Introduction to C#

Agenda

- .Net Framework
- Basic Programming
- Basic Object Oriented Programming Concepts
- Exception Handling

What is the .Net Framework?

- .NET is a software framework that supports large class libraries and several programming languages.
- Applications written in a .NET language do not execute directly on the hardware, instead they execute in a software environment known as the Common Language Runtime (CLR).
- This virtual machine provides a number of services such as security, memory management and exception handling.
- The class libraries and the CLR make up the .NET framework

Basic Programming

- Variables
- Types
- Declarations
- Constants
- Calculations
- Assignments
- Conversions

Data Types

Data Type	Use For	Storage Size in bytes
Boolean	True or False value	2
Byte	0 to 255, binary data	1
Char	Single Unicode character	2
Date	1/1/0001 through 12/31/9999	8
Decimal	Decimal fractions, such as dollars/cents	16
Single	Single precision floating-point numbers with six digits of accuracy	4
Double	Double precision floating-point numbers with 14 digits of accuracy	8
Short	Small integer in the range -32,768 to 32,767	2
Integer	Whole numbers in the range -2,147,483,648 to +2,147,483,647	4
Long	Larger whole numbers	8
String	Alphanumeric data: letters, digits, and other characters	Varies
Object	Any type of data	4

Declaring Variables

- To declare a variable, you must supply both a type and a name.
- The type determines what the data inside the variable means.
- The name allows you to identify the variable.
- E.g.
  - string a="Hello";
  - string b;
  - b = "World!";
  - int intValue1=80;
  - int intValue2;
  - intValue2 =20;
  - const int days = 365;
  - int sum = intValue1+ intValue2;

Conversions

Method	Description	Examples
ToBoolean()	converts a type to a Boolean value	string sample = "True"; bool myBool = Convert.ToBoolean(sample);
ToChar()	converts a type to a char type	String val = “T"; char res = Convert.ToChar(val)
ToDouble()	converts a type to a double type	double doubleVal = Convert.ToDouble("85");
ToInt16()	converts a type to a 16-bit int type	short result = Convert.ToInt16("123");

Flow Control

Flow control allows us to perform a different set of instructions depending on some conditions. This condition may depend on user input or the result of some previous calculation.

Conditional Operators

- For any flow control statements, we must be able to use conditional operators. These operators compare the values of two variables. These operators are:

C# Operators	Meaning
==	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal to

Boolean Logic

- A special kind of mathematics where there are only 2 possible values.
- These values are **true** and **false**.
- Special operators are used with these values to create expressions.
- A boolean expression will evaluate to either **true** or **false** in its entirety.

Logical Operators

- We can make more complicated comparisons by using logical operators. These operators are:

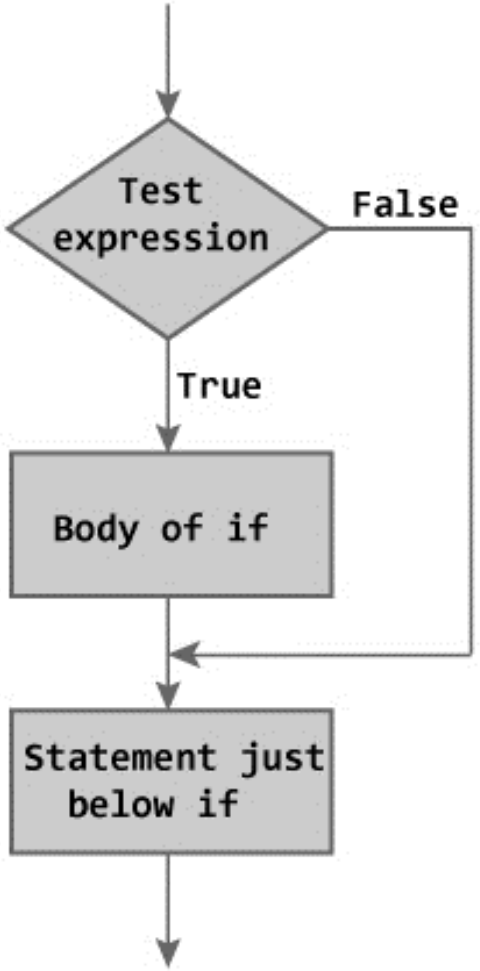
C# Operators	Meaning
&&	And: Both must be true
||	Or: Either must be true
^	Xor : One but not both must be true
!	Not: Reverses Condition

If statements

- If statements allow a set of instructions to be executed depending on the condition. These statements follow the pattern:

```
if(condition) {
        statements;
}
```

Flowchart of if

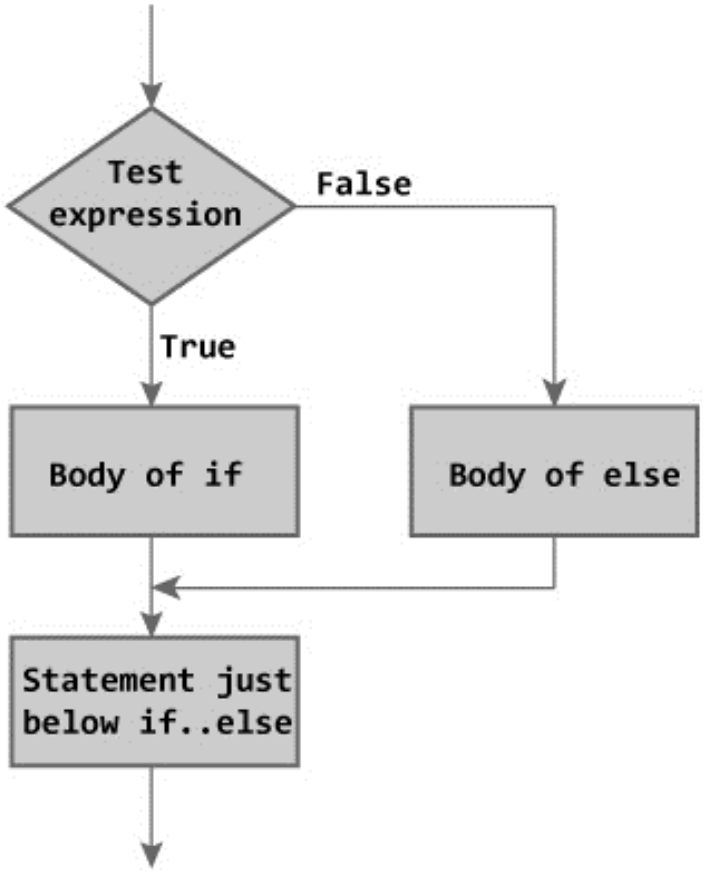


If/Else statements

- The if/else statement allows one of two instructions paths to be taken depending on the result of a condition. These statements follow the pattern:

```
if(condition1) {
        statements;
} else {
        statements;
}
```

Flowchart of if...else



DOs and DON'Ts:

- Write the nominal path through the code first, then code the unusual cases
- Put the normal/expected case immediately after the "If" and not after the "Else". Error conditions should always go into the "Else"
- Careful to branch correctly on equality to prevent "off-by-one" errors eg. Using > instead of >= or Using < instead of <=
- Sometimes an empty "Else" block with comments is useful to indicate that the case has been considered and deemed unnecessary.
- With nested and complicated "If/Elseif" statements, make sure ALL possibilities are covered

Flow Control - Loops

- Another important method of controlling the flow of a program is a **loop**.
- Loops repeatedly perform a set of instructions until they terminate.

While Loops

- A While loop will continue executing as long as a certain condition is evaluated to true. They follow this form:

```
while(condition) {
     statements;
     statements;
     statements;
}
```

While Loops

- A While loop will continue executing as long as a certain condition is evaluated to true. They follow this form:

```
while(condition) {
     statements;
     statements;
     statements;
}
```

Do While

```
do {
     statements
}while (condition);
```

For Loops

- For loops are most commonly used for looping over a pre-defined range.

```
for(int i = start; i < end; i ++ step)
    statements;
}
```

Loops – Best Practice

- Consider a while loop if there is a condition in the loop that jumps out – terminates the loop.
- Consider a For loops if there is a pre-defined range.
- Use very meaningful names in long loops.
- In nested For loops, carefully named variables prevent the “index cross talk”
- Don’t mess with the loop index to make the loop terminate

Loops – Best Practice

Some Remedies and best-practice:

- Minimize the factors that affect the loop (simplify)
- Make your loops short enough to view all at once
- Limit nesting loops-within-loops to three levels
- Move sections of the loop-body into functions/routines when the complexity becomes too high
- Make long loops especially clear (a single clear and unmistakable exit, use comments to indicate where)
- Enter loop in one location only!
- Initialize the loop immediately before it – proximity principle
    - Keep loop 'housekeeping' chores at either beginning or end
- Use *while(True)* for infinite loops
    - Sometimes the code needs to run forever
- Revert to For loops from While when you find yourself forgetting to modify loop control variables at the bottom
    - Conversely, don't cram the For header – revert to While

Conventions and best-practices

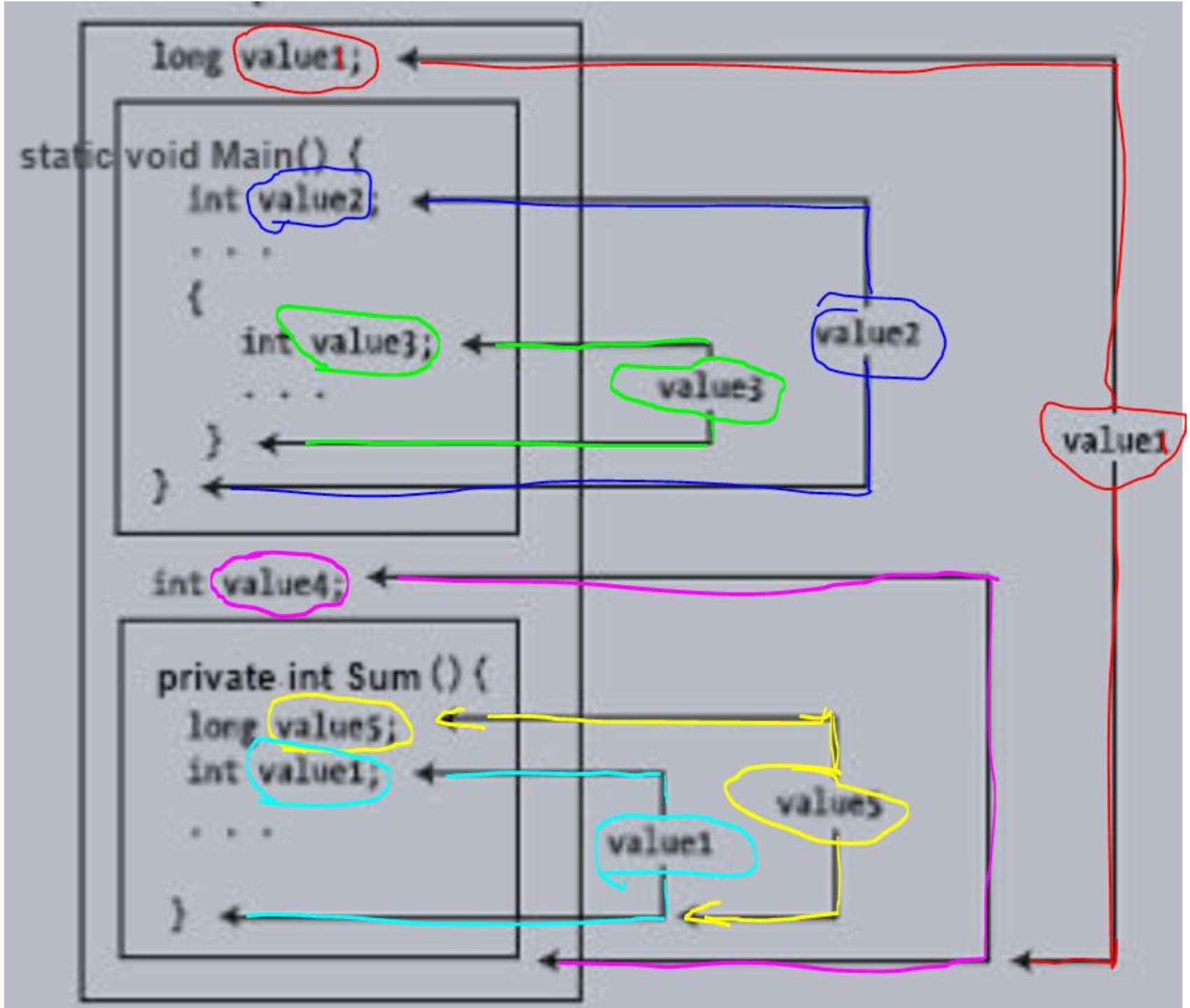
- Booleans: do not use ‘flag’ as a name
  - Boolean variables should always imply **true/false**
  - Eg. if(flagSet) **BAD**
    if(dataIsReady) **GOOD**
  - Bad names: status, flag, negative names!
  - **Good names: done, error, found, success, ok, isLast**

- When experiencing “trying to figure out” a section of code – consider renaming **all** variable names

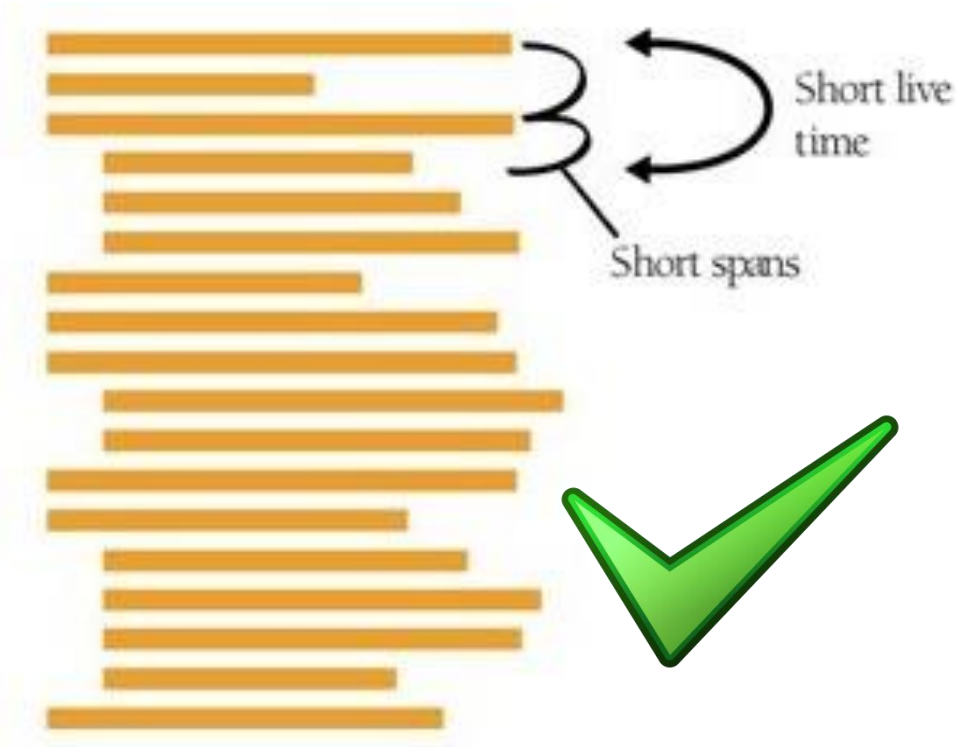
- Favour “read-time” over “write-time” convenience

Scope of a Variable

- Scope, or visibility, refers to the extent to which your variables are known and can be referenced throughout a program.


long value1;
static void Main() {
int value2;
. . .
{
int value3;
. . .
}
}
value2
value3
value1
int value4;
private int Sum () {
long value5;
int value1;
. . .
}
value5
value1


Long live
time
Long
spans


Short live
time
Short spans

Parameters: ref

- **pass-by-reference** mechanism
- The **ref** keyword is used to pass an argument as a reference. This means that when value of that parameter is changed in the method, it gets reflected in the calling method.
- An argument that is passed using a **ref** keyword must be initialized in the calling method before it is passed to the called method.

Parameters: out

- **pass-by-reference** mechanism
- Variables passed as out arguments do not have to be initialized before being passed in a method call.
- The usage of the keyword out, guarantees that the procedure to which the variable is being passed by value **must** and **will** assign a value to this variable before the procedure ends. It is a guaranteed contract.
- **out** specifies that the parameter is an output parameters, i.e. it has no value until it is explicitly set by the method.

ref vs out

//ok

- int x;
- Foo(out x);

// Error: y should be initialized before calling the method

- int y;
- Foo(ref y);

Methods/Functions/ Subroutines – Convention

- Names begin with capitals; every subsequent word capitalised – **P**ascal**C**asing
- Names must be meaningful – contain verbs e.g. **C**reate**P**assword**H**ash, **R**emove**A**ll(), GetCharArray(), Invoke()…
- Cannot begin with a number
- Variables: ✓ camel Casing: exitButton (btnExit)

namespaces

- .NET uses namespaces to organize its many classes
  - System.Console.WriteLine("Hello World!");

  - using System;
  - Console.WriteLine("Hello World!");

- declaring your own namespaces can help you control the scope of class and method names in larger programming projects.

OO - Constructors

A Class defines the template or blueprint of an Object.

Instances of a Class can be instantiated.

Instantiating objects of a class requires the use of a **Constructor**.

Constructors set up an instance when it is created.

OO - Constructors

Some variables can be created and initialised trivially as:

```
int i = 0;
```

The above are **value-type** variables or **primitives**.

However, some variables require the explicit use of a **constructor**:

```
public class Student {
        public bool isInitialized;
        public Student ()
        {
                isInitialized = true;
        }
}
```

```
class TestStudent {
        static void Main()
        {
                Student tong = new Student ();

                Console.WriteLine(t.isInitialized);
        }
}
```

The above are **reference-type** variables and are instantiations of class types.

OO - Constructors

Constructors can also take parameters that are used to initialise the instance.

Constructors are important as they allow us to ensure that an instance (with possibly private parameters) has been properly initialised with correct values in the data members.

```
Dim t As Timer = New Timer(1000)
```

```
Timer t = new Timer(1000);
```

Constructors

Multiple constructors can be created provided they all take different parameters.

This enables objects to be instantiated in different ways.

The only condition for these constructors is that the **number or type of parameters must** be different.

No two constructors may have the same parameter list or else the compiler cannot decide which one to use.

Constructors

In C# **constructors** are a Subroutine with the **same name as the class**

```
public class Student {
    private string _name;
    private int _age;
    private int _id;

    public Student(string name, int age, int id) {
        _name = name;
        _age  = age;
        _id   = id;
    }
}
```

Constructors

Objects can now be instantiated in a variety of ways:

```
Dim tim As New Student(“Tim”, 25, 01234567)
Dim tom As New Student(“Tom”, 27)
```

```
Student tim = new Student(“Tim”, 25, 01234567);
Student tom = new Student(“Tom”, 25);
```

Constructors

An alternative to multiple constructors is to implement constructors with same **optional parameters** that use default values.

These constructors have optional parameters which have default values.

If the user supplies a value it will be used, else the default value is used instead.

Constructors

```
public class Student {
    private string _name;
    private int _age;
    private int _id;

    public Student(string name, int age, int id = 0) {
        _name = name;
        _age  = age;
        _id   = id;
    }
}
```

Constructors

This single constructor with an **optional parameter** allows the creation of instances of Student in two different ways.

```
Student tim = new Student(“Tim”, 25, 01234567);
Student tom = new Student(“Tom”, 25);
```

Properties

Encapsulating data is the centrepiece of Object Oriented Programming.

Unless it is necessary, all data members should be declared as **private**.

However, these members must at times be accessed from outside of the object.

Properties can be used to *get* or *set* the values of data members.

Properties

These properties can be written as follows:

```
public class Student {
  // this is a field. It is private to your class and stores the actual data.
   private string _name;
  // this is a property.
   public string Name {
     get {
       return _name;
     }
     set {
       _name = value;
     }
   }
}
```

Properties

Now a property of an object can be accessed

```
Student tim = new Student(“Tim”, 25, 01234567);
Console.WriteLine(tim.Name);
tim.Name = “Timothy”;
```

Properties

For example, one can make it **read-only**.

```
public class Student {
  private string _name;
  public string Name {
    get {
      return _name;
    }
  }
}
```

Properties

Or perform some kind of **validation** before changing the value.

```
class Student {
  private string _name;
  public string Name {
    get {
      return _name;
    }
    set {
      if(value != “”) {
        _name = value;
      }
    }
  }
}
```

Properties

Using this mechanism, the members of the user-defined class can now be accessed in a **controlled** way.

Members can be accessed/modified, but only if the object approves the access or modification.

Encapsulation is therefore preserved.

Behaviour

Classes with only data members are of limited use.

In order to complete the expressive power of OO programming, class behaviour must also be defined.

The defined methods **can access the private members and make any changes to them** as necessary.

For example:

Behaviour

```
class Student {
  public string PrintAge() {
    return “Student “ + _name + “(“ + _id + “) is”
                      + _age + “ years old”;
  }
}
```

Behaviour

This behaviour can now be used in code:

```
Student tim = new Student(“Tim”, 25, 01234567);
Console.WriteLine(tim.PrintAge());
```

Output:

```
Student Tim (01234567) is 25 years old
```

Object-Oriented Programming

Real power of OOP resides in fully utilizing advanced concepts:

- Inheritance
- Method overriding
- Polymorphism
- Abstract classes
- Interfaces

When used in conjunction and intent, they enable software to be constructed using **design patterns**

Inheritance

OO concept of inheritance recognises the existence of **hierarchies** in real world objects.

Class inheritance **enables the acquisition** of properties and behaviours from other classes.

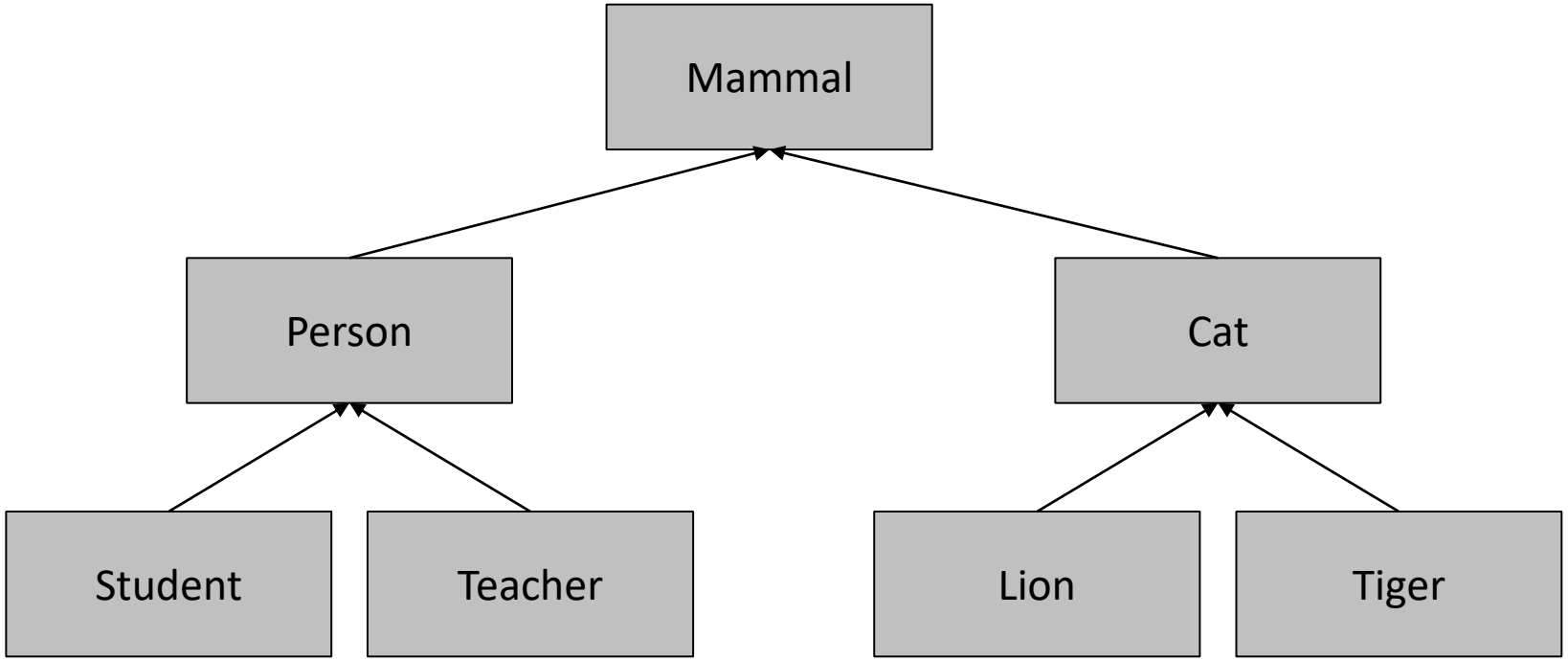
This hierarchy can be **leveraged**: code reuse, better modelling …

For instance

a Student *is a* Person

a Person *is a* Mammal

Inheritance



Inheritance

Inheritance represents an **'is a type of'** relationship between two classes. *A Student 'is a' Person, and a Person 'is a' Mammal.*

This phrase is a good method of determining if inheritance should be used or not.

Inheritance implemented in .NET C# syntax using the keyword :

:

Inheritance

```
public class Mammal {
    protected int _weight;
}

public class Person : Mammal {

    protected string _name;
}

public class Student : Person {

    protected int _ID;
}
```

Inheritance

The Student class defines a single data member unique to a Student - an **_ID** number.

Student inherits the data member **_name** from the Person class and **_weight** from the Mammal class.

In the hierarchy of the objects, Student represents a **specialization** and Mammal represents a **generalization**.

Inheritance

Student is a **subclass (**or a **derived/child class)** of the **superclass (**or a **base/parent class)** Person. Mammal is the superclass of the subclass Person.

All the inherited data members in the above example **are available** to the objects of the Student class.

The code example below illustrates this in the case of a user-defined constructor for the Student class:

Inheritance

```
public class Student : Person {


    protected int _ID;

    public Student(string aName, int anID, int aWeight) {
        _name = aName;
        _ID = anID;
        _weight = aWeight;
    }
}
```

Inheritance

```
public class Person : Mammal {

    protected string _name;
    public Person(string aName, int aWeight) {
        _name = aName;
        _weight = aWeight;
    }
}
```

Overriding

Method overriding is a language feature for **modifying** inherited behaviours.

Permits the derived class to implement or **fine-tune** a method inherited from a base class.

Overriding

When the given abstract or virtual method is called from an instance of a base class, the **overridden** method will be used.

Overriding provides a ***specific implementation*** through keywords:

Base class: `virtual` or abstract if the class is abstract

Derived class: `override`

Overriding

```
public class Person : Mammal {
    protected string _name;

    public virtual void Print() {
       Console.WriteLine(_name);
    }
}

public class Student : Person {
    protected int _ID;

    public override void Print() {
     Console.WriteLine(_name + _ID);
    }
}
```

Overriding

This can be used as follows:

```
Person p = new Person(“Tim”,60);
p.Print();
```

```
Student s;
s = new Student(“Tim”,1234,60);
s.Print();
```

Polymorphism

Polymorphism is a powerful OOP technique.

Literally means **'many forms'**.

In essence**,** Polymorphism allows a derived class to be treated as an instance of its base class.

Inheritance, Polymorphism

```
Abstract Class:
public abstract class A{
    public int _a;
    public abstract void M();
}
public class SubAClass : A {
   public override void M()
   {Console.WriteLine("Hello World!");}
}
```

Access Modifiers

Access modifiers must be carefully considered with class hierarchies.

Following access modifiers can be used:

private - Not accessible outside the class or in a derived class.
public - Accessible by any part of the program.
protected - Accessible only by the class or a derived class.
internal - Accessible by any code in the same assembly,
but not from another assembly

Object Oriented Programming

Why use OOP? Pros:

i.code reuse - reduction of code duplication by moving code up to superclasses

ii.Extensibility- New functionality is easy to add without affecting existing functionality.

iii.data hiding – information is hidden from other objects

iv.Modularity - easy to change one module without affecting the other.

Object Oriented Programming

Cons:

i.dependencies

ii.complicated and hard to understand

iii.may promote bad coupling

iv.more prone to negative effects of changing requirements

Input Validation

There are two main methods of doing input validation:

Prevention and Exception Handling

These methods stop an application crashing if invalid data is used in some operation.

Prevention

Prevention is the simplest input validation technique

Prevention stops invalid input causing the application to crash by verifying the data **before** it is used

The implementation achieved through conditional constructs like "**if/else**" statements

Prevention

For example, a calculator application:

```
while (true) {
        Console.WriteLine("Please enter an expression");
        int a, b;
        char c;
        a = int.Parse(Console.ReadLine());
        c = char.Parse(Console.ReadLine());
        b = int.Parse(Console.ReadLine());
        if (c == '+') {
            Console.WriteLine("= " + (a + b));
        } else if (c == '-') {
            Console.WriteLine("= " + (a - b));
        } else if (c == '*') {
            Console.WriteLine("= " + a * b);
        } else if (c == '/') {
            Console.WriteLine("= " + a / b);
      }
}
```

Prevention

C#

```
Please enter an expression:
3
+
4
= 7
Please enter an expression:
10
/
5
= 2
Please enter an expression:
3
/
0
DivideByZeroException
```

Prevention

Divide-by-zero causes crashes in C#

The programmer must check that the second operand for division is not 0

```
} else if(c == '/') {
   if(b == 0) {
    Console.WriteLine("Cannot divide by zero!");
   } else {
    Console.WriteLine("= " + (a/b));
   }
}
```

Prevention

```
Please enter an expression:
3
/
0
Cannot divide by zero!
```

Exception Handling

- Alternative is exception handling
- Rather than attempting to prevent an error from ever occurring, exception handling assumes the input is correct but has code to handle an error if it occurs
- Exception indicates something has gone wrong during execution

Exception Handling

- Exception indicates something has gone wrong during execution
- Exceptions result in application crashes
- Exception handling allows a program to continue executing
- Promotes development of fault tolerant software

Exception Handling

The anatomy of exception handling:

1.When an error occurs at runtime, an exception is **thrown**

2.Code to **catch** exceptions can be written

3.When a specific exception type is caught, it can be appropriately **handled**, uncaught exceptions break software

4.Once an exception is handled, another block of code can be executed for **finalizing** (tidying up) anything else that generally needs to be performed following exceptions. This block is optional and called regardless of errors.

- Releases database connections, file handlers etc.

Exception Handling

Exceptions can be **thrown** by a function, subroutine or operation.

Exceptions can be caught by the application using a **try/catch** block.

```
try {

} catch(exception variable) {

} finally {

}
```

Exception Handling

We can use exception handling to prevent our previous calculator example code from crashing:

```
} else if (c == '/') {
   try {
      Console.WriteLine("= " + a / b);
   } catch (DivideByZeroException e) {
      Console.WriteLine("Cannot Divide by Zero!");
   }
}
```

Exception Handling

Try/catch blocks can be used to encompass larger blocks of code

Multiple types of exceptions can be caught by the same try/catch block

The exception types are organized in an **hierarchical order**, from most specific to most generic

Exception Handling

```
} catch(DivideByZeroException e) {
    Console.WriteLine("Cannot divide by zero!");
  } catch(FormatException e) {
    Console.WriteLine("Invalid Integer!");
  } catch(Exception e) {
    Console.Writeline("Error: " + e.Message);
  }
}
```

Exception Handling

Class Exception catches **any** abnormal condition

Class Exception is useful for catching **unpredicted** errors

Useful mechanism

- Can throw exceptions from procedures and allow calling code to handle the exception
- Can query the exception object to find out where the error occurred

Agenda

- .Net Framework
- Basic Programming
- Basic Object Oriented Programming Concepts
- Exception Handling