, the Content Manager signals its displeasure by throwing an exception. If your program doesn't catch the exception, it fails at that point.

Expression An expression is a collection of operands and operators that can be evaluated to produce a result. You have seen numeric expressions, logical expressions, and text expressions, as shown in this code example:

```
int i = 0;
i = i + 1;
// arithmetic expression adding
   1 to i
bool iIsPositive;
iIsPositive = i > 0;
// logical expression
string IValue;
IValue = "Value of i is : " +
i.ToString();  // text expression
```

Field A field is a member of a class or structure that stores data within an instance:

```
class Player
{
    public string Name;
    public int Score;
}
```

The `Player` class contains two fields: the `Name` of the player, which is a string, and the `Score` the player has reached, which is an integer. A program uses a field by giving the identifier of the instance, followed by a period (.), followed by the name of the field:

```
Player p = new Player();
p.Name = "rob";
p.Score = 100;
```

The `Name` and `Score` fields can be accessed in this way because they have been made public. Fields can also be made private, in which case they are not visible to code outside the class or structure.

File extension Files on a computer system have filenames that are used to locate them. The file extension is information on the end of the filename made up of a number of characters after a period (.) character. The Microsoft Windows operating system uses the file extension to select the application to be used to open a particular file. "Program.cs" identifies a C# program file, for example, while "Background.png" would identify a Portable Network Graphics (PNG) file.

Folder See *Directory*.

Framework A framework is a set of software resources that programmers can fit together and extend to create solutions to problems. The Microsoft .NET Framework provides a way that programs can run on a computer platform. It also provides a comprehensive set of resources that can be used to create general-purpose applications. The XNA Framework provides resources for the creation of games.

Fully qualified name A fully qualified name is one that provides a complete path to the resource that is being identified. It identifies all the namespaces in the path to the resource with that name:

```
Microsoft.Xna.Framework.Graphics.
Color background;
```

You can avoid having to use the fully qualified name of a resource by adding a `using` directive at the top of your program source. A set of `using` directives is inserted automatically into the Game1.cs file when XNA Game Studio creates a new game project.

Garbage collector The garbage collector is a process that runs as part of a .NET application and searches for and removes resources that are no longer used.

Generic method A C# method is supplied with parameters for the method to act on. In a generic method, the parameters are not restricted to one particular type. Generic methods are used when the programmer wants to create a method to perform a particular action but wants the action to be performed on variables of different types. In Microsoft XNA, the Content Manager provides a generic method called `Load`, which is supplied with the type of the item to be loaded.

```
Texture2D cheeseTexture =
   Content.Load<Texture2D>
   ("Images/Cheese");
SoundEffect cymbolTing =
   Content.Load<SoundEffect>
   ("cymbolTing");
```

The `Load` method can then perform the appropriate load action and deliver a result of the required type.

The `List` collection class also uses this mechanism so that it can manage a list of whatever type you give it.

```
public List<BaseSprite>
   GameSprites =
   new List<BaseSprite>();
```

This creates a `List` that can hold references to `BaseSprite` instances.

Header (of a method) A C# method can be broken into two parts: the block of code, which is the body of the method and contains the statements that the method performs, and the header, which indicates the type returned by the method, the identifier (which is the name of the method), and the parameters that the method accepts. Look at this code example:

```
int doAdd(int first, int second)
// declare a method
{
   return first + second;
}
```

The header of the method `doAdd` is `int doAdd(int first, int second)`.

IDE See Integrated Development Environment (IDE).

Identifier An identifier is a name chosen by the programmer to identify something in a program. This includes the names of variables and the names of classes, structures, and methods. The C# compiler has rules concerning the construction of identifiers; they can contain letters (*a–z* or *A–Z*), digits (*0–9*), and the underscore (_) character. An identifier must not start with a digit. The case of the letters in an identifier is significant in that the identifiers `count` and `Count` could both be used in the same program to refer to different variables:

```
int count;
double Count;
// legal C# - but the Great
   Programmer wouldn't approve
```

C# has a convention that local variables, parameters, and private members of a class or structure should have identifiers that start with a lowercase letter. Identifiers for classes and structures and public members of classes and structures should have identifiers that start with an uppercase letter.

Instance Instances of objects are created as a program runs. If the object is manipulated by value, there is no need to use `new` to create an instance of it, although you can use `new` if you wish to call the constructor for that type.

Integer An integer is a numerical value that has no fractional part. The C# language provides a number of integer types; the programmer should choose the type that provides the most appropriate range of values for the program being written.

Integrated Development Environment (IDE) An Integrated Development Environment (IDE) combines an editor for creating the source code, a compiler, and a debugger in a single tool that can be used for development. XNA Game Studio is based on the Microsoft Visual Studio IDE.

Intellisense Intellisense is the name given to the feature of XNA Game Studio that provides context-sensitive help and suggestions to you as you write your program source. The system constantly monitors what you are typing and suggests appropriate items on the basis of what it sees.

Interface An interface defines a set of actions. The actions are defined in terms of a number of method definitions. A class that implements an interface must contain code for each of the

methods defined in the interface. Look at this code:

```
interface ISinger
{
   void SingSong(int loudNess);
}


class OperaSinger : ISinger
{
   public void SingSong(int
loudNess)
   {
   }

   public void SingAria
      (int loudness, int vibrato)
   {
   }
}

class PoliceMan : ISinger
{
   public void SingSong
      (int loudNess)
   {
   }

   public void MakeArrest ()
   {
   }
}

ISinger singer = new PoliceMan();
```

The interface `ISinger` contains a single method called `SingSong`, which is supplied with a parameter to indicate how loudly the song is to be sung. Both `OperaSinger` and `PoliceMan` implement the interface, meaning that either of them can be asked to sing (although you might get a better tune out of `OperaSinger`). This means that you can regard `PoliceMan` and `OperaSinger` in terms of their singing ability, even though they are completely different classes. A reference to `ISinger` could be made to refer to either an `OperaSinger` or a `PoliceMan` interface and ask it to sing by calling the `SingSong` method. Viewing classes in terms of what they can do is a large part of component-based development.

Keyword A keyword is a word that is part of the C# language. Keywords that you have seen include `for`, `if`, `new`, `class`, `struct`, `switch`, and `case`. Keywords have a particular meaning, and you cannot create an identifier that is the same as a keyword. The XNA Game Studio editor displays keywords in bright blue.

Literal A literal is something in a program that is literally just there. Examples of literals include values in expressions and strings:

```
width = width + 2;
playerName = "rob";
```

In the preceding statements, the literals are the value "rob" and the value 2.

Local A variable that is local to a block is declared in the block and is discarded when the execution of the program leaves that block. Local variables are used in the situation where you want a variable for a very short part of the program.

Localization Localization is the name for the process of making a program work in a manner appropriate to a particular part of the world. It includes aspects such as the language used for the user interface, the character set, and how dates, times, and currency values are displayed.

Logical Logical values can be either true or false. C# provides the `bool` type to hold logical values, it provides comparison operators (for example, *LESS THAN*) that compare values and produce logical results, and it provides logical operators (for example, *OR*) that allow logical values to be combined.

Machine code This is a generic term for low-level instructions that can be processed by a computer. This is in contrast with source code, which contains the program instructions that are written by the programmer and that contain a high-level description of a solution to a problem in a form that can be read by humans. The compiler takes the high-level source code and converts this into a form that is eventually made into machine code for execution on a target device.

Member A member of a class is declared within that class. It can either do something (if it is a method) or hold some data (if it is a variable). Methods are sometimes called *behaviors*. Data members are sometimes called *fields*.

Method A method is a block of code preceded by a method header. The method has a particular identifier and may return a value. It may also accept one or more parameters to work on. Methods are used to break a large program up

into a number of smaller units, each of which performs one part of the task. They are also used to allow the same piece of program to be used in lots of places in a large development. If a method is public, it can be called by code in other classes. An object exposes its behaviors by using a public method.

Microsoft XNA The best game development environment in the world, bar none.

Modifier A modifier is used to modify a declaration. It gives the compiler additional information about the thing that is being declared. Examples of modifiers are `public`, `private`, and `static`.

Namespace A namespace is a way of categorizing related resources. Each resource provided by a framework must have a unique name. Putting all the names at the same level would result in confusion; for example, the name `Device` could have many possible meanings—you might want to have audio devices, graphics devices, and so on. A namespace is a space where particular names have meaning. You could create a `Graphics` namespace and an `Audio` namespace, each of which could hold a `Device` resource:

```
namespace Graphics
{
   class Device
   {
   }
}

namespace Audio
{
   class Device
   {
   }
}
```

You would refer to the devices created by this code as `Graphics.Device` and `Audio.Device`. It is possible for a namespace to contain a namespace, allowing a hierarchy of names to be created. A particular source file can contain a number of different namespaces, and a namespace can be spread over several source files.

The XNA Framework is organized into a series of namespaces, each of which holds a set of related resources. You can access them by using a particular namespace or by giving a fully qualified name.

Narrowing Narrowing can occur when a variable of one type is assigned to another. C# provides a number of different data types that are used to hold values in programs. Each type has a particular range and precision. For example, the byte type can hold values in the range 0 to 255, whereas an integer can hold values in the range –2,147,483,648 to 2,147,483,647. Narrowing would occur if a program assigned a value from an integer variable into a byte. If the integer had a value greater than 255, the narrowing would result in the corruption of the value. The C# compiler insists the programmer use a cast to confirm that a narrowing operation is valid.

null The C# keyword `null` allows a program to express the fact that a reference points nowhere. Newly created reference variables are set automatically to refer to `null`, and it is possible to test for this condition in your programs as follows:

```
Player p;
if ( p == null )
{
   // will get here because p is
      initially null
}
```

You can actually assign the value `null` to a reference to indicate that the reference is not set to refer anywhere.

Object An object is an instance of a given data type. Many types are provided by C# and Microsoft XNA, and you can create your own types by declaring classes (`class`) and structures (`struct`).

Operand An operand is something that is worked on in an expression by an operator. Operands are either literals, variables, or expressions.

Operator An operator is used in an expression and identifies an operation to be performed on one or more operands. Arithmetic operators that you have seen include plus (+), minus (–) , multiply (*) and divide (/). Relational operators include less than (<), greater than (>), equals (==) and not equals (*!*=). Logical operators that you have seen include logical *AND (&&)* and logical OR *(||)*.

Overflow Overflow occurs if the capacity of a variable is exceeded when a program is running. Variables are declared as being of a particular type, and the programmer must be careful to use the type appropriately. For example, the byte type is able to hold values that range from 0 to 255. If a program put 255 into a byte variable and then added 1 to this variable, the result would cause the variable to overflow because the byte type is not able to represent that value. While some forms of program error, such as exceeding the array bounds, cause an exception to be thrown, this is not always the case with overflow.

Overload A method is overloaded when a method with the same name but a different set of parameters is declared in the same class. Methods are overloaded when there is more than one way of providing information for a particular action; for example, a date can be set by providing day, month, and year information, or by a text string or by a single integer that is the number of days since January 1. Three different overloaded methods could be provided to set the date, each with the name `SetDate`. In that case, the `SetDate` method could be said to have been overloaded.

Override Sometimes you may want to make a more specialized version of an existing class. This may entail providing updated versions of methods in the class. You do this by creating a child class that extends the parent and then overriding the methods that need to be changed. When the method is called on instances of the child class, the new method is called, not the overridden one in the parent. You can use the `base` keyword to get access to the overridden method if necessary.

Parameter A parameter is supplied by a call to a method to give the method something to work on. A parameter that is a value type is passed into the method by value. A parameter that is a reference type is passed into the method by the value of the reference. If you want to pass a value type by reference, you must mark the parameter as a reference type using the `ref` qualifier. A special kind of reference that can only be used to deliver a result (that is, you can't follow the reference to read the thing it refers to) can be specified by using the `out` qualifier.

Pixel A pixel, or "picture element," gives the color of a single small area of the display screen. The more pixels that a screen contains, the higher the quality of the picture, but more memory will be used, and it will take longer to create an image.

Precision C# provides several types that can hold numbers with fractional parts, and these types have different precisions. The precision of a type determines how accurately that type can represent a particular value. Because computer storage is finite, the precision to which numbers are stored is limited. The float type can represent values to a precision of 7 digits, whereas the double type provides 15 to 16 digits of precision.

Private A `private` member of a class is visible to code only in methods inside that class. It is conventional to make data members of a class private so that they cannot be changed by code outside the class. The programmer then can provide methods or C# properties to manage the values that can be assigned to the private members. The only reason for not making a data member `private` is to remove the performance hit of using a method to access the data.

Program A program is a description of a solution to a problem. The program sets out the steps to be taken and decisions to be made and that ultimately are to be performed by some sort of computer hardware.

Programming language A programming language is a special form of language that has a simple and unambiguous syntax and grammar. It is designed so that programs written in the language can be converted easily into forms that can be executed by computer hardware.

Project A project is a collection of program files and other resources that can be brought together to produce a single assembly that can be deployed as part of a solution to a problem. XNA Game Studio manages projects and also brings a number of projects together to create a single solution.

Property Properties are extremely useful and make your code a lot cleaner. Essentially, you can have code like the following:

```
x.Width = 99;
```

This looks like an assignment to a member of a class, but it can be much more than that and can result in additional code running. The `Width` property could be managed like this:

```
class ThingWithWidth {
  private int widthValue;
  public int Width
  {
     get
     {
        return widthValue;
     }

     set
     {
        widthValue = value;
     }
  }
}
```

When a program performs the assignment to the property, the set portion (the "setter") runs. The `value` keyword is set to the value of the incoming property. This `set` code performs a simple assignment to the `widthValue` data member, but you could validate the value and throw an exception if you don't like it. I've decoupled the name of the property value from the value of the property (one convention is to put the word `value` on the end of the name of the internal value). Of course, you don't actually have to have a value inside the class; you could calculate a result rather than return a member.

When setting the value, you can run additional code whenever the value of your property changes. This makes creating state machines easy. Furthermore, you don't have to implement both a `get` and a `set` behavior; you can have just one so that you can create write-only (or read-only) properties. You can have lots of getters for the same property; perhaps you would like to read the speed in kilometers per hour as well as miles per hour.

The only downside is that you must be aware that substantial amounts of code can run when you perform innocent-looking assignments.

Protected A protected member of a class is visible to methods in the class and to methods in classes that extend this class. It is kind of a halfway house between private (no access to methods outside this class) and public (everyone has access). It lets you designate members in parent classes as being visible in the child classes.

public A public member of a class is visible to methods outside the class. It is conventional to make the method members of a class public so that they can be used by code in other classes. A public method is how a class provides services to other classes.

Range The range of a given type sets out the largest and smallest values that can be held in a variable of that type. Each C# type has a particular range, and one of the tasks for programmers is to select a type with a range that is appropriate for the data they wish to store.

Reference A reference is like a tag that can be attached to an instance of a class and has a particular name. C# uses a reference to find its way to an instance of the class and use its methods and data:

```
class Player
{
   public string Name;
   public int Score;
}

Player p = new Player();
p.Score = 100;
```

The variable `p` is a reference variable that can refer to instances of the class `Player`. It is set to refer to a new `Player` instance. The reference is then used to access the `Score` field inside the instance referred to by `p`.

One reference can be assigned to another. If you do this, the result is that there are now two references that refer to a single object in memory. In C#, references are type-safe in that a reference to one particular object, such as a `Texture2D`, would not be allowed to refer to any other type of texture. This means that when the reference is followed to an object, the actions performed with that object are always appropriate.

SDK See Software Development Kit (SDK).

Signature A given C# method has a particular signature that allows it to be identified uniquely in a program. The signature is the name of the method and the type and order of the parameters to that method:

- `void Silly(int a, int b)` has the signature of the name `Silly` and two `int` parameters.

- `void Silly(float a, int b)` has the signature of the name `Silly` and a `float` parameter followed by an integer parameter.

This means that the code

```
Silly(1, 2) ;
```

would call the first method, whereas

```
Silly(1.0f, 2) ;
```

would call the second.

Note that in C#, the return type of the method is not part the method signature.

Software Development Kit (SDK) A Software Development Kit (SDK) is a collection of tools and library resources that can be used to create software on a particular platform.

Solution XNA Game Studio brings together one or more project files to produce a single solution. The same project file can be used in more than one solution, which allows libraries of code to be created and reused. Within a solution, one of the projects is designated the startup project and will be the one that runs when the system produced by the solution is started.

Source code Source code is the text written by programmers. It is stored in plain text on the development computer and converted by a compiler into the machine code that actually performs the program instructions on the target machine.

State At any given instant, a running program is in a particular state. Many game programs contain variables that explicitly manage the state of items in the game. It is often the case that an enumerated type is created to represent a particular state.

Statement A statement is a single action that a program performs. Statements in C# programs are separated by the semicolon (;) character.

static In the context of C#, the `static` keyword makes a member of a class part of a class rather than part of an instance of the class. This means that you don't need to create an instance of a class to use a static member. It also means that static members are accessed by means of the name of their class rather than a reference to an instance. Static members are useful for creating class members that are to be shared with all the instances, such as currency conversion rates for all the accounts in a bank.

string The `string` data type lets programs work with strings of text. The string is held as a one-dimensional array of characters. Strings can be used with the + operator, which cause them to be concatenated together. String literals are denoted in a program enclosed in double quotes. A string literal can contain control characters; see the *Char entry* in this glossary for details of these. Here's an example of strings in code:

```
string firstname = "Rob";
string surname = "Miles";
string fullname =
   firstname + " " + surname;
```

Structure A structure is a collection of data items. It is managed by value, not by reference, and `struct` contents are copied on assignment:

```
struct Particle
{
    public int X;
    public int Y;
}

Particle position;
position.X = 99;
position.Y = 00;
Particle[] Smoke = new
Particle[1000];
```

The `Particle` structure simply holds the `X` and `Y` positions of a particle. Because it is a `struct`, I can declare a variable of type `Particle`, and an instance is created automatically. The `Smoke` array, which contains 1,000 particles, is also created automatically. There is no need to use `new` to create any `Particle` instance.

Structures are also passed by value into methods. Structures are useful for holding a simple set of related data in a single unit. They are not as flexible as objects (which are managed by reference), but they can be more efficient to use because accessing structure items does not require a reference to be followed in the same way as for an object. An array of `struct` values is stored in a single block of memory that contains a row of the items. An array of items managed by reference (for example, instances of a class) is stored as an array of references, with

each element in the array able to refer to one instance.

Subscript A subscript is a value that is used to identify the element in an array. It must be an integer value. Subscripts in C# always start at 0 (this identifies the initial element of the array) and extend up to the size of the array minus 1. This means that if you create a four-element array, you get hold of elements in the array by subscript values of 0, 1, 2, or 3. The best way to regard a subscript is that it is the distance down the array that you are going to move to get the element that you want. This means that the first element in the array must have a subscript value of 0.

switch The `switch` construction allows a program to select one option from several based on a control expression. Switches are often used to select particular behavior based on the value of an enumerated type. Here's an example:

```
switch (state)
{
   case GameState.titleScreen:
       drawTitle ();
       break;
   case GameState.playingGame:
       drawGame();
       break;
   case GameState.highScoreDisplay:
       drawHighScore();
       break;
   default:
       doShowError();
       break;
}
```

The `switch` construction uses the value of the control expression to decide which option to perform. It executes the case that matches the value of the control expression. The `break` statement after the call of the relevant method is used to stop the program running on and performing the code that immediately follows. In the same way as you break out of a loop, when the break is reached, the switch is finished and the program continues running at the statement after the switch.

Another useful feature is the `default` option, which gives the switch somewhere to go if the `switch` value doesn't match any of the cases available.

this The `this` keyword means "a reference to the current instance." Its use is implied within methods in classes:

```
class Player
{
   public string Name;
   public int Score;

   public void IncreaseScore ()
   {
      this.Score = this.Score + 1;
   }
}
```

It would be possible to write `Score` rather than `this.Score` in the `IncreaseScore` method because the compiler inserts `this.` automatically if required.

The `this` reference can also be used to pass an instance as a parameter in a call to another method:

```
DisplayScore(this);
```

The `DisplayScore` method accepts a reference to a `Player` as a parameter. It can be called from a method in the `Player` class to display the score of that player instance.

Type In C#, all data items have a particular type associated with them. Some types are built into the C# language. These types, such as `int`, `float`, and `bool`, are available to all programs written in the language. Other types can be added from libraries, such as `DateTime`. Finally, you can create your own types to hold a collection of data and behaviors that are specific to the problem at hand.

The C# compiler ensures that whenever variables of different types are used together, there is no potential for errors to occur or data to be lost. For example, an attempt to move a value from a variable of floating-point type into an integer results in the compiler generating an error unless programmers use a cast to indicate that they are aware of the issue, and in this context, the action is valid. Type checking is performed at compile time (which is called *static type checking*) and also when the program runs. This means that even if the programmer uses a cast to force one thing to be used as another, at run time any inappropriate mixing of types would be rejected. This extra stage makes C# programs much safer, but the

extra run-time type checking slows down the program.

Type-safe We have seen that C# is quite fussy about combining things that should not be combined. Try to put a `float` value into an `int` variable, and the compiler rejects the code. The reason for this is that the designers of the language have noticed a few common programming mistakes and have designed for these mistakes to be detected before the program runs, not afterwards when it has crashed. One of these mistakes is to use values or items in contexts where it is either not meaningful to do so (such as putting a `string` into a `bool`) or where doing so could result in losing data or accuracy (such as putting a `double` into a `byte`). This kind of fussiness is called type safety, and C# is very big on it. Some other languages are much more relaxed when it comes to combining things, working on the assumption that the programmer knows best. They assume that just because code has been written to do something, that thing must be the right thing. But C# is not that way; and neither am I. I think it is important that developers get all the help they can to stop them doing stupid things, and a language that stops you from combining things in a way that might not be sensible is a good thing in my book.

Of course, if you really want to impose your will on the compiler and force it to compile your code in spite of any type-safety issues, you can do this by using casting.

using The word `using` can serve as either a compiler directive or a keyword in a program.

C# provides the `using` directive, which you can use as follows:

```
using Audio;
```

The `using` directive must appear at the start of a source file. It identifies a namespace that is to be used to resolve the names of classes in that file. If the `Audio` namespace contains a class called `Device`, I could add a `using` directive to my program so that I can create instances of `Device` without having to add `Audio` to qualify the name.

I can still use other `Device` classes, such as `Graphics.Device`, but I need to give its fully qualified name. You can add multiple `using` directives at the start of a source file; when you create a new project, you often find that a number of them have been added automatically. If there is a name clash (for example, you use two namespaces that each contain a class called `Device`), the compiler requires you to use the fully qualified name for that particular class. It can also be sensible to use the fully qualified name in circumstances where you want a reader of the program source to identify easily where a class is defined.

C# provides the `using` keyword, which lets you state precisely where in a program a variable is being used:

```
using (PongGame game = new
PongGame())
{
    game.Run();
}
```

The `using` keyword is followed by the declaration of a variable to be used in the block which follows the `using` statement. When the block is complete, the garbage collector knows the variable is no longer required and can be removed. Without the `using` statement, the Garbage Collector would have to deduce that there were no remaining references to the variable `game` in the preceding code.

Value type A value type holds a simple value. Value types are passed as values into method calls, and their values are copied on assignment; that is, `x = y` causes the value in `y` to be copied into `x`. Subsequent changes to the value in `x` do not affect the value of `y`. Note that this is in contrast to reference types, where the result of this assignment would be that `x` and `y` refer to the same instance.

Variable A variable holds a value that is being used by a program. A given variable has a unique identifier and is declared as having a particular type. Variables can be local to a block or they can be members of a class.

Virtual method I can call a method (a member of a class) to do a job. Sometimes I may want to extend a class to produce a child class that is a more specialized version of that class. In that case, I may want to replace (override) the method in the parent with a new one in the child class. For this to take place, the method in the parent class must have been marked as `virtual`. Only virtual methods can be overridden. Making a method virtual slightly slows down access to it because the program must look for any overrides of the method before calling it. This is why not all methods are made virtual initially.

void A `void` method performs a task but does not return a value. A programmer who wants to create a method that does not return a value can tell the compiler this by making the type of the method `void`.

while The `while` keyword is used in looping constructions which are described in the *do – while* item in this glossary.

Widening Widening is the reverse of narrowing. When a value is widened, it is moved from a type with a narrower range and precision into one that has a wider range, such as from the `byte` type (with a range of 0 to 255) into an integer type (with a range of 2,147,483,648 to 2,147,483,647). The compiler is quite able to produce code that performs this conversion because there is no chance of data being lost.

Workspace A workspace is analogous to an XNA Game Studio solution in that it contains programming resources and projects that are used to create a solution.

Xbox Live Xbox Live is a networking solution for Xbox 360 and PC games. Gamers pay a subscription that gives them an identity on the Xbox Live network and allows them to engage in network play using Xbox games. They can also download game demos and other content that is then stored on the hard disk of their Xbox 360. An Xbox Live account is required if you wish to join the App Hub.

XNA See Microsoft XNA.