Classes & Objects

Classes

As we have seen in the previous modules, built-in data types are used to store a single value in a declared variable. For example, **int x** stores an integer value in a variable named **x**.
In object-oriented programming, a **class** is a data type that defines a set of variables and methods for a declared **object**.
For example, if you were to create a program that manages bank accounts, a **BankAccount** class could be used to declare an object that would have all the properties and methods needed for managing an individual bank account, such as a **balance** variable and **Deposit** and **Withdrawal** methods.

A class is like a **blueprint**. It defines the data and behavior for a type. A class definition starts with the keyword **class** followed by the class name. The class body contains the data and actions enclosed by curly braces.

```
class BankAccount
{
  //variables, methods, etc.
}
```

> The class defines a data type for objects, but it is not an object itself. An **object** is a concrete entity based on a class, and is sometimes referred to as an **instance of a class**.

Objects

Just as a built-in data type is used to declare multiple variables, a class can be used to declare multiple **objects**. As an analogy, in preparation for a new building, the architect designs a blueprint, which is used as a basis for actually building the structure. That same blueprint can be used to create multiple buildings.
Programming works in the same fashion. We define (design) a class that is the blueprint for creating objects.
In programming, the term **type** is used to refer to a class **name**: We're creating an object of a particular **type**.

Once we've written the class, we can create objects based on that class. Creating an object is called **instantiation**.

> An object is called an instance of a class.

Objects

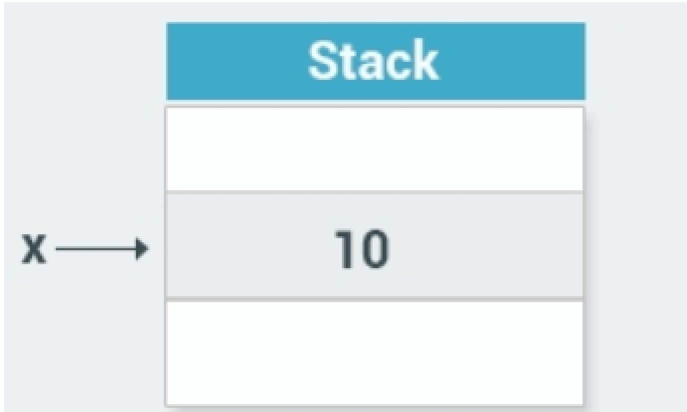
Each object has its own characteristics. Just as a person is distinguished by name, age, and gender, an object has its own set of values that differentiate it from another object of the same type.
The characteristics of an object are called **properties**.
Values of these properties describe the current state of an object. For example, a Person (an object of the class Person) can be 30 years old, male, and named Antonio.

Objects aren't always representative of just physical characteristics.
For example, a programming object can represent a date, a time, and a bank account. A bank account is not tangible; you can't see it or touch it, but it's still a well-defined object because it has its own properties.

> Let's move on and see how to create your own custom classes and objects!

Value Types

C# has two ways of storing data: by **reference** and by **value**.
The built-in data types, such as int and double, are used to declare variables that are **value types**.
Their value is stored in memory in a location called the **stack**.
For example, the declaration and assignment statement int **x = 10;** can be thought of as:



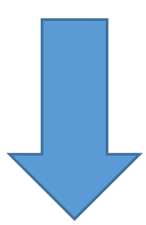
> The value of the variable x is now stored on the **stack**.

Reference Types

Reference types are used for storing objects. For example, when you create an object of a class, it is stored as a reference type.
Reference types are stored in a part of the memory called the **heap**.
When you instantiate an object, the data for that object is stored on the heap, while its heap memory address is stored on the stack.
That is why it is called a reference type - it contains a reference (the memory address) to the actual object on the heap.

As you can see, the **p1** object of type Person on the stack stores the memory address of the heap where the actual object is stored.

> **Stack** is used for static memory allocation, which includes all your value types, like x.
> **Heap** is used for dynamic memory allocation, which includes custom objects, that might need additional memory during the runtime of your program.



Example of a Class

Let's create a **Person** class:

```
class Person
{
 int age;
 string name;
 public void SayHi()
 {
  Console.WriteLine("Hi");
 }
}
```

The code above declares a class named **Person**, which has **age** and **name** fields as well as a **SayHi** method that displays a greeting to the screen.
You can include an **access modifier** for fields and methods (also called **members**) of a class.
Access modifiers are keywords used to specify the accessibility of a member.
A member that has been defined **public** can be accessed from outside the class, as long as it's anywhere within the scope of the class object. That is why our **SayHi** method is declared **public**, as we are going to call it from outside of the class.

> You can also designate class members as **private** or **protected**. This will be discussed in greater detail later in the course. If no access modifier is defined, the member is **private** by default.

Example of a Class

Now that we have our Person class defined, we can instantiate an object of that type in Main.
The **new** operator instantiates an object and returns a reference to its location:

```
class Person {
 int age;
 string name;
 public void SayHi() {
  Console.WriteLine("Hi");
 }
}
static void Main(string[] args)
{
 Person p1 = new Person();
 p1.SayHi();
}
//Outputs "Hi"
```

Try It Yourself

The code above declares a Person object named **p1** and then calls its public **SayHi()** method.

> Notice the **dot operator** (.) that is used to access and call the method of the object.

Example of a Class

You can access all public members of a class using the dot operator.
Besides calling a method, you can use the dot operator to make an assignment when valid.
For example:

```
class Dog
{
  public string name;
  public int age;
}

static void Main(string[] args)
{
  Dog bob = new Dog();
  bob.name = "Bobby";
  bob.age = 3;

  Console.WriteLine(bob.age);
  //Outputs 3
}
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Encapsulation

Part of the meaning of the word **encapsulation** is the idea of "surrounding" an entity, not just to keep what's inside together, but also to protect it.
In programming, encapsulation means more than simply combining members together within a class; it also means restricting access to the inner workings of that class.
Encapsulation is implemented by using **access modifiers**. An access modifier defines the scope and visibility of a class member.

Encapsulation is also called **information hiding**.

Encapsulation

C# supports the following access modifiers: **public**, **private**, **protected**, **internal**, **protected** **internal**.
As seen in the previous examples, the **public** access modifier makes the member accessible from the outside of the class.
The private access modifier makes members accessible only from within the class and hides them from the outside.

protected will be discussed later in the course.

Encapsulation

To show encapsulation in action, let's consider the following example:

```
class BankAccount
{
  private double balance=0;
  public void Deposit(double n)
  {
    balance += n;
  }
  public void Withdraw(double n)
  {
    balance -= n;
  }
  public double GetBalance()
  {
    return balance;
  }
}
```

Try It Yourself

We used encapsulation to hide the **balance** member from the outside code. Then we provided restricted access to it using public methods. The class data can be read through the **GetBalance** method and modified only through the **Deposit** and **Withdraw** methods.
You cannot directly change the **balance** variable. You can only view its value using the public method. This helps maintain data integrity.
We could add different verification and checking mechanisms to the methods to provide additional security and prevent errors.

> In summary, the benefits of encapsulation are:
> - Control the way data is accessed or modified.
> - Code is more flexible and easy to change with new requirements.
> - Change one part of code without affecting other parts of code.

Constructors

A class constructor is a special member method of a class that is executed whenever a new object of that class is created.
A constructor has exactly the same name as its class, is public, and does not have any return type.
For example:

```
class Person
{
  private int age;
  public Person()
  {
    Console.WriteLine("Hi there");
  }
}
```

Now, upon the creation of an object of type Person, the constructor is automatically called.

```
static void Main(string[] args)
{
  Person p = new Person();
}
// Outputs "Hi there"
```

Try It Yourself

> This can be useful in a number of situations. For example, when creating an object of type BankAccount, you could send an email notification to the owner.
> The same functionality could be achieved using a separate public method. The advantage of the constructor is that it is called automatically.

Constructors

Constructors can be very useful for setting initial values for certain member variables.
A default constructor has no parameters. However, when needed, parameters can be added to a constructor. This makes it possible to assign an initial value to an object when it's created, as shown in the following example:

```
class Person
{
  private int age;
  private string name;
  public Person(string nm)
  {
    name = nm;
  }
  public string getName()
  {
    return name;
  }
}
static void Main(string[] args)
 {
  Person p = new Person("David");
  Console.WriteLine(p.getName());
}
//Outputs "David"
```

Try It Yourself

Now, when the object is created, we can pass a parameter that will be assigned to the **name** variable.

> Constructors can be **overloaded** like any method by using different numbers of parameters.

Properties

As we have seen in the previous lessons, it is a good practice to encapsulate members of a class and provide access to them only through public methods.
A **property** is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they actually include special methods called **accessors**.
The accessor of a property contains the executable statements that help in getting (reading or computing) or setting (writing) a corresponding field. Accessor declarations can include a **get** accessor, a **set** accessor, or both.
For example:

```
class Person
{
  private string name; //field

  public string Name //property
  {
    get { return name; }
    set { name = value; }
  }
}
```

The Person class has a **Name** property that has both the **set** and the **get** accessors.
The set accessor is used to assign a value to the name variable; get is used to return its value.

> **value** is a special keyword, which represents the value we assign to a property using the **set** accessor.
> The name of the property can be anything you want, but coding conventions dictate properties have the same name as the private field with a capital letter.

Properties

Once the property is defined, we can use it to assign and read the private member:

```
class Person
{
  private string name;
  public string Name
  {
    get { return name; }
    set { name = value; }
  }
}
static void Main(string[] args)
{
  Person p = new Person();
  p.Name = "Bob";
  Console.WriteLine(p.Name);
}
```

Try It Yourself

> The property is accessed by its name, just like any other public member of the class.

Properties

Any accessor of a property can be omitted.
For example, the following code creates a property that is read-only:

```
class Person
{
 private string name;
 public string Name
 {
  get { return name; }
 }
}
```

A property can also be **private**, so it can be called only from within the class.

Properties

So, why use properties? Why not just declare the member variable public and access it directly?
With properties you have the option to control the logic of accessing the variable.
For example, you can check if the value of **age** is greater than 0, before assigning it to the variable:

```
class Person
{
 private int age=0;
 public int Age
 {
  get { return age; }
  set {
   if (value > 0)
    age = value;
  }
 }
}
```

You can have any custom logic with **get** and **set** accessors.

Auto-Implemented Properties

When you do not need any custom logic, C# provides a fast and effective mechanism for declaring private members through their properties.
For example, to create a private member that can only be accessed through the **Name** property's **get** and **set** accessors, use the following syntax:

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

As you can see, you do not need to declare the private field name separately - it is created by the property automatically. **Name** is called an **auto-implemented property**. Also called auto-properties, they allow for easy and short declaration of private members.
We can rewrite the code from our previous example using an auto-property:

```
class Person
{
  public string Name { get; set; }
}
static void Main(string[] args)
{
  Person p = new Person();
  p.Name = "Bob";
  Console.WriteLine(p.Name);
}
// Outputs "Bob"
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

End.