Inheritance

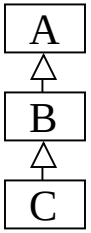
Syntax


```
class A {// base class
    int a;
    public A() {...}
    public void F() {...}
}

class B : A {                    // subclass (inherits from A, extends A)
    int b;
    public B() {...}
    public void G() {...}
}
```

- B inherits *a* and *F*(), it adds *b* and *G*()
  - constructors are not inherited
  - inherited methods can be overridden (see later)
- Single inheritance: a class can only inherit from one base class, but it can implement multiple interfaces.
- A class can only inherit from a class, not from a struct.
- Structs cannot inherit from another type, but they can implement multiple interfaces.
- A class without explicit base class inherits from *object*.

Assignments and Type Checks

```
class A {...}
class B : A {...}
class C: B {...}
```



Assignments

```
A a = new A();      // static type of a: the type specified in the declaration (here A)
                    // dynamic type of a: the type of the object in a (here also A)
a = new B();        // dynamic type of a is B
a = new C();        // dynamic type of a is C

B b = a;            // forbidden; compilation error
```

Run-time type checks

```
a = new C();
if (a is C) ...     // true, if the dynamic type of a is C or a subclass; otherwise false
if (a is B) ...     // true
if (a is A) ...     // true, but warning because it makes no sense

a = null;
if (a is C) ...     // false: if a == null, a is T always returns false
```

Checked Type Casts

Cast

```
A a = new C();
B b = (B) a;        // if (a is B) static type(a) is B in this expression; else exception
C c = (C) a;

a = null;
c = (C) a;          // ok ➔ null can be casted to any reference type
```

as

```
A a = new C();
B b = a as B;       // if (a is B) b = (B)a; else b = null;
C c = a as C;

a = null;
c = a as C;         // c == null
```

Overriding Methods

Only methods that are declared as virtual can be overridden in subclasses

```
class A {
    public          void F() {...}   // cannot be overridden
    public virtual  void G() {...}   // can be overridden in a subclass
}
```

Overriding methods must be declared as override

```
class B : A {
    public          void F() {...}   // warning: hides inherited F() ➔ use new
    public          void G() {...}   // warning: hides inherited G() ➔ use new
    public override void G() {       // ok: overrides inherited G
        ... base.G();                // calls inherited G()
    }
}
```

- Method signatures must be identical
  - same number and types of parameters (including function type)
  - same visibility (public, protected, ...).
- Properties and indexers can also be overridden (virtual, override).
- Static methods cannot be overridden.

Dynamic Binding (simplified)


```
class A {
    public virtual void WhoAreYou() { Console.WriteLine("I am an A"); }
}

class B : A {
    public override void WhoAreYou() { Console.WriteLine("I am a B"); }
}
```

A message invokes the method belonging to the dynamic type of the receiver
(not quite true, see later)

```
A a = new B();
a.WhoAreYou();                  // "I am a B"
```

Benefit: every method that can work with *A* can also work with *B*

```
void Use (A x) {
    x.WhoAreYou();
}

Use(new A());       // "I am an A"
Use(new B());       // "I am a B"
```

Hiding

Members can be declared as new in a subclass.

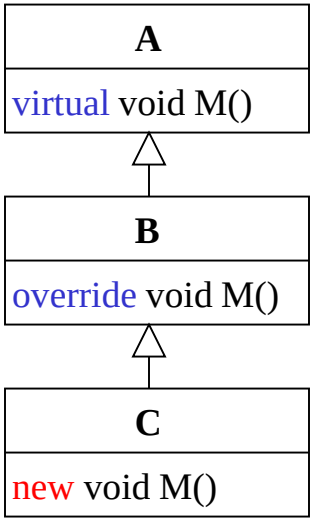
They *hide* inherited members with the same name and signature.

```
class A {
    public int x;
    public void F() {...}
    public virtual void G() {...}
}

class B : A {
    public new int x;
    public new void F() {...}
    public new void G() {...}
}

B b = new B();
b.x = ...;                    // accesses B.x
b.F(); ... b.G();             // calls B.F and B.G

((A)b).x = ...;               // accesses A.x!
((A)b).F(); ... ((A)b).G();   // calls A.F and A.G!
```

Dynamic Binding (with Hiding)

Method resolution for obj.M()

```
st = static type of obj;
dt = dynamic type of obj;
m = Method "M" of st;
for (all types t between st (exclusive) and dt (inclusive)) {
    if (t has an overriding method "M") m = "M" of t;
    else if (t has a non-overriding method "M") break;
}
call m;
```




```
A obj = new C();
obj.M();
```

Works as expected for simple cases

```
class Animal {
    public virtual void WhoAreYou() { Console.WriteLine("I am an animal"); }
}
class Dog : Animal {
    public override void WhoAreYou() { Console.WriteLine("I am a dog"); }
}

Animal pet = new Dog();
pet.WhoAreYou();          // "I am a dog"
```

A More Complex Example


```
class Animal {
    public virtual void WhoAreYou() { Console.WriteLine("I am an animal"); }
}
class Dog : Animal {
    public override void WhoAreYou() { Console.WriteLine("I am a dog"); }
}
class Beagle : Dog {
    public new virtual void WhoAreYou() { Console.WriteLine("I am a beagle"); }
}
class AmericanBeagle : Beagle {
    public override void WhoAreYou() { Console.WriteLine("I am an american beagle"); }
}

Beagle pet = new AmericanBeagle();
pet.WhoAreYou();                        // "I am an american beagle"

Animal pet = new AmericanBeagle();
pet.WhoAreYou();                        // "I am a dog" !!
```

Fragile Base Class Problem

Initial situation

```
class LibraryClass {
    public void CleanUp() { ... }
}
```

```
class MyClass : LibraryClass {
    public void Delete() { ... erase the hard disk ... }
}
```

Later: vendor ships new version of *LibraryClass*

```
class LibraryClass {
    string name;
    public virtual void Delete() { name = null; ... }
    public void CleanUp() { Delete(); ... }
}
```

```
class MyClass : LibraryClass {
    public void Delete() { ... erase the hard disk ... }
}
```

In Java the call *myObj.CleanUp()* would erase the hard disk!

- In C# nothing happens, as long as *MyClass* is not recompiled. *MyClass* still relies on the old version of *LibraryClass* (Versioning)
  ➔ old *CleanUp()* does not call *LibraryClass.Delete()*.
- If *MyClass* is recompiled, the compiler forces *Delete* to be declared as *new* or *override*.

Constructors and Inheritance

Implicit call of the base class constructor			Explicit call

class A {
  ...
}
 	
class B : A {
  public B(int x) {...}
}
	class A {
	  public A() {...}
	}
	
	class B : A {
	  public B(int x) {...}
	}
		class A {
		  public A(int x) {...}
		}
		
		class B : A {
		  public B(int x) {...}
		}
			class A {
			  public A(int x) {...}
			}
			
			class B : A {
			  public B(int x)
			  : base(x) {...}
			}

B b = new B(3);	B b = new B(3);	B b = new B(3);	B b = new B(3);
OK - Default constr. A() - B(int x)	**OK** - A() - B(int x)	**Error!** - no explicit call of the A() constructor - default constr. A() does not exist	**OK** - A(int x) - B(int x)

Visibility protected and internal

protected	Visible in the declaring class and its subclasses (more restricive than in Java)
internal	Visible in the declaring assembly (see later)
protected internal	Visible in declaring class, its subclasses and the declaring assembly

Example

```
class Stack {
    protected int[] values = new int[32];
    protected int top = -1;
    public void Push(int x) {...}
    public int Pop() {...}
}
class BetterStack : Stack {
    public bool Contains(int x) {
        foreach (int y in values) if (x == y) return true;
        return false;
    }
}
class Client {
    Stack s = new Stack();
    ... s.values[0] ...     // illegal: compilation error
}
```

Abstract Classes

Example

```
abstract class Stream {
    public abstract void Write(char ch);
    public void WriteString(string s) { foreach (char ch in s) Write(s); }
}

class File : Stream {
    public override void Write(char ch) {... write ch to disk ...}
}
```

Note

- Abstract methods do not have an implementation.
- Abstract methods are implicitly *virtual*.
- If a class has abstract methods (declared or inherited) it must be *abstract* itself.
- One cannot create objects of an abstract class..

Abstract Properties and Indexers

Example

```
abstract class Sequence {
    public abstract void Add(object x);                 // method
    public abstract string Name { get; }                // property
    public abstract object this [int i] { get; set; }   // indexer
}

class List : Sequence {
    public override void Add(object x) {...}
    public override string Name { get {...} }
    public override object this [int i] { get {...} set {...} }
}
```

Note

- Overridden indexers and properties must have the same get and set methods as in the base class

Sealed Classes

Example

```
sealed class Account : Asset {
    long val;
    public void Deposit (long x) { ... }
    public void Withdraw (long x) { ... }
    ...
}
```

Note

- *sealed* classes cannot be extended (same as *final* classes in Java), but they can inherit from other classes.
- *override* methods can be declared as *sealed* individually.
- Reason:
  - Security (avoids inadvertent modification of the class semantics)
  - Efficiency (methods can possibly be called using static binding)

Class System.Object

Topmost base class of all other classes

```
class Object {
    protected object     MemberwiseClone() {...}
    public Type          GetType() {...}
    public virtual bool  Equals (object o) {...}
    public virtual string ToString() {...}
    public virtual int   GetHashCode() {...}
}
```

Directly usable:

Type t = x.GetType(); returns a type descriptor (for reflection)

object copy = x.MemberwiseClone(); does a shallow copy (this method is protected!)

Overridable in subclasses:

x.Equals(y)	should compare the values of *x* and *y*
x.ToString()	should return a string representation of *x*
int code = x.getHashCode();	should return a hash code for *x*

Example *(for using object)*


```
class Fraction {
    int x, y;
    public Fraction(int x, int y) { this.x = x; this.y = y; }
    ...
    public override string ToString() { return String.Format("{0}/{1}", x, y); }
    public override bool Equals(object o) { Fraction f = (Fraction)o; return f.x == x && f.y == y; }
    public override int GetHashCode() { return x ^ y; }
    public Fraction ShallowCopy() { return (Fraction) MemberwiseClone(); }
}
```

```
class Client {
    static void Main() {
        Fraction a = new Fraction(1, 2);
        Fraction b = new Fraction(1, 2);
        Fraction c = new Fraction(3, 4);

        Console.WriteLine(a.ToString());          // 1/2
        Console.WriteLine(a);                     // 1/2 (ToString is called automatically)

        Console.WriteLine(a.Equals(b));           // true
        Console.WriteLine(a == b);                // false

        Console.WriteLine(a.GetHashCode());       // 3

        a = c.ShallowCopy();
        Console.WriteLine(a);                     // 3/4
    }
}
```

Example (for overloading == and !=)


```
class Fraction {
      int x, y;
      public Fraction(int x, int y) { this.x = x; this.y = y; }
      ...
      public static bool operator == (Fraction a, Fraction b) { return a.x == b.x && a.y == b.y; }
      public static bool operator != (Fraction a, Fraction b) { return ! (a == b); }
}
```

```
class Client {
      static void Main() {
          Fraction a = new Fraction(1, 2);
          Fraction b = new Fraction(1, 2);
          Fraction c = new Fraction(3, 4);

          Console.WriteLine(a == b);                  // true
          Console.WriteLine((object)a == (object)b);  // false

          Console.WriteLine(a.Equals(b));             // true, because overridden in Fraction
      }
}
```

- If == is overloaded,!= must be overloaded as well.
- Compiler prints a warning if == and != are overloaded, but *Equals* is not overridden.