Inheritance & Polymorphism

every class inherits from the `Object` class

C# is a hierarchical object language

Equals
GetHashCode
GetType
ToString

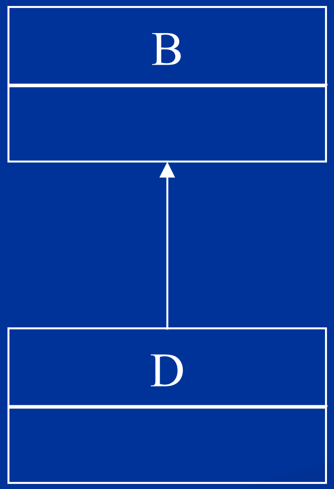
multiple inheritance is not allowed

instead, `interfaces` are used

the inheritance syntax is ':'

Inheritance & Polymorphism

For further examples, the base class will often be named B, and a derived class will often be named D



Object polymorphism

```
public class B
{
    string bst;
    int a;
}

public class D : B
{
    string dst;
}
```

```
public class test
{
        static void Main(string []
args)
        {
                B b1, b2;
                D d1, d2;

                b1 = new B();
                b2 = b1
                d1 = new D();
                d2 = d1;
        }
}
```

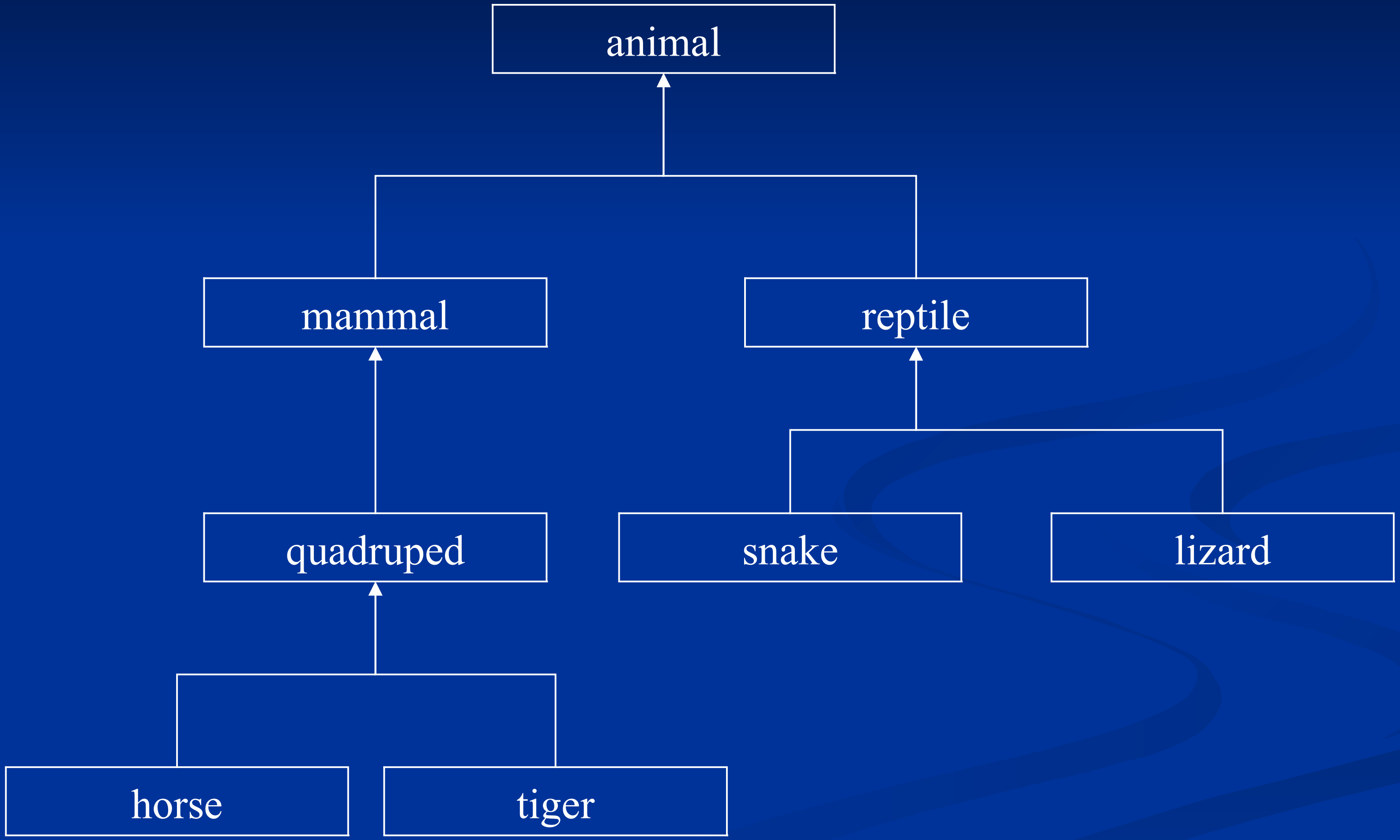
natural downcasting

```
public class test
{
        static void Main(string []
args)
        {
                B b1;
                D d1;

                d1 = new D();
                b1 = d1;
        }
}
```

only the "B" part of d1 is copied to b1

always allowed by the compiler

natural downcasting



natural downcasting

```
animal a = new mammal();
mammal m = new horse();
snake s = new snake();
```

- but, is it possible to consider `m` as a `horse` object rather than a `mammal` object ?

explicit polymorphism

```
mammal m = new horse();
horse h;

h=m; // not allowed, must be explicit

h = (horse)m; // compiler compliant
```

can produce execution errors !

checking class

use the boolean `is` operator to access the dynamic class of an object.

```
B b1;
b1 = new D();
if (b1 is D) // true here !
{
    …
}
```

```
mammal m;
m = new tiger();
if (m is tiger) // true
here !
{
        …
}
```

checking class

```
void f(animal a)
{
    if (a is snake)
    {
        ((snake)a).doSomething();
    }
    else if (a is lizard)
    {
        ((lizard)a).doSomething();
        ((lizard)a).doSomeLizardAction(…);
    }
    else if…
}
```

method polymorphism

different from Java !

```
class B
{
   public int x=1;

   public void met(int
   a)
   {
        x = x+a;
   }
}
```

```
class D : B
{
   public void met(int a)
   {
          x = x+a*10;
   }
}
```

same signature : masking

early binding

```
public class test
{
    B objb;
    D objd;
    objb = new B();
    objd = new D();

    objb.met(10);        11
    objd.met(10);        101
}
```

```
public class test
{
    B objb;
    D objd;
    objb = new D();
    objd = new D();

    objb.met(10);     11 or
    objd.met(10);     101 ?
                      101
}
```

method polymorphism

objb is declared (statically) with B class

the `met` method from B class will be used

this can be quite confusing (especially for Java developpers)

C# compiler delivers a warning

method polymorphism

use the `new` keyword to explicitly state that masking is intended :

```
class D : B
{
    public new void meth(int a)
    {
            x = x+a*10;
    }
}
```

method polymorphism

dynamic linking (method redefinition)

late binding (done at runtime)

the method to be called depends on the dynamic class of the object

uses the **`virtual/override`** keywords

virtual : as in C++, occurs at top level of a class hierarchy

method polymorphism

in derived class that redefines a method declared as `virtual` :

use the `override` keyword

if omitted, method is considered as masking the superclass method (as if you wrote `new`)

Inheritance

implicit call to the base class constructor

keyword base (analogous to Java super keyword)

two simple classes to illustrate simple inheritance

`instrument`

and

`piano` (a piano is an instrument)

example

```
class instrument // inherits automatically from object
{
   protected string name;  // protected : grants
                    //access to derived classes only
   public instrument()
   {
       P.rintln("constructing instrument");
   }

   public instrument(string s)
   {
       name = s;
       P.rintln("constructing instrument named "+s);
   }

   public override string ToString() // override allows
   polymorphism wrt object
   {
       return "this instrument is named "+s;
   }
}
```

example (continued)

```
class piano : instru // inherits automatically from
  object
{
  public piano() // or public piano():base()
  {
      P.rintln("constructing piano");
  }

  public piano(string s):base(s) // explicit call to
  instrument constructor
  {
      P.rintln("constructing piano named "+s);
  }

  public override string ToString()
  {
      return "this piano is named "+s;
  }
}
```

example (finished)

```
class test // also inherits from object
   (not an important information by the way)
{
   [STAThread]
   static void Main(string[] args)
   {
       piano p=new piano("Stenway");
       instrument i = new instrument();

       object o = new piano("Pleyel");
       P.rintln(new piano());

       P.ause();
   }
}
```

example (outputs)

```
constructing instrument named Steinway
constructing piano named Stenway
constructing instrument
constructing instrument named Pleyel
constructing piano named Pleyel
constructing instrument
constructing piano
this piano is named
```

abstract classes

a method can be declared abstract :

no code, just a *prototype* or *signature*

if a method is abstract, the class containing this method must be declared `abstract` as well.

objects of an abstract class can't be created

the role of an abstract class is to be derived

derived class will override and *eventually* implement abstract methods from the base class

abstract classes : example

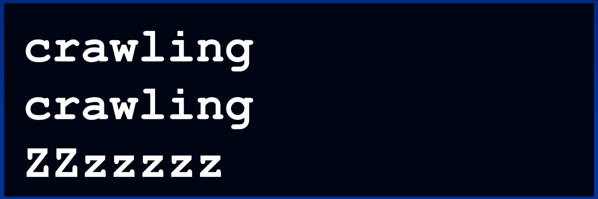
```
abstract class animal
{
    public abstract void move();
}
abstract class reptile : animal
{
    public void hibernate()
    {
        System.Console.Write("ZZzzzzz");
    }
}

class snake : reptile
{
    public override void move()
    {
        System.Console.Write("crawling");
    }
}
```

a code for the move() method is provided

an abstract method is virtual

using polymorphism

```
class test
{
    static void Main(string [] args)
    {
        animal a;
        reptile r;

        a = new snake();        // ok
        r = new snake();        // ok

        a.move();
        r.move();
        r.hibernate();
    }
}
```



```
crawling
crawling
ZZzzzzz
```

late binding : method calls based on the dynamic class of the objects on which they operate

using polymorphism

```
class test
{
    static void Main(string [] args)
    {
        animal [] zoo = {new snake(), new lizard(), new horse(),
    new lion(), new platypus()};

        // time for a walk in the park

        foreach (animal a in zoo)
        {
            a.move();
        }
    }
}
```

Interfaces

dealing with multiple inheritance : C++ or Java ?

Java-like approach is used : a class can derive only from one class but can derive from several interfaces.

syntax :

```
class D : B,I₁,I₂,…,Iₙ
```

where B is the base class and I_i is an interface

Interfaces

an interface :

- specifies some behaviors with no implementation;
- is a contract, and may contain methods, properties, *events* and *indexers*, but no attributes;
- contains only signatures;
- all method, properties, events, indexers are public;
- can be derived;
- can inherit from another interface.

A class inheriting from an interface must implement all methods, properties, events and indexers.

Interfaces

to build objects, create a class that implements the interface

two examples compared : the animal hierarchy

- with (abstract) classes
- with interfaces

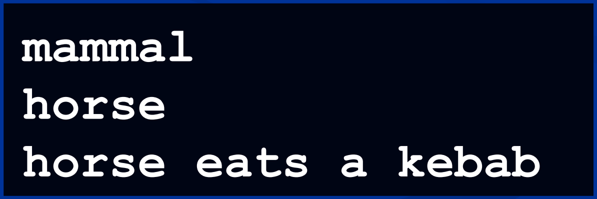
example

```
abstract class animal
{
    public abstract string name{get;}
    public abstract string catego{get;}
    public abstract void eat(string stg);
}

abstract class mammal:animal
{
    public override string catego
    {
          get {return "mammal";}
    }
}
```

```
class horse : mammal
{
    public override string name
    {
              get{return "horse";}
    }

    public override void eat(string s)
    {

    System.Console.Write(this.name+"
              eats a "+s);
    }

}
```

example

```
class test
{
   static void Main(string [] args)
   {
         horse h = new horse(); // or animal h = new horse()

         System.Console.WriteLine(h.catego);
         System.Console.WriteLine(h.name);

         h.eat("kebab");

         System.Console.Read();
   }
}
```



```
mammal
horse
horse eats a kebab
```

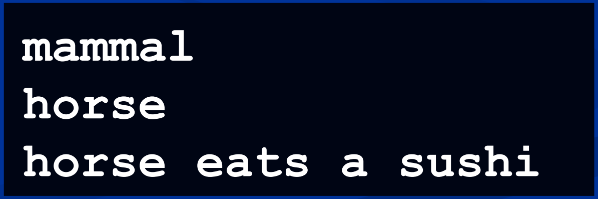
example

```
interface Ianimal
{
    string name{get;}
    string catego{get;}
    void eat(string stg);
}

abstract class mammal: Ianimal
{
    public string catego
    {
        get {return "mammal";}
    }

    public abstract void eat(string stg);
    public abstract string name{get;}

}
```

```
class horse : mammal
{
    public override string name
    {
            get{return "horse";}
    }

    public override void eat(string s)
    {

    System.Console.Write(this.name+"
            eats a "+s);
    }

}
```

reference to an interface

```
class test
{
    static void Main(string [] args)
    {
        Ianimal h = new horse();
        System.Console.WriteLine(h.catego);
        System.Console.WriteLine(h.name);

        h.eat("sushi");

        System.Console.Read();
    }
}
```



```
mammal
horse
horse eats a sushi
```

Conclusion

with OO languages

2 development phases are expressed :

analysis and design (through UML)

writing application code

three tools

interfaces

abstract classes

classes

Conclusion

- use interfaces to specify behaviors;
- use abstract classes when you have to write generic code;
- use classes when you have to write class-specific code;
- use late binding as much as possible;
- try to delay application code writing as late as possible in your development process.

Exception handling

defensive coding :

trying to anticipate any error :

- coding error (bugs)
- external events (exceptions) : connection lost, drive failure, peripheral errors

Exception handling

- use the `try/ catch / finally` coding structure to handle exceptions

- unhandled exceptions brutally end program execution (the exception go up the stack until it finds a method that catches it)

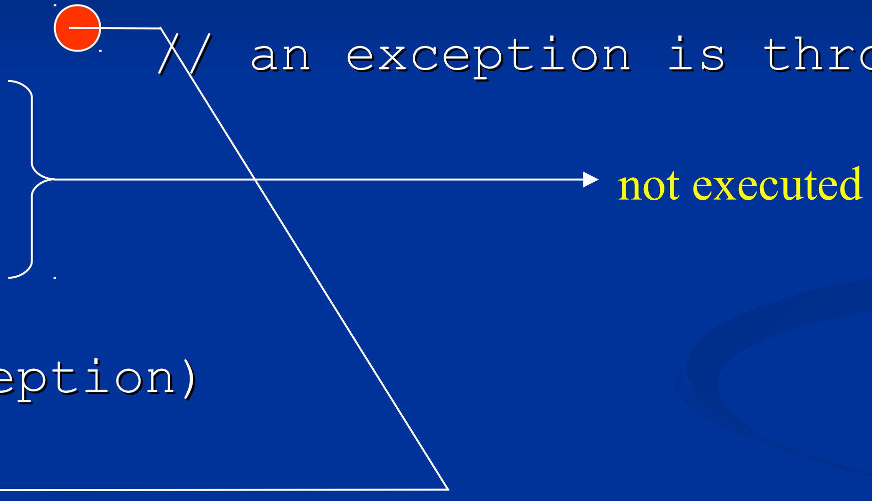
- No « throws » keyword in C# ! Methods do not declare the possible exceptions they would throw

Exception handling

```
try
{
  code
}
catch(Exception)
{
  exception handling code
}
[finally
{
  code always executed
}]
```

Program sequence

```
try // let there be an error provoked by line 2

{
   line 1;
   line 2;        // an exception is thrown
   line 3;
   …
   line n;
}
catch(Exception)

{
   exception handling code
}

next instructions
```

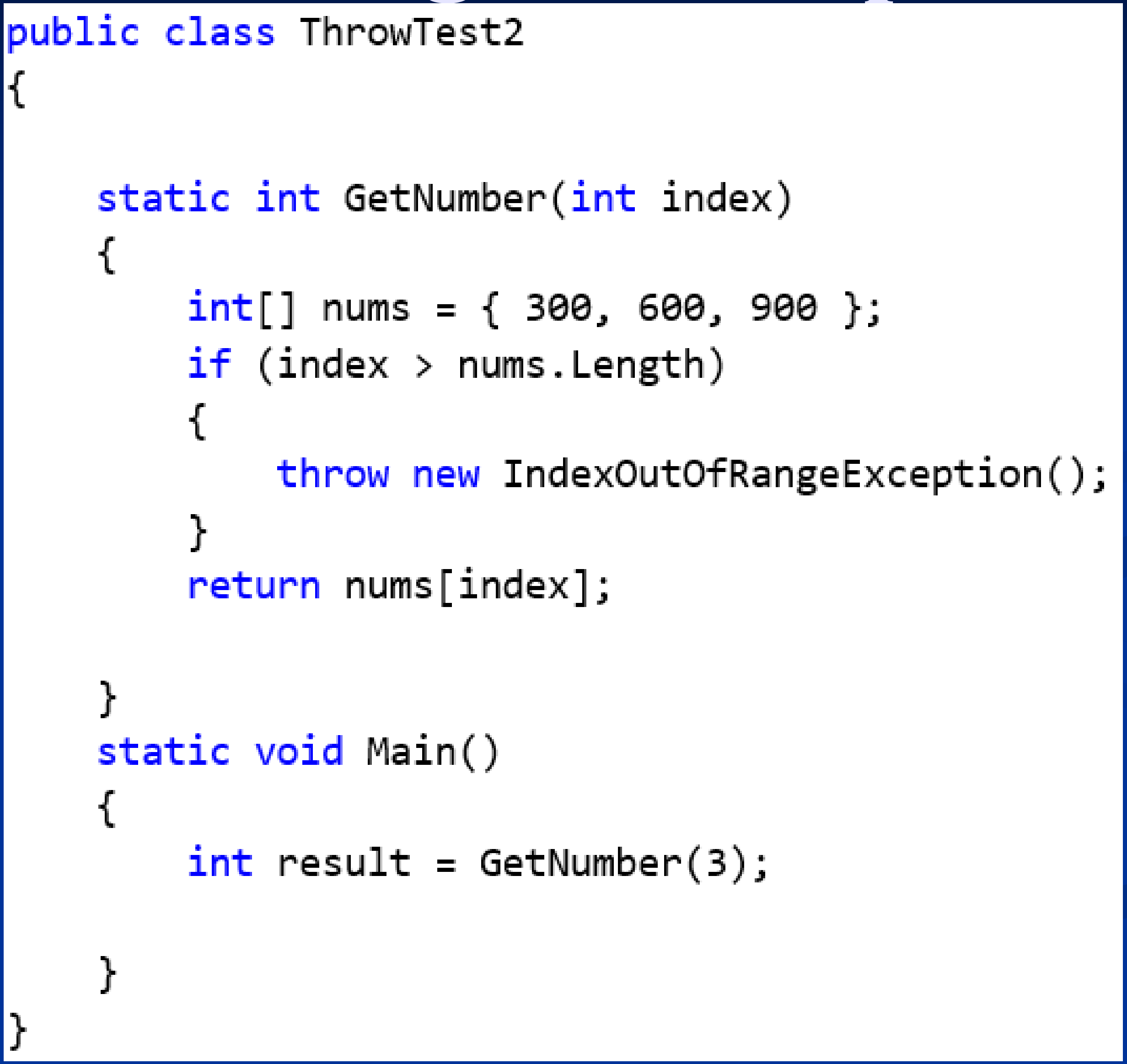


not executed

Try-Catch hierarchy

- Many possible « catch » blocks can be added to handle different exceptions
- The order is important as the first catch compatible with the error is used !
  - More specialized exceptions should be specified before the more general ones
- A throw statement can be used in a catch block to re-throw the exception

Throwing an Exception



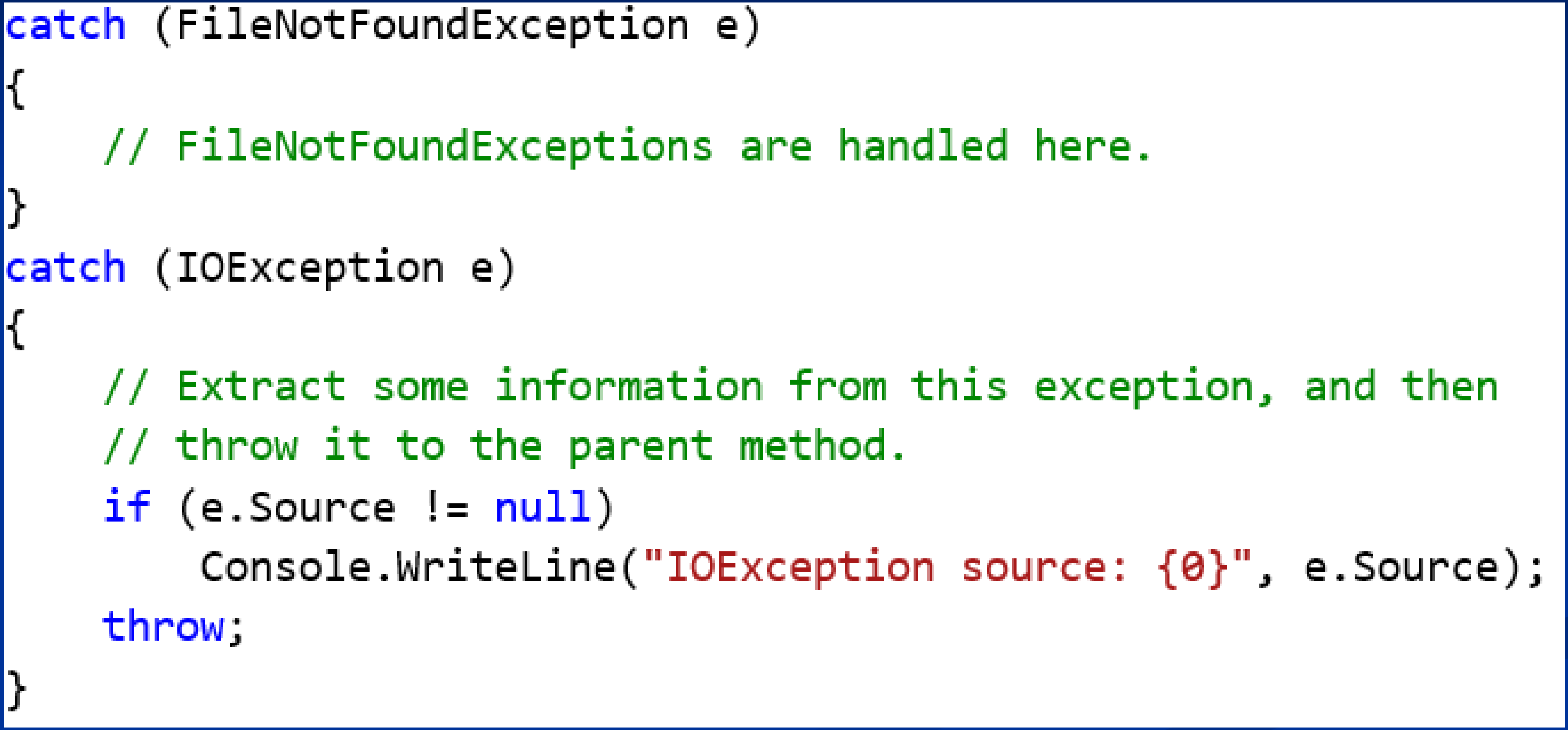
```
public class ThrowTest2
{

    static int GetNumber(int index)
    {
        int[] nums = { 300, 600, 900 };
        if (index > nums.Length)
        {
            throw new IndexOutOfRangeException();
        }
        return nums[index];

    }
    static void Main()
    {
        int result = GetNumber(3);

    }
}
```

Exception processing

in the catch statement, declare an Exception object :

```
try
{
  something
}
catch(Exception ex) // or any other
  exception
{ // use ex object
  Console.Write(ex.TargetSite);
}
```

Catching Exceptions



```
catch (FileNotFoundException e)
{
    // FileNotFoundExceptions are handled here.
}
catch (IOException e)
{
    // Extract some information from this exception, and then
    // throw it to the parent method.
    if (e.Source != null)
        Console.WriteLine("IOException source: {0}", e.Source);
    throw;
}
```

Why finally ?

```
int i=123;
string s = « hello » ;
object o = s ;
try {
  i = (int) o ; // throws an invalid cast exception    …. //
    instructions here are not executed
} finally {
  // last instructions run before leaving this function
}
```

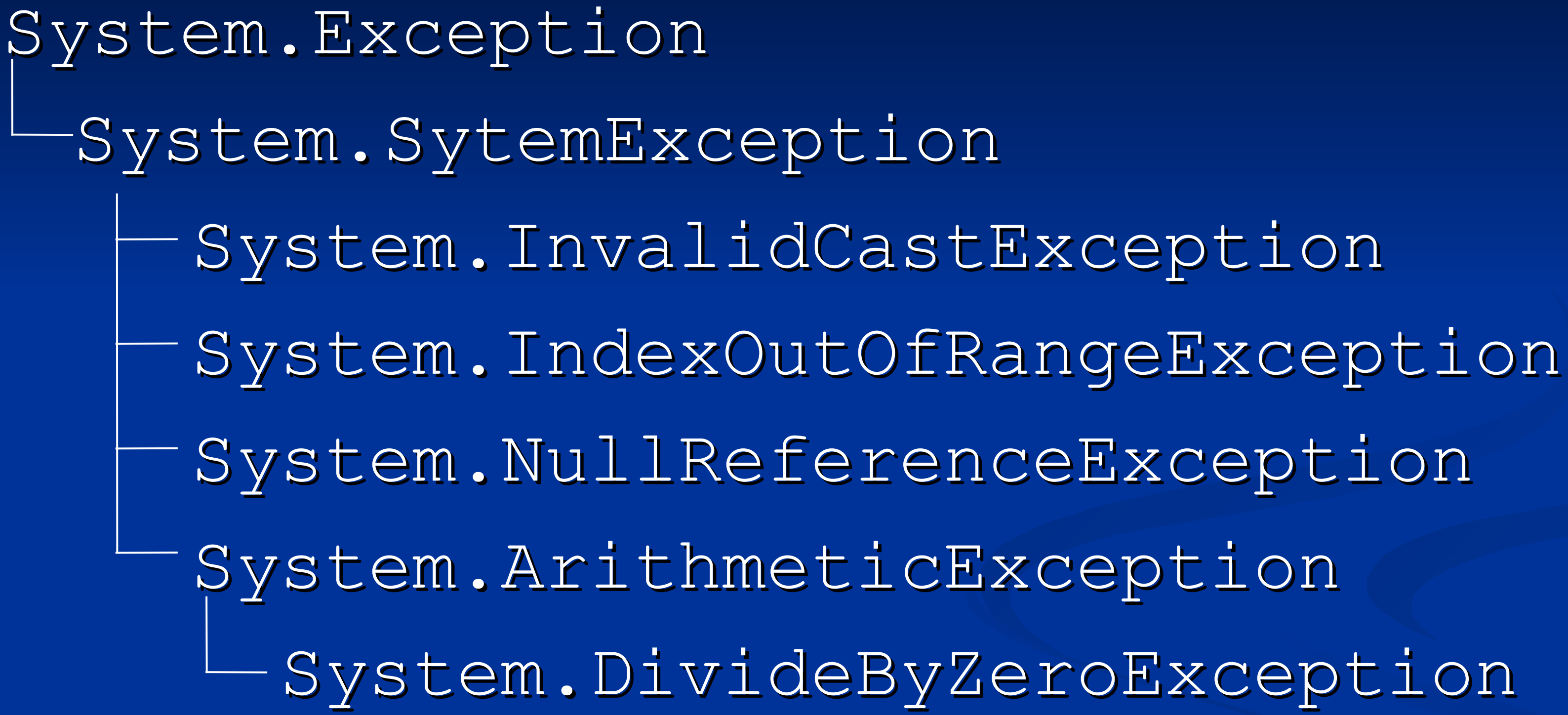
Exception hierarchy

`System.Exception` class

all existing exceptions inherit from `System.Exception`

all user exceptions must inherit from an existing `Exception` class

Exception hierarchy



- System.Exception
  - System.SytemException
    - System.InvalidCastException
    - System.IndexOutOfRangeException
    - System.NullReferenceException
    - System.ArithmeticException
      - System.DivideByZeroException

Exception processing

use the Exception **properties** to collect information on what happened

```
Message
Source
StackTrace
TargetSite
```