Sydney

The Fundamentals of C#

Static keyword, encapsulation and access modifiers (public, private)

Objectives:

- Static keyword in C#
- What is encapsulation?
  - Properties
- C# access modifiers
  - public
  - private

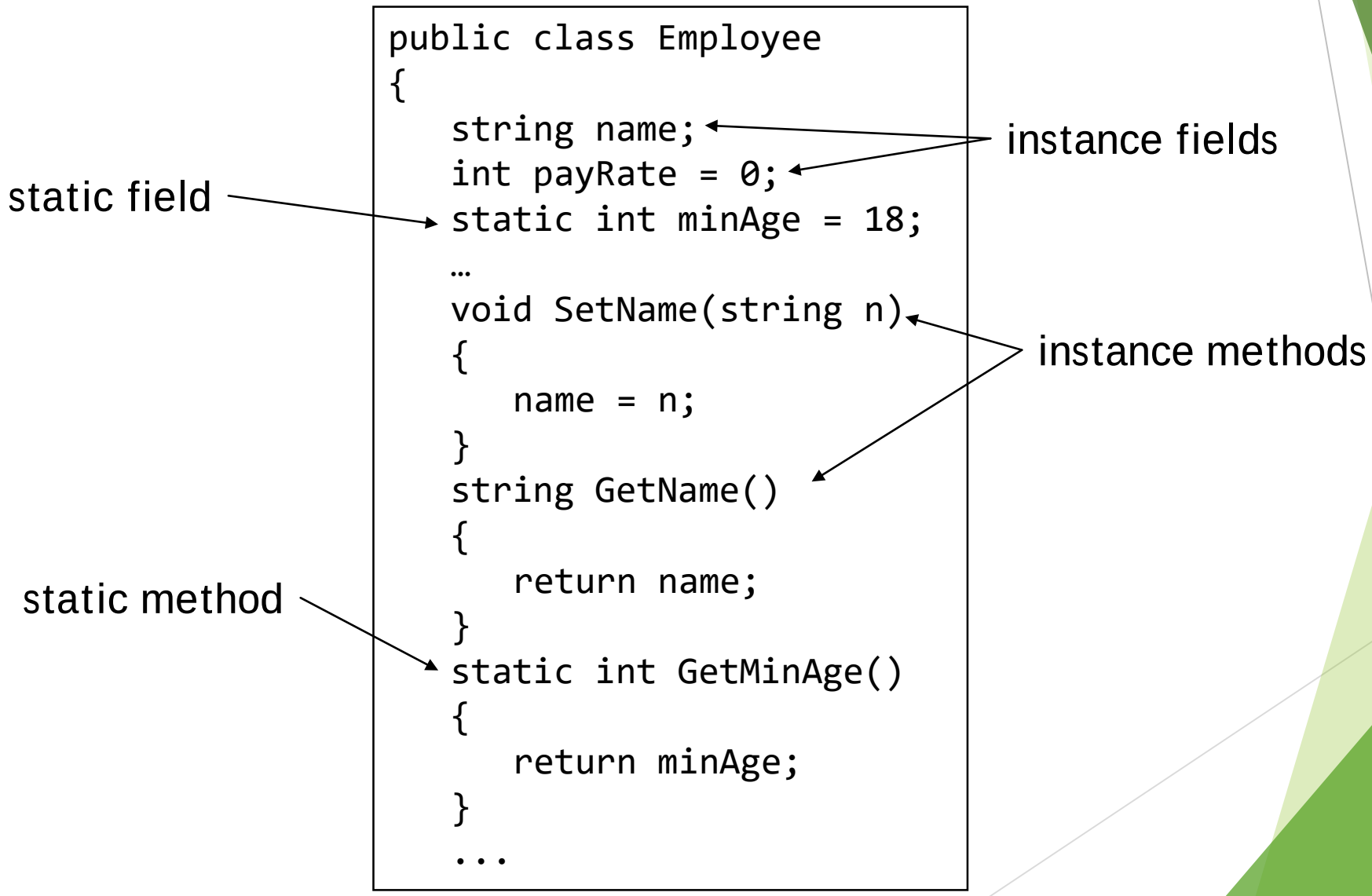
static keyword

Static fields and methods belong to a type (or class), rather than to an object.

The **WriteLine** and **ReadLine** methods of the **Console** class are examples of static methods.

Declaring static fields and methods requires use of the **static** keyword. Let's see an example.

example


public class Employee
{
string name;
int payRate = 0;
static int minAge = 18;
…
void SetName(string n)
{
name = n;
}
string GetName()
{
return name;
}
static int GetMinAge()
{
return minAge;
}
...
static field
instance fields
instance methods
static method

static fields

For a static field, there is exactly one value for the type, **ever**. No object needs to be created in order to access a static field.

Accessing a static field is done by typing the name of our class, then a dot, then the name of the field we want to access.

instance fields

For an instance field, there is a value for each object (instance) we create. If we had 10 objects, each would have its own instance field. In order to use an instance field, we must first initialise a new object using the **new** keyword.

Accessing an instance field is done by typing the name of the variable storing our object, then a dot, then the name of the field we want to access.

static and instance

```
Employee x = new Employee();
Employee y = new Employee();
Employee z = new Employee();
```

instance

```
x.name
y.name
z.name
```

3 different values

static

```
Employee.minAge
```

One value

static and instance

instance

```
x.name
y.name
z.name
```

```
Employee.name
```

static

```
x.minAge
y.minAge
z.minAge
```

```
Employee.minAge
```
✓

static fields and methods

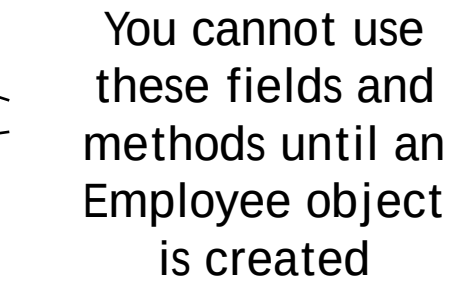
A few important limitations to be aware of with static.

Static methods **CANNOT** access instance fields/methods. This is because instance fields/methods require an object to be initialized, but static methods do not.

However, instance fields/methods **CAN** access static fields/methods.

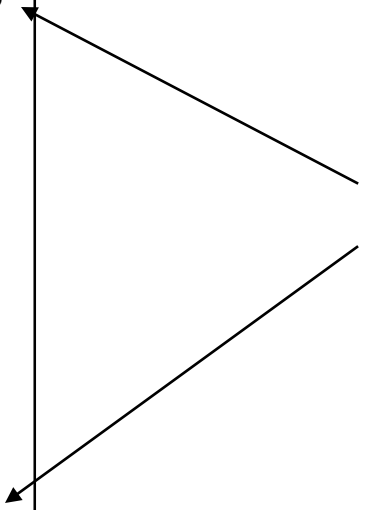
non static fields and methods

```
public class Employee
{
    string name;
    int payRate = 0;
    static int minAge = 18;
    …
    void SetName(string n)
    {
        name = n;
    }
    string GetName()
    {
        return name;
    }
    static int GetMinAge()
    {
        return minAge;
    }
    ...
```



static fields and methods

```
public class Employee
{
    string name;
    int payRate = 0;
    static int minAge = 18;
    …
    void SetName(string n)
    {
        name = n;
    }
    string GetName()
    {
        return name;
    }
    static int GetMinAge()
    {
        return minAge;
    }
    ...
```



static fields and methods exist and can be used without an Employee object being created

static fields and methods

```
public class Whatever
{
    public static void Main(string[] args)
    {
        Employee.minAge = 17;
        Employee x = new Employee();
        Employee y = new Employee();
        x.name = "Fred";
        y.name = "Jane";
        …
        …
        Console.WriteLine(Employee.minAge);
        Console.WriteLine(Employee.name);
```

valid

error!

```
public class Employee
{
    string name;
    int payRate = 0;
    static int minAge = 18;
    …
    void SetName(string n)
    {
        name = n;
    }
    string GetName()
    {
        return name;
    }
    static int GetMinAge()
    {
        return minAge;
    }
    ...
```

static fields and methods

```
public class Employee
{
    string name;
    int payRate = 0;
    static int minAge = 18;
    …
    …
    static string Display()
    {
        string str;
        str = minAge;
        str += "\t";
        str += name;
        return str;
    }
    ...
```

A static method may not refer to an instance field

error!

examples

static

```
double d = Math.Sqrt(5);
double x = Math.Pow(2, 5);
double area = Math.PI * r * r;
```

Sqrt() and Pow()
are static methods
of the Math type

PI is a static field
of the Math type

They are used
without a Math object

instance

```
String s1 = "…
String s2 = "…
int x = s1.Length
String s3 = s1.Substring(0);
```

Length is an instance property
and Substring()
is an instance method
of the String class

They cannot be used
without a String object

Method call summary

A call to a static method defined in another class must always be preceded by the class name.

```
Employee.GetMinAge()
```

An instance method belongs to an object. A call to an instance method in another class must always be preceded by a reference to the object it belongs to.

```
c.GetArea()
```

(where c is a variable storing a Circle object.)

Method call summary

A call to a static method defined in the same class **DOES NOT** need to be preceded by the name of the class.

A call to an instance method defined in the same class **DOES NOT** need to be preceded by a variable that stores an object.

Encapsulation

Encapsulation is one of the fundamental principles of OOP (object-orientated programming).

It states that the fields (state) of an object are accessed **only** by that object.

Nothing outside the object can directly access the fields, but may access them indirectly if the object permits it.

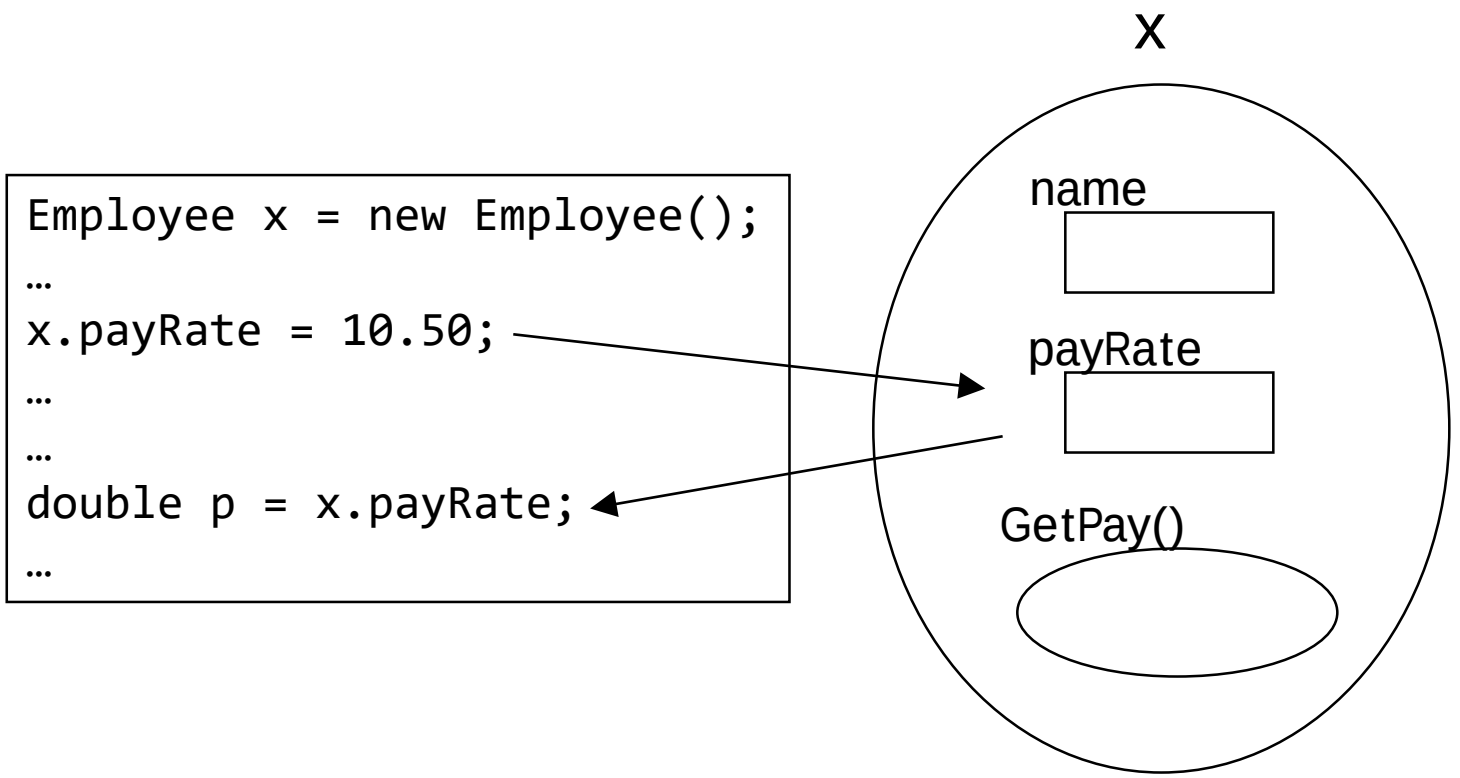
Employee class

```
public class Employee
{
   string name;
   double payRate;
   …
   …
```

```
Employee x = new Employee();
x.name = "Fred";
x.payRate = 10.50;
…
…
Console.WriteLine(x.name + ...
```

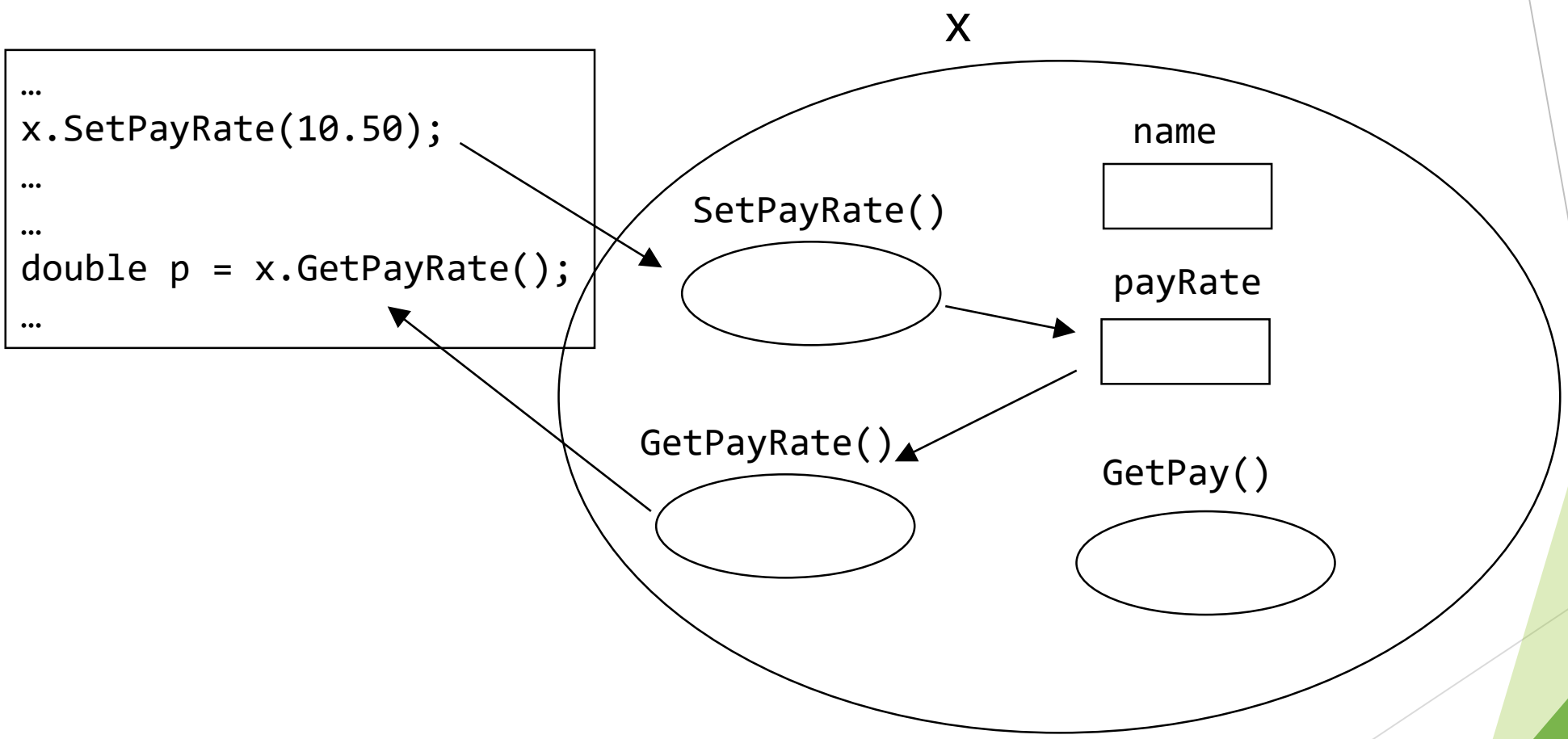
Because we can freely access and modify the *name* and *payRate* fields of an Employee object, this violates the encapsulation principle.

Set and get a field value



Direct access violates the encapsulation principle

Set and get a field value with methods



With encapsulation, only the object's methods can directly access fields

Properties

Properties are special methods in C# called *accessors*. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.

This is exactly what we need for encapsulation.

Encapsulate Employee (with methods)

```
public class Employee
{
   double payRate;
   …
   void SetPayRate(double p)
   {
      payRate = p;
   }
   double GetPayRate()
   {
      return payRate;
   }
   ...
```

```
public static void Main(string[] args)
{
   Employee x = new Employee();
   …
   x.SetPayRate(10.50);
   …
   …
   double p = x.GetPayRate();
   …
   …
```

All fields are accessed indirectly via methods

Encapsulate Employee (with properties)

```
public class Employee
{
   double payRate;
   …
   public double PayRate
   {
      get
      {
         return payRate;
      }
      set
      {
         payRate = value;
      }
   }
   ...
```

```
public static void Main(string[] args)
{
   Employee x = new Employee();
   …
   x.PayRate = 10.50;
   …
   …
   double p = x.PayRate;
   …
   …
```

All fields are accessed indirectly via properties

Encapsulation: Why bother?

One of the biggest problems with large IT systems, is when a change to one small part of the system requires changes to many other parts of the system.

It is often a huge and difficult task to identify and make all the required changes. This task is also prone to many errors, requiring a form of testing known as **regression testing**, further adding to the time and complexity. This is known as the *cost of change*.

Encapsulation: Why bother?

In order to better articulate this *cost of change*, a common term is **technical debt**.

The more poorly designed our code is, the greater the debt we incur. In turn, the *cost of change* goes up.

Encapsulation doesn't solve all of these problems, but it is one of the ways we promote good design in our code.

Encapsulation: Why bother?

Encapsulation promotes information hiding, which means that other parts of our code know a lot less about the internal workings of an object.

With encapsulation, changes can be more easily made to a class (the template of an object), without it affecting the other parts of our code that utilise objects based on that class. In effect, we *reduce the cost of change*.

Example of isolating a change with encapsulation

```
public class Employee
{
   string name;
   double payRate;
   …
   public double PayRate
   {
      get { return payRate; }
      set { payRate = value; }
   }
   ...
```

A decision is made to replace the payRate field in the Employee class with a salary field.

Many other classes in the system access this field by using the PayRate property getter/setter

Example of isolating a change with encapsulation

Old Employee class

```
public class Employee
{
   string name;
   double payRate;
   …
   public double PayRate
   {
      get { return payRate; }
   }
   ...
```

New Employee class

```
public class Employee
{
   string name;
   int salary;
   …
   public double PayRate
   {
      get { return payRate; }
   }
   ...
```

Error

Example of isolating a change with encapsulation

The PayRate property is kept and changed so that it gives the same result as before.

All other classes in the system that use Employee objects by calling the property PayRate are not affected by this change.

New Employee class

```
public class Employee
{
   string name;
   int salary;
   …
   public double PayRate
   {
      get { return (salary / 52.0 / 40.0); }
   }
   ...
```

Access modifier

In order to hide an object's fields and enforce encapsulation, we use the "private" access modifier.

In C# it's often a convention to precede private fields with an underscore.

```
public class Employee
{
      private string _name;
      private double _salary;
      ...
```

Access modifiers

Access modifiers are a way to control how visible, or *accessible*, our code is to other parts of our code.

In C# there are four main access modifiers.

- public
- internal
- protected
- private

Access Modifiers

Access modifiers can be applied to the following in C#.

- Classes
- Fields
- Properties
- Methods

private access modifier

A **private** field, property, or method can be accessed only from within the same class.

- Most fields should be private → to enforce encapsulation
- Some properties/methods can be private → for internal use only
- Some properties/methods must not be private → otherwise there would be no possible use for the class

public access modifier

A **public** field, property, method or class can be accessed from anywhere.

This makes public the most *visible*, or *accessible*, modifier. Unless we intend to hide class members (fields, properties, methods) from other parts of our code, we will generally use the **public** access modifier.

Default (no access modifier)

A class with no access modifier is **internal** by default. This means it can be accessed only by other classes in the same project.

Class members (fields, properties, or methods) with no access modifier are **private** by default.

This is why we've had to type public all this time!

Demonstration

- Static keyword in C#
- What is encapsulation?
- C# access modifiers
  - public
  - private