Computer Science 4
Semester 4

Contents

Chapter 1

The Fall of Communism

The most basic C# program (equivelant to `int main(void)` in C) is

```
using System;

namespace name_of_program_here {
class Program {
    static void Main(string[] args) {
        /*
         * Your code goes here
         * */
    }
}
}
```

1.1 Similarities with C

```
/* Variable assignment */
int x = 5, i = 0;
float y = 4.5;
double z = 3.1415;
char c = 'A';

/* Basic Operations */
x + y;
x * y;
x++;
etc...

/* Conditionals */
if (x == y) {
    /*...*/
} else if (x < y) {
    /*..*/
} else {
    /*...*/
}
```

```
/* Ternary Operator */
int w = (x < y)? y : x;

/* Loops */
while (i < 6) {
    /*...*/
}

do {
    /*...*/
} while (i < 6);

for (int i = 0; i < 6; i++) {
    /*...*/
}
```

1.2 Basic C#

1.2.1 Input Output

Basically how to take input from the user and give an output back.
In C we have the `scanf()`, which is used to get input from the user and place it in to a variable. in C# we have the `Console.ReadLine()` which is similar however it *returns* the user input as a string rather than placing it in a variable so if we want user input in C we do

```
    int x;
    scanf("%d",&x);
```

While in C# we have to read the input as a string then convert it to an integer like so

```
int x = System.Int32.Parse(Console.ReadLine());
```

Likewise, in C we use `printf()` to print something to the screen, but in C# we use `Console.WriteLine()` to do the same thing, however the difference is that C forces you to use a string to format before it is printed while C# can handle the formatting for you, for example the C code

```
int x = 5;
printf("%d\n",x);
```

is equivelant to the following C# code

```
int x = 5;
Console.WriteLine(x); // print directly, no need to specify `%d` or `\n`
```

1.2.2 Format Strings

Strings in C# can be defined using `string name = "World"`. If we want to format strings when printing we do `printf("Hello, %d!\n",name)`, but in C# this can be done in a much simpler fashion using

```
Console.WriteLine($"Hello, {name}!")
```

or alternatively using

```
Console.WriteLine("Hello, " + name + "!")
```

Note:-

2 strings can be *concatenated* together in C#(added together) like so

```
string firstName = "Hamboola ";

string lastName = "Habooling";

string fullName = firstName + lastName; /* fullName => "Hamboola Hambooling" */
```

1.3 Classes

In C#, a class is a blueprint or a template for creating objects of a particular type. Classes encapsulate data and behavior into a single unit, making it easier to manage and maintain code.
To define a class in C#, you use the `class` keyword followed by the name of the class. Here is an example:

```
public class Person {
    // class members go here
}
```

In this example, we've defined a class called `Person`. The `public` keyword indicates that the class is accessible from other parts of the program.

Within a class, you can define members such as fields, properties, methods, and events. These members define the data and behavior that are associated with the class. Here's an example of a class with some members:

```
public class Person {
    // fields
    private string _name;
    private int _age;

    // properties
    public string Name {
        get { return _name; }
        set { _name = value; }
    }

    public int Age {
        get { return _age; }
        set { _age = value; }
    }
    public int Address { get; set;} // Instead of modifying a field, it modifies itself

    // methods
    public void SayHello() {
        Console.WriteLine("Hello, my name is " + _name + " and I am " + _age + " years old.");
    }
}
```

In this example, we've defined the Person class with two fields (`_name` and `_age`), two properties (`Name` and `Age`), and one method (`SayHello`).

Fields are variables that hold data associated with the class. Properties provide a way to access and modify the values of fields. Methods define behavior associated with the class.
Once you've defined a class, you can create objects of that class using the `new` keyword:

```
    Person person = new Person();
```

This creates a new instance of the `Person` class and assigns it to the `person` variable. You can then access the members of the class using the dot notation:

```
    person.Name = "Alice";
    person.Age = 30;
    person.SayHello();
```

This sets the `Name` and `Age` properties of the `person` object and calls the `SayHello` method, which outputs a message to the console.

1.3.1 Difference between a field and a property

In C#, a field is a variable that holds data associated with a class or struct. A property, on the other hand, provides a way to access and modify the values of fields.

The main difference between fields and properties is that fields are accessed directly using the dot notation, while properties are accessed using getter and/or setter methods.

In the previous listing, the `Person` class has a private field `_name` and a public property `Name`.

To access the `_name` field directly, you use dot notation, like this:

```
    Person person = new Person();
    person._name = "Alice"; // This is not allowed since _name is private
```

In this example, we're trying to set the value of the `_name` field directly, but since it's declared as private, we get a compilation error.

To access the `Name` property, you also use dot notation, but you use the getter and/or setter methods defined in the property:

```
    Person person = new Person();
    person.Name = "Alice"; // This calls the setter method of the Name property
    string name = person.Name; // This calls the getter method of the Name property
```

In this example, we're using the `Name` property to set and get the value of the `_name` field. When we set the value of the `Name` property, it calls the setter method that assigns the value to the `_name` field. When we get the value of the `Name` property, it calls the getter method that returns the value of the `_name` field.

In summary, fields and properties are both used to store and retrieve data, but properties provide a level of abstraction that allows you to control how the data is accessed and modified.

1.3.2 Constructors

In C#, a constructor is a special method that is used to initialize an object when it is created. It has the same name as the class and does not have a return type, and it can be overloaded with different parameter lists to provide different ways of creating objects.

When an object is created using the `new` keyword, the constructor is called automatically, and any initialization code that is included in the constructor is executed. This can include setting default values for fields, allocating memory for objects, or initializing any required resources.

Going back to out previous example:

```
public class Person {
    public string Name { get; set; }
    public int Age { get; set; }

    public Person(string name, int age) {
        Name = name;
        Age = age;
    }
}
```

In this case, the constructor takes two parameters and assigns them to the `Name` and `Age` properties of the object. We can now create objects of this class using the constructor:

```
Person p1 = new Person("Alice", 30);
Person p2 = new Person("Bob", 25);
```

In addition to providing initialization code, constructors can also be used to enforce business rules or perform validation on input parameters. For example, we could modify the constructor of the `Person` class to ensure that the age parameter is greater than zero:

```
public Person(string name, int age) {
    Name = name;
    if (age ≤ 0)
        throw new ArgumentException("Age must be greater than zero"); // Simply throws an error
    Age = age;
}
```

Now, if we try to create a person with an age less than or equal to zero, an exception will be thrown:

```
Person p3 = new Person("Charlie", 0); // throws ArgumentException
```

1.3.3 Inheretence

Inheritance is a concept in object-oriented programming that allows a class to inherit properties and methods from a parent or base class. In C#, a class can inherit from another class using the ”extends” keyword.

The parent class is also called a base class or superclass, while the child class is also called a derived class or subclass. The derived class inherits all the properties and methods of the base class and can add its own properties and methods or override those inherited from the base class.

For example, consider a base class called `Animal` that has a property called `Name`:

```
public class Animal {
    public string Name { get; set; }
}
```

We can now create a derived class called `Dog` that inherits from the `Animal` class and adds its own property called `Breed`:

```
public class Dog : Animal {
    public string Breed { get; set; }
}
```

In this case, the `Dog` class inherits the `Name` property from the `Animal` class and adds its own `Breed` property. We can now create objects of the `Dog` class and set the `Name` and `Breed` properties:

```
Dog dog = new Dog();
dog.Name = "Fido";
dog.Breed = "Labrador";
```

In addition to inheriting properties, a derived class can also inherit methods from the base class. For example, consider a base class called `Shape` that has a method called Area:

```
public class Shape {
    public double Area() {
        return 0;
    }
}
```

We can now create a derived class called `Rectangle` that overrides the `Area` method to calculate the area of a rectangle:

```
public class Rectangle : Shape {
    public double Width { get; set; }
    public double Height { get; set; }

    public override double Area() {
        return Width * Height;
    }
}
```

In this case, the `Rectangle` class inherits the `Area` method from the `Shape` class and overrides it with its own implementation. We can now create objects of the `Rectangle` class and call the `Area` method:

```
Rectangle rect = new Rectangle();
rect.Width = 5;
rect.Height = 10;
double area = rect.Area(); // area = 50
```