Industrial Programming

Lecture 4: C# Objects & Classes

What is an Object

- Central to the object-oriented programming paradigm is the notion of an ***object***.
- Objects are the nouns…
  - a person called John…
- Objects have *characteristics* (fields, attributes or properties)
  - John has black hair and is 20 years old
- Objects can perform *actions* (methods or functions)
  - John can tell me the sum of two numbers

What is a Class

- Before we create an object, we must describe it:
  - What characteristics it has (attributes/fields).
  - What actions it can perform (methods/functions).
- The **class** is the blueprint / template / plan / recipe / description for the object.
  - Here we describe the fields & methods for all objects of this type.
- An **object** is an **instance** of a **class**.
  - Creating an object is often called **instantiation**.
- For example, we can define a class person with attributes name, age etc and then instantiate it to a name John

What is a Class (cont'd)

- We can create **many** objects from one class.
  - may spend a lot of time creating the class initially, but creating many objects from the class is easy! Re-use!
  - for example, we can have lots of buttons that all have the same attributes and methods available.
- BUT… all objects created from one class are **NOT identical**.
  - **Same** name & data types for the fields, but **different** field values (different state).
  - Different button names, text, size etc.
- The values of these fields define the **state** of an object.

x

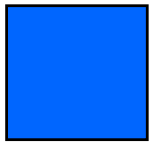
These shapes have common attributes. We could define a **Rectangle** class.

Attributes:

Name:

Data Type:

y



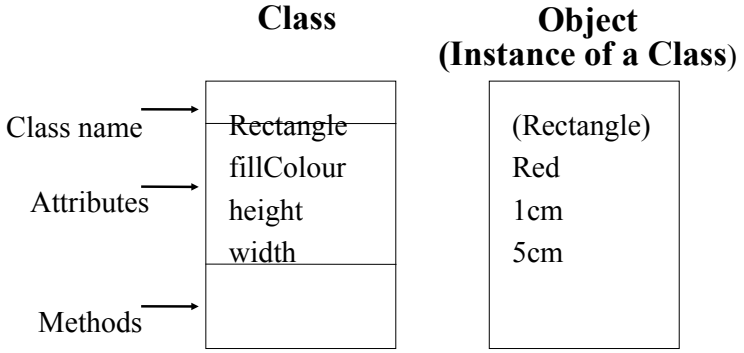
Each object has the same attribute names & data types, but different attribute values, i.e.

z

x)

y)

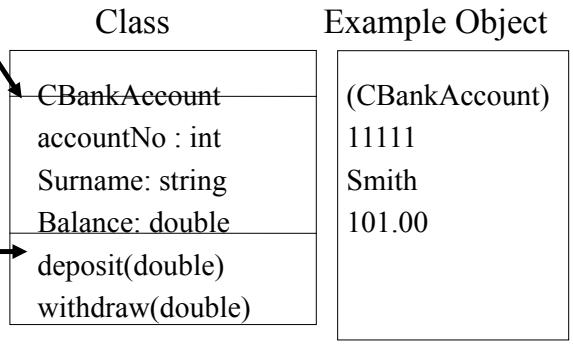
z)



Shows ***state***.
No need to indicate methods as they exist for a whole class

Bank Account

- Attribute:
  - Account No.
  - Surname
  - Balance (in £'s)
- Methods:
  - deposit
  - withdraw



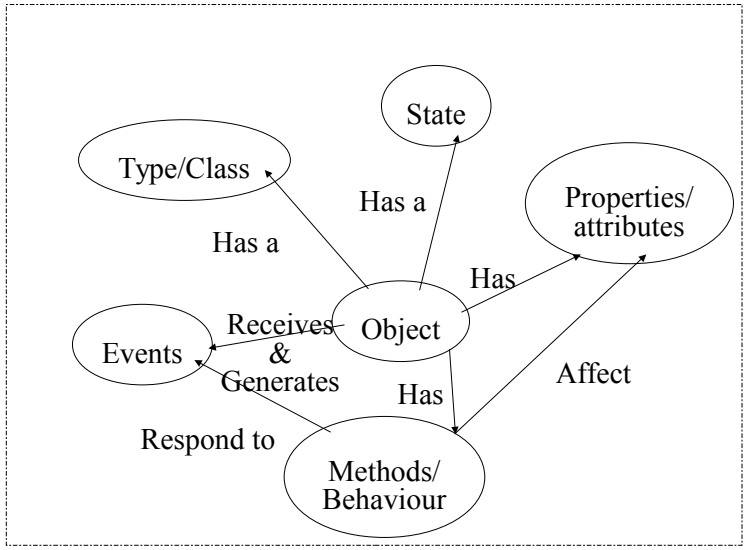
Declaring & Creating Objects

- **Declare** a name for an object of a specific type (class), e.g.

```
BankAccount myAccount;
```

- **Create** (instantiate) an object of a specific type (class), e.g.

```
myAccount = new
  BankAccount(11111,"Smith");
```



Simple C# Class

```
using System;
class Point
{
  public int x;
  public int y;

  public Point(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
}
class Test{
  public static void Main()
  {
    Point point1 = new Point(5,10);
    Point point2 = new Point(20, 15);
    Console.WriteLine("Point1({0}, {1})", point1.x, point1.y);
    Console.WriteLine("Point2({0}, {1})", point2.x, point2.y);
  }
}
```


Example Explained

- A class named *Point* is defined.
- It has two integer members *x* and *y*.
- The class includes a constructor.
  - A special method called to construct an instance of the class.
  - It takes two integer parameters.
  - Keyword *this* refers to the current instance.

Class and Constructor

- Classes are declared by using the keyword *class* followed by the *class* name and a set of *class* members surrounded by curly braces.
- Every *class* has a constructor, which is called automatically any time an instance of a *class* is created.
- The purpose of constructors is to initialise *class* members when an instance of the *class* is created.
- Constructors do not have return values and always have the same name as the *class*.

Constructors (cont'd)

- A *default constructor*, without parameters, is generated automatically.
- The user can define more constructors by overloading the default constructor, passing values to initialise fields.
- Object initialisers allow you to separate initialisation from the constructor method:

```
BankAccount bonusAcc = new
  BankAccount(1112,"Smith")
  { balance = 20; };
```


Default values of fields

- Unless explicitly initialised in the constructor, fields will have these default values:

`Numeric (int, long...)`	`0`
`Bool`	`False`
`Char`	`'\0'`
`Enum`	`0`
`Reference`	`null`

Anonymous Types

- Anonymous types reduce the coding overhead in creating a class.
- They are typically used for types that are used only once.
- They are useful in the context of LINQ when connecting to databases.
- Notation:

```
var myCircle = new { radius=3 };
```

- Note that this variable is read-only.

Example Explained (cont'd)

- Another class named *Test* is defined.
- It contains a static *main function* where program execution starts.
- In the *main function,* two *Point* objects (instances) are created (using *new*).
- The x and y coordinates of the two points are printed out.
- The data fields are accessed directly *(public* fields*),* not a good idea. Why?

Hiding Data Fields

```
using System;
class Point{
   private int x;
   private int y;

  public Point(int x, int y){
    this.x = x;
    this.y = y;
  }
  public int GetX() {return(x);}
  public int GetY() {return(y);}
}
class Test{
  public static void Main(){
  Point point1 = new Point(5,10);
  Point point2 = new Point(20, 15);
  Console.WriteLine("Point1({0}, {1})", point1.GetX(), point1.GetY());
  Console.WriteLine("Point2({0}, {1})", point2.GetX(), point2.GetY());
  }
}
```


Hiding Data Fields (cont'd)

- Access modifier *private* is used to hide data fields.
- Member functions are used to access the data fields.
- Member functions *GetX() and GetY()* take no input parameters and return an integer (the coordinate value).

Access Modifiers

- `public`: no access restrictions
- `private`: only methods of the same class can access the field
- `protected`: only methods in the same class and in classes derived from it can access the field
- `internal`: accessible to methods of any class in this class' assembly (collection of files, wrapped up in a executable or library)

Instance and Static Members

- Fields and methods can be instance or static members of the class.
- Each object has its own copy of an instance field. All fields so far have been instance fields.
- A static field exists just *once for a class* and is shared by all objects of that class. This is useful for counting the number of objects of a class.
- An example of a static field in the `Points` class:

```
public static int noOfPoints = 0;
```

Static Methods

- The same distinction between instance and static members exists for methods.
- An instance method is always applied to an object and can access the object's fields via the `this` variable. E.g.

```
point1.GetX();
```

- A static method is associated to class rather than an object and takes all arguments via its parameters. E.g.

```
Console.WriteLine(“Hello world!”);
```


C# Properties

- Properties are another way of hiding fields
- **Properties look like attributes but behave like methods**:

```
class Point{
 private int x;
 private int y;

 public int PointX {
  get { return x; }
  set { this.x = value; }
 }
 // analogous for PointY
```


C# Properties (cont'd)

- Every lookup for PointX will be translated into a call of the get function, e.g.

```
Console.WriteLine("Point1({0}, {1})", point1.PointX,
    point1.PointY);
```

- Every assignment to PointX will be translated into a call to the set function, e.g.

```
point1.PointX += 10;
```

- As shorthand notation you can use automatic properties, matching the names to the fields

```
public int PointX { get ; set ; }
```

- A private field `PointX` will be generated automatically by the compiler

C# Methods

- The constructor, GetX() and GetY() are methods.
- A method can have four parts:
  - Method name.
  - Parameters list.
  - Return type.
  - Access modifier.

Semantic information should be added in comments:

- What is the meaning of a parameter?
- What are the invariants of the method/class?

Overloading

- C# allows you to define different versions of a method/function in class, and the compiler will automatically select the matching one based on the *parameters* supplied.
- Generally, you should consider overloading a method when you need several methods that take different parameters, but *conceptually do the same thing*.
- You should not use overloads when two methods really do different things.

Example: Overloading

```
public class AddingNumbers
  {
   public int add(int a, int b)
    {
     return a+b;
    }
   public int add(int a, int b, int c)
   {
    return a+b+c;
   }
 }
```

Calling Overloaded Methods

```
int i = add(2, 3);
int j = add(2, 3, 4);
```


Operator Overloading

- Using the `operator` keyword, it is possible to overload existing operators,

```
public static
Complex operator +(Complex a, Complex b)
    {
     return new Complex(a.Real+b.Real,
                        a.Imag+b.Imag);
    }
```


Inheritance

- A central concept in object-oriented programming.
- A class is derived from another class.
- This allows the programmer to build a class hierarchy.
- A main activity in program design is the design of a suitable class hierarchy.
- Useful for code reuse.

Inheritance Example: Base Class

```
using System;
class Person{
   private string fName;
   private string lName;
   private string address;

   public Person(string fName, string lName, string
   address){
   this.fName = fName;
   this.lName = lName;
   this.address = address;
   }
   string GetfName(){return fName;}
   string GetlName(){return lName;}
   string GetAddress(){return address;}
}
```


Inheritance Example: Subclass

```
using System;
class Student: Person{
   private string matricNo;
   private string degree;

   public Student(string fName, string lName, string
   address, string matricNo, string degree): base(fName,
   lName, address){
      this.matricNo = matricNo;
      this.degree = degree;
   }
   string GetMatricNo(){return matricNo;}
   string GetDegree(){return degree;}
   }
```


Test Class

```
class Test{
 public static void Main(){
   Person p = new Person("John", "Smith",
   "Edinburgh");
   Student s = new Student("Brian", "Hillman",
   "London", "99124678", "CS");
   Console.WriteLine("Student matric no: {0} ",
   s.GetMatricNo());
   Console.WriteLine("Student address: {0} ",
   s.GetAddress());
   Console.WriteLine("Person address: {0} ",
   p.GetAddress());
   }
}
```


Example Explained

- *Person* is a base class.
- *Student* is a subclass of *Person.*
- It inherits all the fields and methods in *Person* and defines new ones.
- Its constructor uses `this` to distinguish member fields from method arguments.
- Its constructor uses the notation

`:base (fName,lName,address)`

to call the constructor of the base class with these arguments.

Interfaces

- An interface is a *contract*.
- A class that implements an interface must implement *all* of its methods.
- Whereas a class can inherit from just one class, it can implement several interfaces.
- These interfaces characterise various roles the class can take.

An Example of an Interface

- An interface `IStorable`, with methods for reading and writing data:

```
interface IStorable {
    void Read ();
    void Write(object obj);
    int Status { get ; set ;}
  }
```


An Example of an Interface

- Here is one possible implementation

```
public class Document : IStorable {
  public Document (string str) {
    Console.WriteLine("Creating document with: {0}", str);
  }
#region IStorable
  public void Read () {
    Console.WriteLine("Executing document's read method for IStorable");
  }
  public void Write(object obj) {
    Console.WriteLine("Executing document's write method for IStorable");
  }
  // property required by IStorable
  public int Status { get; set ; }
#endregion
}
```


3 Pillars of Object-oriented Programming

- **Encapsulation**: each class should be self-contained to localise changes. Realised through public and private access modifiers.
- **Specialisation**: model relationships between classes. Realised through inheritance.
- **Polymorphism**: treat a collection of items as a group. Realised through methods at the right level in the class hierarchy.

Exercises

(a) Implement the bank account example as discussed in the lecture.

(b) Complete the Points example and implement access to the x- and y-fields, using direct access, public methods, and (automatic) properties, respectively.

Exercises

(a) Use inheritance and overloading to define a method Area, that works on different shapes, namely circles, rectangles and squares.

(b) Let the user decide how many of these objects to construct, with which parameters, calculate the overall and per-shape area and print it

(c) Write a (polymorphic) function that takes an array of shapes and calculates the total area covered by all elements.

Exercises

(a) Modify the ReadLine exercise from the previous lecture to generate instances of classes containing an unsigned short, unsigned int and unsigned long field, respectively.

(b) Implement basic arithmetic on complex numbers using operator overloading.

(c) Implement the data structure of binary search trees with operations for inserting and finding an element.