CS 683 Emerging Technologies
Spring Semester, 2003
Doc 22 C# Inheritance
Contents

References

C# Language Specification, http://download.microsoft.com/download/0/a/c/0acb3585-3f3f-4169-ad61-efc9f0176788/CSharp.zip

Programming C#, Jesse Liberty, O'Reilly, Chapter 5

Constants Verses static readonly

Contants are known at compile time
Readonly vaules are not know until runtime

```
using System;
namespace Program1
{
   public class Utils
   {
      public static readonly int X = 1;
   }
}
namespace Program2
{
   class Test
   {
      static void Main() {
         Console.WriteLine(Program1.Utils.X);
      }
   }
}
```

Inheritance

Single Inheritance

```
public class A
	{
	int value = 0;
	public A(int value)
		{
		this.value = value;
		}
	}

class B : A
	{
	int x;

	public B(int first, int second) : base(first)
		{
		x = second;
		}
	}
```

Class Access Modifiers

- public
- protected internal
- protected
- internal
- private (default)

The class modifier limits the access of its members

```
class B : A
        {
        int x;

        public B(int first, int second) : base(first)
                {
                x = second;
                }
        }
```

Hiding & New

```
class Base
{
  public void F() {}
}
class Derived: Base
{
  public void F() {}              // Warning, hiding an inherited name
}
```

```
class Base
{
  public void F() {}
}
class Derived: Base
{
  new public void F() {}
}
```

Some Fun

```
class Base
{
  public static void F() {}
}
class Derived: Base
{
  new private static void F() {}      // Hides Base.F in Derived only
}
class MoreDerived: Derived
{
  static void G() { F(); }            // Invokes Base.F
}
```

```
class Base
{
  public static void F() {}
}
class Derived: Base
{
  new public static void F() {}
}
class MoreDerived: Derived
{
  static void G() { F(); }            // Invokes Derived.F
}
```

What Gets Printed?

```
using System;

class Base
{
	public void Foo()
		{
		Console.WriteLine("Base");
		}
}
class Derived: Base
{
	public void Foo()
		{
		Console.WriteLine("Derived");
		}
}


class Tester
{
	public static void Main()
	{
		Base top = new Derived();
		top.Foo();
	 }
}
```

Polymorphism

Base method must be declared virtual
Derived method must be declared override

```
class Base
{
	public virtual void Foo()
		{
		Console.WriteLine("Base");
		}
}
class Derived: Base
{
	public override void Foo()
		{
		Console.WriteLine("Derived");
		}
}


class Tester
{
	public static void Main()
	{
		Base top = new Derived();
		top.Foo();
	}
}
```

The Rules

A method named N is invoked with an argument list A
On an instance with a compile-time type C & a run-time type R

R is either C or a class derived from C

The invocation is processed as follows:

- First, overload resolution is applied to C, N, and A, to select a specific method M from the set of methods declared in and inherited by C.
- Then, if M is a non-virtual method, M is invoked.
- Otherwise, M is a virtual method, and the most derived implementation of M with respect to R is invoked.

The most derived implementation of a virtual method M with respect to a class R is determined as follows:

- If R contains the introducing virtual declaration of M, then this is the most derived implementation of M
- Otherwise, if R contains an override of M, then this is the most derived implementation of M.
- Otherwise, the most derived implementation of M is the same as that of the direct base class of R.

Example

```
using System;
class A
{
   public void F() { Console.WriteLine("A.F"); }
   public virtual void G() { Console.WriteLine("A.G"); }
}
class B: A
{
   new public void F() { Console.WriteLine("B.F"); }
   public override void G() { Console.WriteLine("B.G"); }
}
class Test
{
   static void Main() {
      B b = new B();
      A a = b;
      a.F();
      b.F();
      a.G();
      b.G();
   }
}
```

Output

```
A.F
B.F
B.G
B.G
```

What Happens Here?

```
using System;
class A
{
      public virtual void F() {Console.WriteLine("A");}
}
class B: A
{
  // Get a compile warning, hiding inherited F()
      public virtual void F() {Console.WriteLine("B");}
}

class C : B
{
      public override void F() {Console.WriteLine("C");}
}


class Tester
      {
      public static void Main()
            {
            A top = new C();
            top.F();
            B middle = new C();
            middle.F();
            }
      }
```

Override Rules

The method overridden by an override declaration is known as the overridden base method.

For an override method M declared in a class C, the overridden base method is determined by examining each base class of C, starting with the direct base class of C and continuing with each successive direct base class, until an accessible method with the same signature as M is located.

For the purposes of locating the overridden base method, a method is considered accessible if it is public, if it is protected, if it is protected internal, or if it is internal and declared in the same program as C.

A compile-time error occurs unless all of the following are true for an override declaration:

- An overridden base method can be located as described above.
- The overridden base method is a virtual, abstract, or override method. In other words, the overridden base method cannot be static or non-virtual.
- The overridden base method is not a sealed method.
- The override declaration and the overridden base method have the same return type.
- The override declaration and the overridden base method have the same declared accessibility. In other words, an override declaration cannot change the accessibility of the virtual method.

Calling Base Method

```
using System;

public class A
{
	virtual public void F()
	{
		Console.WriteLine("A");
	}
}

class B: A
{
	public override void F()
	{
		base.F();
		Console.WriteLine("B");
	}
}
```

Sealed

A sealed method overrides an inherited virtual method with the same signature

```
using System;
class A
{
   public virtual void F() {
      Console.WriteLine("A.F");
   }
   public virtual void G() {
      Console.WriteLine("A.G");
   }
}
class B: A
{
   sealed override public void F() {
      Console.WriteLine("B.F");
   }
   override public void G() {
      Console.WriteLine("B.G");
   }
}
class C: B
{
   override public void G() {
      Console.WriteLine("C.G");
   }
}
```

C cannot override F()

What Happens Here

```
using System;

class A
{
	public virtual void F() {
		Console.WriteLine("A.F");
	}
	public virtual void G() {
		Console.WriteLine("A.G");
	}
}
class B: A
{
	sealed override public void F() {
		Console.WriteLine("B.F");
	}
	override public void G() {
		Console.WriteLine("B.G");
	}
}
class C: B
{
	override public void G() {
		Console.WriteLine("C.G");
	}
	new public void F() {
		Console.WriteLine("C.F");
	}
}
```

Sealed Classes

A sealed class cannot have any derived classes

A sealed class cannot have any virtual methods

```
using System;

sealed class A
{
        public void F() {
                Console.WriteLine("A.F");
        }
}
class B: A      //Compile Error
{
}
```

Abstract Classes

An abstract method is declared with the modifier abstract

An abstract method
- Is a virtual method
- Cannot have a body
- Cannot be explicitly declared virtual

A class that contains an abstract method must be declared abstract

An abstract class

Does not have to have abstract methods

Cannot be instantiated directly

Example

```
public abstract class Shape
{
   public abstract void Paint(Graphics g, Rectangle r);
}
public class Ellipse: Shape
{
   public override void Paint(Graphics g, Rectangle r) {
      g.DrawEllipse(r);
   }
}
public class Box: Shape
{
   public override void Paint(Graphics g, Rectangle r) {
      g.DrawRect(r);
   }
}
```

Abstract Classes can have constructors

```
abstract class A
{
	int cat;

	abstract public void F();

	public A(int value)
	{
		cat = value;
	}
}
```

Cannot call a base abstract method

```
abstract class A
{
   public abstract void F();
}
class B: A
{
   public override void F() {
      base.F();     // Error, base.F is abstract
   }
}
```

Overriding with Abstract

Class B forces class C to implement F()

```
class A
{
   public virtual void F() {
      Console.WriteLine("A.F");
   }
}

abstract class B: A
{
   public abstract override void F();
}

class C: B
{
   public override void F() {
      Console.WriteLine("C.F");
   }
}
```

Protected access for instance members

When a protected instance member is accessed outside the program text of the class in which it is declared, the access is required to take place through an instance of the derived class type in which the access occurs.

```
public class A
{
   protected int x;
   static void F(A a, B b) {
      a.x = 1;    // Ok
      b.x = 1;    // Ok
   }
}
public class B: A
{
   static void F(A a, B b) {
      a.x = 1;    // Error, must access through instance of B
      b.x = 1;    // Ok
   }
}
```

Accessibility constraints

The following accessibility constraints exist:

- The direct base class of a class type must be at least as accessible as the class type itself.
- The explicit base interfaces of an interface type must be at least as accessible as the interface type itself.
- The return type and parameter types of a delegate type must be at least as accessible as the delegate type itself.
- The type of a constant must be at least as accessible as the constant itself.
- The type of a field must be at least as accessible as the field itself.
- The return type and parameter types of a method must be at least as accessible as the method itself.
- The type of a property must be at least as accessible as the property itself.
- The type of an event must be at least as accessible as the event itself.
- The type and parameter types of an indexer must be at least as accessible as the indexer itself.
- The return type and parameter types of an operator must be at least as accessible as the operator itself.
- The parameter types of an instance constructor must be at least as accessible as the instance constructor itself.

Examples

- The direct base class of a class type must be at least as accessible as the class type itself.

```
class A {}

public class B: A {} //Compile Error
```

- The return type and parameter types of a method must be at least as accessible as the method itself.

```
class A {}

public class B
{
   A F() {}
   internal A G() {}
   public A H() {}        //Compile Error
}
```

- The type of a field must be at least as accessible as the field itself.

```
class A {}
public class B
{
    public A sam;
}
```