

C# Fundamentals

Brian G. Patrick

C# is an object-oriented (OO)
programming language

But what does that mean?

It means that ...

the C# programming language supports the
three fundamental tenets
of the object-oriented paradigm.

Object-Oriented Tenets

- **Encapsulation**
 - Bundles together data members (attributes) and methods (operations) within a single syntactic unit.
 - Data members define representation; methods define behaviour.
 - This syntactic unit is called a **class** in C#.
 - Each class defines a **reference type**.
 - Each instance of a class is an **object** of that class.

Class = Data Members (what) + Methods (how)

Object-Oriented Tenets

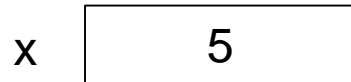
- **Inheritance**
 - Permits a (descendant or child) class to reuse, extend and modify the attributes and behaviour of its parent class.
- **Polymorphism and Dynamic Binding**
 - Permits an object of a parent class to dynamically (at runtime) take on the behaviour of a descendant class.

Value vs Reference Types

Value types

- Include the predefined simple types in C# such as int, float, char, and bool (and their variations).
- A variable of a value type directly accesses its value in memory.

```
int x = 5;
```

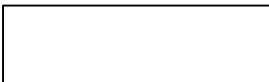


Value vs Reference Types

Reference types

- Include all types (classes, interfaces, arrays) whose variables indirectly access its associated object in memory.
- A variable of a reference type **refers** to an object in memory.

```
string s;
```

s 

```
s = new string("Brian");
```

s 

Let's look at each tenet in turn

Encapsulation

using fractions as an example

What is a fraction?

The Numerator

The top half of the **fraction**. The number of parts you have.

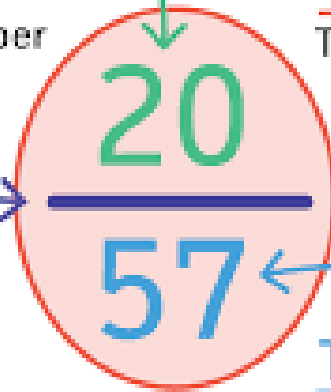
The Fraction

The whole thing!

The line that separates the **numerator** and the **denominator** (doesn't have a name).

The Denominator

The bottom half of the **fraction**. The number of parts in a whole.



Declare a class

```
public class Fraction  
{  
    ...  
}
```

- **public** is an access modifier which places no restrictions on access to a class, data member, method etc. Hence, the type Fraction is publicly available.

Data Members

```
public class Fraction
{
    private int numerator;        // Data member or field
    private int denominator;      // Data member or field
    ...
}
```

- **private** is an access modifier which restricts access to a data member, method etc. to within the class itself. Hence, numerator and denominator can only be accessed within the class Fraction.

Properties

```
public class Fraction
{
    ...
    private int numerator;
    public int Numerator
    // Read/write property associated with the data member numerator
    {
        get { return numerator; }
        set { numerator = value; }
    }
    // or simply
    // public int Numerator { get; set; }
    ...
}
```

Used as:

```
int x = f.Numerator; // get
f.Numerator = 6;     // set
```

```
public class Fraction
{
    ...
    private int numerator;
    public int Numerator
    // Read only property associated with the data member numerator
    {
        get { return numerator; }
    }
    // or simply
    // public int Numerator { get; }
    ...
}
```

Used as:

```
int x = f.Numerator; // get
f.Numerator = 6;      // Error: Read-only
```

```
public class Fraction
{
    ...
    private int denominator;
    public int Denominator
    {
        get { return denominator; }
        set { if (value != 0)
                denominator = value;
            else
                throw new ArgumentException("Denominator is zero");
            }
    } ...
}
```

Used as:

| | |
|------------------------|---------------------|
| int y = f.Denominator; | // get |
| f.Denominator = 0; | // Exception thrown |

Constructors

A **constructor**

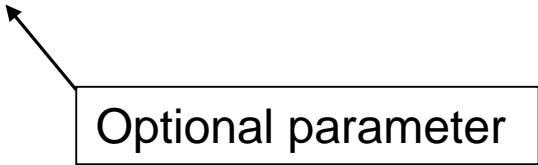
- is a special method that shares the same name of the class.
- is called implicitly whenever an object of that class is created using the operator **new**.
- does **not** have a return parameter.

Note: A class may have any number of constructors. Each one must be differentiated based on its parameter list.


```
public class Fraction
{
    ...
    public Fraction ( )           // Constructor 1
    {
        Numerator = 0;
        Denominator = 1;
    } ...
}
```

Used as: Fraction g = new Fraction(); // 0 / 1

```
public class Fraction
{
    ...
    public Fraction (int numerator, int denominator=1)    // Constructor 2
    {
        this.numerator = numerator;
        this.denominator = denominator;
        // What could throw an exception?
    } ...
}
```



Used as:

```
Fraction f = new Fraction(4,-3);    // 4 / -3
Fraction g = new Fraction(6);       // 6 / 1
```

The **this** keyword refers to the current instance of the class. It is used to differentiate between parameters and data members with the same name. It is also used to call one constructor from another within the same class.

Methods

```
public class Fraction
{
    ...
    public Fraction Multiply (Fraction f)
    {
        return new Fraction (Numerator * f.Numerator,
                               Denominator * f.Denominator);
    } ...
}
```

Used as:

```
Fraction u = new Fraction (1,2);
Fraction v = new Fraction (-3,4);
Fraction w = u.Multiply(v);           // (1 * -3) / (2 * 4)
                                       // -3 / 8
```

Static Methods

```
public class Fraction
{
    ...
    static public Fraction Multiply (Fraction f1, Fraction f2)
    {
        return new Fraction ( f1.Numerator * f2.Numerator,
                               f1.Denominator * f2.Denominator);
    } ...
}
```

A **static** method belongs to the class itself and cannot be invoked by an instance of that class. It is invoked instead using the class name.

Used as:

```
Fraction u = new Fraction (1,2);
Fraction v = new Fraction (-3,4);
Fraction w = Fraction.Multiply(u,v);
```

Overloaded Operators

```
public class Fraction
{
    ...
    public static Fraction operator * (Fraction f1, Fraction f2)
    {
        return new Fraction ( f1.Numerator * f2.Numerator,
                               f1.Denominator * f2.Denominator) ;
    } ...
}
```

operator * redefines the * operator for instances of Fraction.

Used as:

```
Fraction u = new Fraction (1,2);
Fraction v = new Fraction (-3,4);
Fraction w = u * v;
```

Inheritance

Inheritance

Inheritance permits one class (called the child class) to reuse, extend and modify the behaviour of a parent class.

In C#, a class can only inherit directly from one parent class.

For example:

```
public class Parent  
{ ... }
```

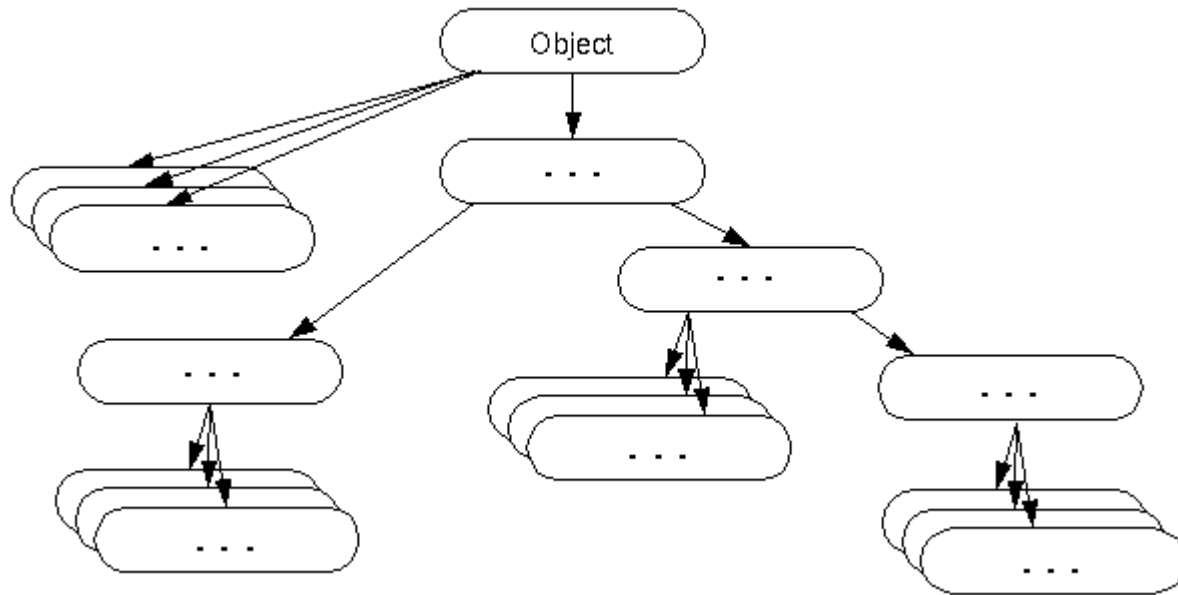
```
public class Child : Parent    // Child class inherits from Parent class  
{ ... }
```

Inheritance implies a
hierarchy of types.

So, is there a common ancestor class for all classes?

Yes, the Object Class

The **Object** class is the root of the type hierarchy in C# from which all classes **implicitly** derive.



Of the four Object methods below, all are **public** and **virtual** except MemberwiseClone whose access modifier is **protected**.

- A virtual method can be overridden (re-implemented) by a descendant class.
- A protected data member or method of a class is only visible to the class itself or its descendants.

```
public class Object
{
    ...
    public virtual bool Equals (Object obj) { ... }
    public virtual int GetHashCode ( ) { ... }
    protected Object MemberwiseClone ( ) { ... }
    public virtual string ToString ( ) { ... }
    ...
}
```

Equals (from Object class)

```
public class Fraction : Object    // Not necessary since
                                   // Object is implicitly inherited
{
    ...
    public override bool Equals (Object obj)
    {
        Fraction f = obj as Fraction;
        if (f != null)
            return Numerator * f.Denominator == f.Numerator * Denominator;
        else
            throw new ArgumentException ("Cannot compare");
    } ...
}
```

ToString (from Object class)

```
public class Fraction
{
    ...
    public override String ToString()
    {
        String sign = ""; // empty string
        if (Numerator == 0) return "0";
        if (Numerator * Denominator < 0) sign = "-";
        if (Denominator == 1)
            return sign + Math.Abs(Numerator);    // Static method
        else
            return sign + Math.Abs(Numerator) + "/" + Math.Abs(Denominator);
    }
}
```

Interfaces

An **interface** resembles a class, but

- it contains no data members,
- all of its methods are implicitly **public** (accessible), **abstract** (no body), and **virtual** (can be overridden), and
- no object can be instantiated from an interface type.

Using good programming practice,

- the name of an interface begins with “I” and
- each interface defines only a few methods.

A class:

- can inherit multiple interfaces and
- must implement all methods defined in the interface(s) it inherits.

Let's look at
two standard interfaces in C#

Comparable

Cloneable

First, Comparable

The Comparable interface has one method called **CompareTo** which, when implemented by the derived class:

- returns a negative integer if the current object is less than obj,
- returns 0 if the current object is equal to obj, and
- returns a positive integer if the current object is greater than obj.

```
public interface Comparable
{
    int CompareTo (Object obj);
}
```

```
public class Fraction : IComparable
{
    ...
    public int CompareTo (Object obj)
    {
        Fraction f = obj as Fraction;
        if ( f != null )
            // Cross multiply
            return Numerator*f.Denominator - Denominator*f.Numerator;
        else
            throw new ArgumentException ("Cannot compare");
    }
}
```

Can you find the bug? Hint: It is not syntactic.

Now, ICloneable

The ICloneable also has one method called **Clone** which, when implemented by the derived class, creates and returns a copy of the current object.

```
public interface ICloneable
{
    Object Clone ( );
}
```

```
public class Fraction : ICloneable
{
    private int numerator;
    private int denominator;
    ...
    public Object Clone ( )
    {
        return MemberwiseClone();    // from Object class
    } ...
}
```

Used as:

```
Fraction f1 = new Fraction(-3,7);
Fraction f2 = (Fraction) f1.Clone(); // Downcast
```

ICloneable?

Suppose a new class called Complex (for complex numbers) uses two fractions to define its real and imaginary parts.

```
public class Complex : ICloneable
{
    private Fraction real;           // real and imaginary are
    private Fraction imaginary;      // reference types (Fraction)
    ...
    public Object Clone ()
    {
        return MemberwiseClone();  // What gets cloned?
    } ...
}
```

Use as:

```
Complex y = (Complex)x.Clone( );    // Only a shallow copy is made
```

ICloneable!

```
public class Complex : ICloneable
{
    private Fraction real;
    private Fraction imaginary;
    ...
    public Object Clone ()
    {
        Complex c = (Complex) MemberwiseClone();
        c.real = (Fraction) real.Clone();
        c.imaginary = (Fraction) imaginary.Clone();
        return c;
    }
}
```

Used as:

```
Complex y = (Complex)x.Clone( );    // A deep copy is made
```

Generics

```
public class Complex<T> : ICloneable
{
    private T real;
    private T imaginary;
    ...
    public Complex (T n1, T n2)
    { ... }
    ...
}
```

What should go here?

Used as:

```
Complex<Fraction> c1 = new Complex<Fraction>( ... );
Complex<double> c2  = new Complex<double>(5.2,3.3);
```

Constrained Generics

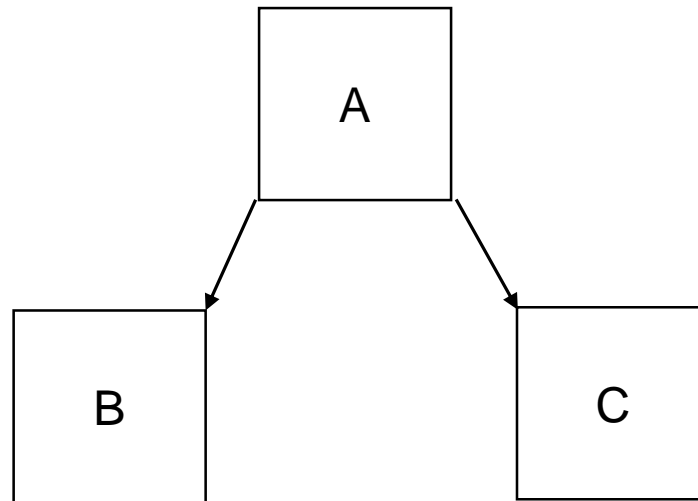
```
public class Complex<T> : ICloneable where T : ICloneable
{
    private T real;
    private T imaginary;
    ...
    public Complex (T n1, T n2)
    { ... }
    ...
}
```

Polymorphism and Dynamic Binding

Polymorphism and Dynamic Binding

Suppose classes B and C inherit from A. Each class has:

- a simple constructor that does nothing except create an object and
- one method that prints the name of its class.



A, B and C

```
class A {  
    public A ( ) { }  
    public virtual void Print ( ) { Console.Write("A") ; }  
}  
class B : A {  
    public B ( ) { }  
    public override void Print ( ) { Console.Write("B") };  
}  
class C : A {  
    public C ( ) { }  
    public override void Print ( ) { Console.Write("C") };  
}
```

Suppose a main program declares three variables a, b, and c of types A, B and C, respectively. Only objects of B and C are created using new. However, a reference variable of A (the parent class) can be assigned references to objects of either B or C as shown below. When the Print method is invoked by a, it is **dynamically bound** at runtime to the Print method of either B or C. The variable a is considered **polymorphic** since it can take on the behaviour of any descendant type.

```
A a;  
B b = new B( );  
C c = new C( );  
...  
a = b;  
a.Print( );           // "B" is printed  
a = c;  
a.Print();           // "C" is printed
```

Polymorphism and dynamic binding are powerful tools.

Since a polymorphic variable can take on different behaviors, a programmer can both reduce and reuse code. **Green computing!**

For example, consider a method *F* that has a polymorphic parameter (of a parent type). Depending what is passed to the parameter at runtime, the function *F* can adapt to the behaviour of the underlying descendant type.

```
public void F (A x)
{
    x.Print();
}
```

Exercises

1. Study Euclid's algorithm for finding the greatest common divisor of two integers.
2. Implement the Complex class where the real and imaginary parts are Fractions. Demonstrate the difference between a shallow and deep copy.
3. Study the Object class in C#.
4. What is meant by an abstract class? How does it differ from an interface?
5. Implement a simple parent class P and a child class C that inherits from P.
 - a) If P has a private data member x, can C access x?
 - b) Suppose C adds a public method F that is not part of P. When an instance c of C is assigned to a variable p of P (i.e. $p = c$), can p then call F?
 - c) How can an instance p of P be assigned to an instance c of C (i.e. $c = p$)?
 - d) If C overrides a method in P, can it change its access modifier?