.Introduction to Object Orientated Development

Contents

[Contents 1](#_Toc492905927)

[Hierarchy of Computer Systems 3](#_Toc492905928)

[Source Code 3](#_Toc492905929)

[ASCII: American Standard Code for Information Interchange 3](#_Toc492905930)

[Binary 4](#_Toc492905931)

[Executable Files 5](#_Toc492905932)

[XML: Extensible Markup Language 5](#_Toc492905933)

[Application Development Software 5](#_Toc492905934)

[Variables 6](#_Toc492905935)

[Declaring Variables 6](#_Toc492905936)

[Variables Vary 6](#_Toc492905937)

[Variable Types 7](#_Toc492905938)

[Types You’ll Use All The Time 7](#_Toc492905939)

[Types For Whole Numbers 7](#_Toc492905940)

[Types For Really Huge and Really Tiny Numbers 7](#_Toc492905941)

[Primitive Types 8](#_Toc492905942)

[Reference Types 8](#_Toc492905943)

[Classes 9](#_Toc492905944)

[You Can Use a Class to Build an Object 9](#_Toc492905945)

[An Object Gets Its Methods from Its Class 9](#_Toc492905946)

[Base Class 9](#_Toc492905947)

[Writing Classes? 10](#_Toc492905948)

[Encapsulation 12](#_Toc492905949)

[Access Modifiers 12](#_Toc492905950)

[Encapsulation Example 12](#_Toc492905951)

[Advantages of Encapsulation 13](#_Toc492905952)

[Constructors 13](#_Toc492905953)

[Example of Constructor 13](#_Toc492905954)

[Example of Constructor That Accepts Parameters 14](#_Toc492905955)

[Properties 15](#_Toc492905956)

[Get Methods 15](#_Toc492905957)

[Getter Example 15](#_Toc492905958)

[Set Methods 16](#_Toc492905959)

[Setter Example 16](#_Toc492905960)

[The “This” Keyword 18](#_Toc492905961)

[“This” Example 18](#_Toc492905962)

[The “Static” Keyword 19](#_Toc492905963)

[Static Member Example 19](#_Toc492905964)

[Inheritance 21](#_Toc492905965)

[Inheritance Example 22](#_Toc492905966)

[Polymorphism 23](#_Toc492905967)

[Polymorphism Example 23](#_Toc492905968)

[Polymorphic Arrays 24](#_Toc492905969)

[Abstraction 25](#_Toc492905970)

[Interfaces 26](#_Toc492905971)

[Use the Interface Keyword to Define an Interface 26](#_Toc492905972)

[IComparable Interface 26](#_Toc492905973)

[Access Modifiers 27](#_Toc492905974)

[An Abstract Class is like a cross between a class and an interface 28](#_Toc492905975)

See AS Software Fact File for more detail.

## Hierarchy of Computer Systems

* CPU (Central Processing Unit) – the brains of the computer where most calculations take place. The CPU, along with the other physical components of the motherboard is known as the hardware.
* OS (Operating System) – Directly on top of hardware. Provides an interface between the user and the hardware. Contains System Software which is low-level software concerned with more technical aspects of setting up and running a computer system.
* Application Software – Higher-level software enables the user to accomplish tasks such as word processing and creating databases.

## Source Code

Source code is any collection human-readable programming language (usually ordinary text). It may include comments. Source code can be written in a text editor such as Notepad++ or in an Integrated Development Environment (IDE). An IDE is a software application which is used to write and modify program code. IDEs usually consist of a source code editor, build automation tools and a debugger (identifies errors).

## ASCII: American Standard Code for Information Interchange

Computers can only understand numbers. An ASCII code is the numerical representation of a character such as ‘a’ or ‘A’. ASCII defines a 7-bit (bit is short for binary digit) digital code for every character on the standard computer keyboard.

ASCII has 128[[1]](#footnote-1) (because 2⁷ = 128) possible encoding, ranging from 0 to 127, and the first 32 of these are non-letter characters such as backspace. This means other characters such as the Euro symbol cannot be represented using ASCII. More modern and flexible encoding include Unicode.

## Binary

Computers use binary to store data. A binary digit or a bit is the smallest unit of data in computing. Bits can be grouped together to make them easier to work with.

|  |  |
| --- | --- |
| Nibble | 4 bits |
| Byte | 8 bits |
| Kilobyte (KB) | 1024 bytes |
| Megabyte (MB) | 1024 kilobytes |
| Gigabyte (GB) | 1024 megabytes |
| Terabyte (TB) | 1024 gigabytes |

In everyday life the counting system known as base 10 is used (also called denary).

It is called base 10 because there are 10 choices of digit between 0 and 9. For the binary numbers there are only two possible digits: 0 or 1. The binary system is also known as base 2.

Another counting system is the hexadecimal system which uses the digits 0 to 9 followed by the letters A to F.

|  |  |  |
| --- | --- | --- |
| **Denary** | **Binary** | **Hexadecimal** |
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 10 | 2 |
| 3 | 11 | 3 |
| 4 | 100 | 4 |
| 5 | 101 | 5 |
| 6 | 110 | 6 |
| 7 | 111 | 7 |
| 8 | 1000 | 8 |
| 9 | 1001 | 9 |
| 10 | 1010 | A |
| 11 | 1011 | B |
| 12 | 1100 | C |
| 13 | 1101 | D |
| 14 | 1110 | E |
| 15 | 1111 | F |

## Executable Files

Also known as binary files. After source code is written and stored to the hard disc drive (HDD), it can be compiled into a file (executable file) that has the program instructions in a form which can be understood by the CPU. These files are no longer legible for the programmer. They contain a series of operation codes for CPU instructions and the memory locations of data.

## XML: Extensible Markup Language

XML is a more structured way to store data. It was created as a standard way to store and transport data. It is closely related to HTML (Hypertext Markup Language). Both XML and HTML are intuitively readable, however HTML was designed to display data and XML was designed to carry and store data.

XML does not process data or perform any calculations on the data. It is a standardized envelope to store data and is the most common mechanism for data transfer between software system.

## Application Development Software

* Software Systems Design – the process of developing software systems. It includes research, prototyping, testing, modification, re-engineering and maintenance of software. With many high level languages, these activities take place within an Integrated Development Environment (IDE)
* IDE – a program that provides comprehensive facilities to a programmer for software development. Most IDEs include a source code editor with syntax error highlighting features, compilation automation tools and debugging aids.

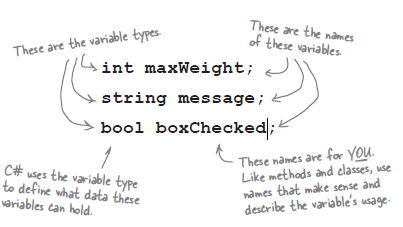
See AS Defining Data Fact File for more detail.

# Variables

Every program is basically a data cruncher. The data can be in different forms; an image or a text document. But it’s still data. A variable is what a program uses to store data.

## Declaring Variables

When you declare a variable, you tell your program its type and its name. Once C# knows your variable’s type, it will keep your program from compiling if you make a mistake and try to do something that doesn’t make sense, like subtracting “Fido” from 53993. Variables declared in methods are called local variables.



## Variables Vary

While your program runs, a variable can be equal to different values at different times. The value ***varies***.

If your program sets the variable myHeight equal to 63:

int myHeight = 63;

any time myHeight appears in the code, C# will replace it with its value, 63. Then later on if you change its value to 12:

myHeight = 12;

C# will replace with myHeight 12 – but the variable is still called myHeight.

## Variable Types

A variable’s type determines what kind of data it can store. There are a bunch of types built into C# and each one stores a different kind of data.

### Types You’ll Use All The Time

Int, string, bool and double are the most common types.

* int can store any whole number from –2,147,483,648 to 2,147,483,647.
* string can hold text of any length (including the empty string "").
* bool is a Boolean value—it’s either true or false.

double can store real numbers from ±5.0 × 10−324 to ±1.7 × 10308 with up to 16 significant figures

Another common type is char which stores a single character. Char values are written inside single quotes ‘ ‘. You can include escape sequences in singles quotes too (‘\n’ is a line break

### Types For Whole Numbers

Most of the time, if you need to store a whole number you can just use int. But sometimes you need something bigger, or smaller. C# gives you other options for storing whole numbers:

* byte can store any whole number between 0 and 255.
* sbyte can store any whole number from –128 to 127.
* short can store any whole number from –32,768 to 32,767.
* ushort can store any whole number from 0 to 65,535.
* uint can store any whole number from 0 to 4,294,967,295.
* long can store any whole number between minus and plus 9 billion billion.
* ulong can store any whole number between 0 and about 18 billion billion.

The u in ushort, uint, and ulong stands for unsigned. It can’t be negative.

The s in sbyte stands for signed which means it can be negative. The sign is a minus sign.

### Types For Really Huge and Really Tiny Numbers

* float can store any number from ±1.5 × 10–45 to ±3.4 × 1038 with 7 significant digits.
* double can store any number from ±5.0 × 10–324 to ±1.7 × 10308 with 15-16 significant digits.
* decimal can store any number from ±1.0 × 10–28 to ±7.9 × 1028 with 28–29 significant digits.

When your program needs to deal with currency, you usually want to use a decimal to store the number.

## Primitive Types

Primitive types (also called value types) are data types provided by a language implicitly. They are passed to method by passing an exact copy of the data.

When instantiated, a single space in memory is allocated to store the value and the runtime deals directly with the data in memory.

Primitive types include, int, bool, char, float, decimal, long and short.

## Reference Types

Common reference types are strings and arrays. An object is a reference type. Reference types store the address of their data rather than the data itself. They reference where the data is held in memory.

When your code needs to work with an object in memory, it uses a reference, which is a variable whose type is a class of the object it’s going to point to. A reference is like a label that your code uses to talk about a specific object.

# Classes

A class is a set of related fields and methods detailing how to construct on object. An object is an instance of a class. Many objects can be instantiated from a class. An object is an instance of a class. Many objects can be instantiated from a single class.

A class is a construct that enables you to create your own custom (objects) types by grouping together variables of other types, methods and events. A class is like a blueprint. It defines the data and behavior of a type.

## You Can Use a Class to Build an Object

A class is like a blueprint for an object. If you wanted to build five identical houses in a suburban housing development, you wouldn’t ask an architect to draw up five identical sets of blueprints. You’d just use one blueprint to build five houses.

## An Object Gets Its Methods from Its Class

Once you build a class, you can create as many objects as you want from it using the new statement. When you do, every method in your class becomes part of the object.

When you create a new object from a class, it’s called an instance of that class. When you are designing a Windows Form Application in the Visual Studio IDE, you use the toolbox. Everything in the toolbox is a class. There is a Button class, a TextBox class, a Label class etc. When you drag a button out of the toolbox, the IDE automatically creates an instance of the Button class and calls it button1. When you drag another button out of the toolbox, it creates another instance called button2. Each instance of Button has its own properties and methods. But every button acts exactly the same way, because they’re all instances of the same class.

## Base Class

A basic class can be referred to as base, super or parent. This class can be built upon to give a refined class referred to as derived, sub or child.

A derived class is a specialization of the base class. For example, if you have a base class Animal, you might have one derived class that is named Mammal and another derived class that is named Reptile. A Mammal is an Animal, and a Reptile is an Animal, but each derived class represents different specializations of the base class.

Derived class may have access to some or all, of the base attributes depending on the visibility assigned. This is known as inheritance and will be covered in more detail later. C# only supports single inheritance. This means that a derived class can only inherit from one base class. However, many derived classes can inherit from the same base class.

## Writing Classes?

Remember, a class defines a data type for objects, but it is not an object itself. An object is a concrete entity based on a class, and is sometimes referred to as an instance of a class.

A class definition starts with the keyword **class** followed by the class name. The class body will contain the fields, properties, methods, and events of the class. Class names should always begin with a capital letter.

class Customer

{

//Fields, properties, methods and events go here...

}

The code above declares a class call Customer.

class Customer

{

//Fields

string name;

int age;

//method

public void customerDetails()

{

Console.WriteLine(“Customer Name: ” + name);

Console.WriteLine(“Customer Age: ” + age);

}

}

The code above declares a class called Customer, which has the fields **name** and **age** as well as a **customerDetails** method that will display customer details to the screen.

Access modifiers can be used for fields and methods of classes. They are keywords used to specify the accessibility of fields and methods. A method can also be referred to as a function or a member.

A method that has been defined as **public** can be accessed from outside the class, as long as it is declared within the scope of the class object (the curly brackets). That is why the **customerDetails** method is declared **public**, as it will be called from outside the class.

Now the Customer class is defined, an object of that type can be instantiated in the Main. The **new** operator instantiates an object and returns a reference to its location.

static void Main(string[] args)

{

Customer customer = new Customer();

customer.customerDetails();

}

Declaring fields and methods as public allows them to be accessed using the **dot** operator.

class Customer

{

//Fields

public string name;

public int age;

//method

public void customerDetails()

{

Console.WriteLine(“Customer Name: ” + name);

Console.WriteLine(“Customer Age: ” + age);

}

}

static void Main(string[] args)

{

Customer customer = new Customer();

customer.name = “Ciara”;

customer.age = 19;

Console.WriteLine(customer.name);

//outputs Ciara

}

# Encapsulation

Encapsulation is the idea of “surrounding” an entity, to keep what’s inside it together and to protect it.

In programming, encapsulation allows us to restrict access to the inner workings of a class using access modifiers. An access modifier defines the scope and visibility of a class member (a field or a method).

In the context of C#, encapsulation refers to an object’s ability to hide data and behavior that are not necessary to the user. It enables a group of fields, methods and other members to be considered a single unit or object. Encapsulation is also known as information hiding.

## Access Modifiers

* public – access is the most permissive. There are no restrictions on accessing public members.
* private – access is the least permissive. Private members are only accessible within the body of the class in which they are declared.
* protected – only allows access that extend (inherit) from that class.

## Encapsulation Example

If a field is declared as private, it can only be accessed inside the class – the fields are hidden within the class. Access to the fields is via public methods where modifications can be controlled and fields can be set as read-only or write-only where suitable.

class Customer

{

//Fields

private double balance = 100;

//methods

public double getBalance()

{

return balance;

}

public double deposit(double n)

{

balance += n;

balance = balance + n;

}

public double withdraw(double n)

{

balance -= n;

}

}

In the above code, encapsulation has been used to hide the balance member from the outside code. Restricted access to the balance member has been provided using a public method. The class data can be read through the getBalance method and modified only through the deposit and withdraw methods.

The user cannot directly access the balance variable but it can view its value by calling the getBalance method. This helps maintain data integrity.

Encapsulation prevents the functionality and data being randomly accessed by code outside of the class. Access to data is tightly controlled. Encapsulation gives maintainability, flexibility, and extensibility to code.

## Advantages of Encapsulation

* The fields of a class can be set to read-only or write-only where appropriate.
* A class can have total control over what is stored in its fields (the user has no direct access)
* The users of a class do not know how the class stores its data. A class can change the data type of a field, and users of the class do not have to change any of their code.
* Protects data from accidental corruption.

# Constructors

A constructor is a special member method of a class that is called (executed) whenever a new object if that class is instantiated (created).

A constructor has exactly the same name as its class, is public and does not have any return type.

## Example of Constructor

class Person

{

//Fields

private string name;

//Constructor

public Person()

{

Console.WriteLine(“Hi”);

}

}

static void Main(string[] args)

{

Person person = new Person();

//outputs Hi

}

Upon the creation of an object if type Person, the constructor is automatically called.

Constructors can be used to set initial values for instance variables by using parameters. In other words, an initial value can be assigned to an object when it is created.

## Example of Constructor That Accepts Parameters

class Person

{

//Fields

private string name;

//Constructor

public Person(string Name)

{

name = Name;

}

public string getName()

{

return name;

}

}

Remember: The Person class is defined, an object of that type can be instantiated in the Main. The **new** operator instantiates an object and returns a reference to its location.

When the object is created, we can pass a parameter that will be assigned to the **name** variable.

static void Main(string[] args)

{

Person person = new Person(“John Doe”);

Console.WriteLine(person.getName());

//outputs John Doe

}

Constructors can be **overloaded** like any other method by using a different number of parameters. Method overloading will be discussed in more detail later.

# Properties

It is good practice to encapsulate members of a class and provide access to them only through public methods.

A property is a member that provides a flexible mechanism to read, write or compute the value of a private field. They can be used as if they are public data members.

## Get Methods

Get methods are also known as getters or accessor methods. A get method is used to return the value of an instance variable.

## Getter Example

class Student

{

//Fields

private string name;

private int age;

//Constructor

public Student(string Name, int Age)

{

name = Name;

age = Age;

}

//Get Method for name field

public string getName()

{

return name;

}

//Get Method for age field

public int getAge()

{

return age;

}

}

The above code shows two get method, one for the **name** variable and one for the **age** variable.

Getters should always be public as they will be accessed from outside the class. They should have a return type as they are being used to return a value. The return type should be the same as the return type for the instance variable they are returning.

static void Main(string[] args)

{

Student student1 = new Student(“John Doe”, 21);

Student student2 = new Student(“Jane Doe”, 23);

Console.WriteLine(“Student 1 Name: “ + student1.getName());

Console.WriteLine(“Student 2 Age: “ + student2.getAge());

//outputs

//John Doe

//23

}

In the main method two instances of the Student object have been created and two parameters have been passed to the constructor of each object.

The name of student 1 is returned by calling the student1.getName() method and the age of student 2 is returned by calling the student2.getAge() method.

## Set Methods

Set methods are also known as setters or mutator methods. A set method is used to set the value of an instance variable.

## Setter Example

class Dog

{

//Fields

private string name;

//Constructor

public Dog(string Name)

{

name = Name;

}

//Get Method for name field

public string getName()

{

return name;

}

//Set Method for Name field

public void setName(string Name)

{

name = Name;

}

}

Setters should always be public because they will be accessed from outside the class. They are declared as void because they don’t return a value. They accept a parameter and its value will be assigned to the instance variable of the object.

static void Main(string[] args)

{

Dog dog = new Dog(“Spot”);

Console.WriteLine(“Dog Name: “ + dog.getName());

//outputs

//Spot

dog.setName(“Fido”);

Console.WriteLine(“Dog Name: “ + dog.getName());

//outputs

//Fido

}

In the main method an instance of the Dog object has been created and a parameter has been passed to the constructor and assigned to the **name** variable of the object.

The value of the name variable is returned by calling the dog.getName() method. The value of the **name** variable is the changed by passing the string “Fido” to the dog.setName() method.

The value of the name variable is returned by calling the dog.getName() method and the new value is displayed.

# The “This” Keyword

The “this” keyword refers to the current instance of the class. It used for other things as well but the purpose of the course it refers to the current instance of the class.

It can be used in the constructor of a class.

## “This” Example

class Student

{

//Fields

private string name;

private int age;

//Constructor

public Student(string name, int age)

{

this.name = name;

this.age = age;

}

//Get Method for name field

public string getName()

{

return this.name;

}

//Get Method for age field

public int getAge()

{

return this.age;

}

//Set Method for name field

public string setName(string name)

{

this.name = name;

}

//Set Method for age field

public string setAge(int age)

{

this.age = age;

}

}

In the above code this.name and this.age refers to the instance variables name and age. They are called instance variables because they belong to an instance of a class. Remember, C# assigns from right to left, so variables being passed into the constructor are being assigned to the instance variables of the class. The get methods will return the value of the instance variables belonging to a Student object. The set methods accepts a parameter and assign the value to the instance variable of an object.

# The “Static” Keyword

A static variable is a variable that has a had a memory location allocated for the entire lifetime of the runoff the program. This is in contrast to a local variable in a class which id=s allocated when the class is first called. Static variables can be allocated and declared without declaring an instance of a class. Static methods use the same principle.

static void Main(string[] args)

{

}

The static keyword can be seen in the declaration of the main method. The main method is the entry point of any C# program. For example, in a Windows Form Application, the main method tells C# which form to call when the program is run.

The “static” keyword or modifier is used to declare a static member, which belongs to the type itself rather than a specific object.

The static modifier can be used with classes, fields, methods, properties, operators, events, and constructors.

Static methods can only access static members. As the main method is static, the methods that it calls must also be static.

static void Main(string[] args)

{

Dog.bark();

}

class Dog

{

public static void Bark()

{

Console.WriteLine("Woof");

}

}

## Static Member Example

An example of a static class, is C#’s Math class. The Math class provides constants (which are implicitly static) and static methods for trigonometric, logarithmic, and other common mathematical functions.

public static class Math

Because the Math class is declared as static, whenever we want to use it, we don’t have to declare a new instance of the class. It is a global class and can be accessed by using the class name.

Say you’ve been tasked to write a method which can be used to calculate the square root of a number. The method would need to accept one value, the number to find the square root of.

public static double getSquareRoot(double number)

{

Double squareRoot;

squareRoot = Math.Sqrt(number);

return squareRoot;

}

The Math class contains the static method Sqrt(double) which accepts a double parameter.

# Inheritance

Inheritance enables you to create new classes that reuse, extend, and modify the behaviour that is defined in other classes. The class whose members are inherited is called the base class, and the class that inherits those members is called the derived class. A derived class can have only one direct base class. However, inheritance is transitive. For instance, if A is larger than B, and B is larger than C, then A is larger than C. So, if Class B inherits from Class A, and Class C inherits from class B, then Class C also inherit from Class A. (AS factfile)

Conceptually, a derived class is a specialization of the base class. For example, if you have a base class Animal, you might have one derived class that is named Mammal and another derived class that is named Reptile. A Mammal is an Animal, and a Reptile is an Animal, but each derived class represents different specializations of the base class.

When you define a class to derive from another class, the derived class implicitly gains all the members of the base class, except for its constructors and destructors. The derived class can thereby reuse the code in the base class without having to re-implement it. In the derived class, you can add more members. In this manner, the derived class extends the functionality of the base class. (msdn)

Inheritance allows you to define a class based on another class. This makes creating and maintaining an application easy. The class whose properties are inherited by another class is called the **base** class. The class which inherits the properties is called the **derived** class.

For example, base class **Animal,** can be used to derive **Cat** and **Dog** classes. The derived class inherits all the features from the base class, and can have its own additional features.

|  |  |
| --- | --- |
| Base Class  Base class features | Derived Class (inherited from base class)  Base class features  Derived class features |

## Inheritance Example

class Animal

{

//Fields

private int legs {get; set;}

private int age {get; set;}

}

The Animal class above has been defined with two integer variables; legs and age.

class Dog : Animal

{

public Dog()

{

legs = 4;

}

public void Bark()

{

Console.WriteLine(“Woof”);

}

}

The Dog class above is derived from Animal. In C#, a colon : is used to show that class is inheriting from another.

All public member of Animal have become public members of Dog which is why the **legs** variable can be accessed in the Dog constructor.

In the main method, an object of type Dog can be instantiated and the inherited members can be accessed.

static void Main(string[] args)

{

Dog dog = new Dog();

Console.WriteLine(dog.Legs);

//outputs

//4

dog.Bark();

//outputs Woof

}

# Polymorphism

It allows you to invoke sub class methods through a super class reference during run-time. This is implicit in java and achieved in C# using ‘virtual’ in the super class and ‘override’ in the sub class. (AS factfile).

The word polymorphism means “having many forms”. It occurs when there is a hierarchy of classes that are related through inheritance from a common base class. Polymorphism means that a call to a member method will cause a different implementation to be executed depending on the type of object that invokes the method. (A single method can have several different implementations).

## Polymorphism Example

Consider a program that will allow the user to draw different shapes. Each shape is drawn differently and you do not know which shape the user will choose. Polymorphism can be used in invoke the appropriate Draw() method by overriding the same method in the base class. Methods like this must be declared using the **virtual** keyword.

class Shape

{

public virtual void Draw(){

Console.WriteLine(“Base Draw”);

}

}

The **virtual** keyword allows method to be overridden in the derived classes. (It lets you work with groups of related objects in a uniform way)

Different shape classes can now be defined that define their own Draw method using the **override** keyword.

class Circle : Shape

{

public override void Draw(){

Console.WriteLine(“Circle Draw”);

}

}

class Rectangle : Shape

{

public override void Draw(){

Console.WriteLine(“Rectangle Draw”);

}

}

The virtual Draw method in the Shape base class can be overridden in the derived classes. Circle and Rectangle have their own Draw methods. Separate Shape objects can be created for each derived type and their Draw methods can be called.

static void Main(string[] args)

{

Shape c = new Circle();

c.Draw();

//outputs Circle Draw

Shape r = new Rectangle();

r.Draw();

//outputs Rectangle Draw

}

## Polymorphic Arrays

An array, of the super type, can be formed of objects of various sub class types and invoke each of their relevant methods. i.e. if they all share the same method name, the appropriate method of each object can be invoked.

For example, if we have super class **Employee** and sub classes **Full-Time**  and **Part-Time** which both inherit from **Employee**.

An array of employees can be declared like this: (this is in the AS Objects Factfile)

Employee[] emps = new Employee[20];

emps[0] = new ContractEmployee( 1000, “John”, “Green”, 18, 45.75);

emps[1] = new ContractEmployee( 1000, “Mary”, “Brown”, 6, 150.00);

emps[2] = new FullTimeEmployee( 1000, “Peter”, “Smith”, 43275.00);

emps[3] = new ContractEmployee( 2004, "Anna", "Doherty", 10, 50.00);

emps[4] = new FullTimeEmployee( 1002, "John", "Moore", 35000.00);

emps[5] = new FullTimeEmployee( 1003, "Richard", "Lowe", 27500.00);

\*please refer to the AS Objects Factfile to see what the classes for these objects look like.

# Abstraction

An abstract class is one that cannot be instantiated but serves as a class definition for derivation. Members that are defined as abstract within an abstract class must be implemented by a subclass. The use of abstract classes allows designers to ensure that derived classes will be properly defined.

Previously you learned that polymorphism is used when you have different derived classes with the same method, which has different implementations in each class. Sometimes, there is no need for virtual methods to have a separate definition in the base class. These methods are defined using the **abstract** keyword and specify that the derived classes must define the method on their own.

You cannot create objects of a class containing and abstract method, which is why the class itself should be abstract.

An abstract method can be used in the Shape class from polymorphism.

abstract class Shape

{

public abstract void Draw();

}

The Draw method is now abstract and has no body, or curly brackets – just a semi colon. The shape class must also be declared as abstract because it contains an abstract method. Abstract method declarations are only permitted in abstract classes.

An abstract class is intended to be a base class of other objects. It acts as a template for derived classes. The derived classes can be defined with their own Draw methods.

class Circle : Shape

{

public override void Draw(){

Console.WriteLine(“Circle Draw”);

}

}

class Rectangle : Shape

{

public override void Draw(){

Console.WriteLine(“Rectangle Draw”);

}

}

* An abstract class cannot be instantiated
* An abstract class may contain abstract method and accessors
* A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors..

static void Main(string[] args)

{

Shape c = new Circle();

c.Draw(); //outputs Circle Draw

Shape r = new Rectangle();

r.Draw(); //outputs Rectangle Draw

}

# Interfaces

Sometimes you need to group objects together based on the things they can do rather than the classes they inherit from. That is where interfaces come in – they you work with any class that can do the job. Any class that implements an interface must promise to fulfil all its obligations – if it doesn’t, the compiler will throw an error.

## Use the Interface Keyword to Define an Interface

Adding an interface to your program is a lot like adding a class, except you never write any methods. You just define the methods return type and parameters, instead of block statements inside curly brackets you just close the line with a semicolon.

Interfaces do not store data, so you can’t declare any fields. But you can add definitions for properties. Get and set accessors are just methods, and interfaces are all about forcing classes to have certain methods with specific names, types and parameters.

Whenever you create an interface, you should make its name start with an uppercase I. It’s not a rule but it makes your code a lot easier to understand.

Everything in a public interface is automatically public, because you’ll use it to define the public methods and properties of any class that implements it.

## An Abstract Class is like a cross between a class and an interface

Suppose you need something like an interface, that requires classes to implement certain methods and properties. But you need to include some code in that interface, so that certain methods don’t have to be implemented in each inheriting class. What you want is an abstract class. You get the features of an interface, but you can write code in it like a normal class.

## IComparable Interface

The IComparable interface only has one method. The CompareTo(Object). This method compares the current instance with another object of the same type and returns an integer that indicates whether the current object precedes, follows or occurs in the same position in the sort order as the other object.

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| Less than zero (-1) | The current instance precedes the object specified by the CompareTo method in the sort order. |
| Zero | This current instance occurs in the same position in the sort order as the object specified by the CompareTo method. |
| Greater than zero (+1) | This current instance follows the object specified by the CompareTo method in the sort order. |

Imagine an object Accommodation. It needs to be sorted by its attribute, rent. A simple implementation of the CompareTo method looks like this:

public int CompareTo(Object obj){

Accommodation other = obj as Accomodation;

return this.rent.CompareTo(other.rent);

}

Imagine an array of type Accommodation called accArray. The code to sort and print its contents are below. The array is sorted in ascending order by default.

Array.Sort(accArray);

Console.WriteLine("\n\tProperties in order of rent charged\n\n”);

for (int x = 0; x < accArray.Length; x++)

{

Console.WriteLine(accArray[x].ToString());

}

# Access Modifiers

When you change an access modifier on a property, field, or method of a class—its members—or the entire class, you change the way other classes can access it.

* **Public means that anyone can access it**. When you mark a class or any class member public, you’re telling C# that any instance of any other class can access it. It’s the least restrictive access modifier. Only mark things as public if there is a good reason – this is how we make sure things are well encapsulated.
* **Private means that only other members cab accessed**. A private class member can only be accessed from other members inside that class or other instances of that class. You can’t mark a class as private unless that class lives inside another class., in which case, it’s only available to instances of its container class.
* **Protected means public to subclasses, private to everyone else**. A subclass can’t access the private fields in its base class – it has to use the base keyword. Sometimes it is convenient if the subclass can access the private fields of the base. Any class member marked as protected can be accessed by any other member of its class, and any member of a subclass of its class.
* **Internal means public only to other classes in an assembly**.
* **Sealed says that this class can’t be subclassed**. (sealed is a modifier – not an access modifier. It only affects inheritance – it doesn’t change how the class can be accessed.)

1. Fact file says 127 [↑](#footnote-ref-1)