Clases

Como hemos visto en los módulos previos, los tipos de datos incorporados son utilizados para almacenar un sólo valor en una variable declarada. Por ejemplo, **int x** almacena un valor entero en una variable llamada **x**.
En programación orientada a objetos, una **clase** es un tipo de dato que define un conjunto de variables y métodos para un **objeto** declarado.
Por ejemplo, si fueras a crear un programa que administre cuentas bancarias, una clase **BankAccount** podría ser utilizada para declarar un objeto que tendría todas las propiedades y métodos necesarios para administrar una cuenta bancaria individual, tales como una variable **balance** y métodos para **Depositar** y **Retirar**.

Una clase es como un **plano de diseño**. Define el tipo de dato y comportamiento para un tipo.
Una definición de clase comienza con la palabra clave **class** seguida por el nombre de la clase.
El cuerpo de la clase contiene los datos y acciones encerradas dentro de llaves.

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

La clase define un tipo de dato para los objetos, pero no es un objeto en sí. Un **objeto** es una entidad concreta basada sobre una clase, y a veces es referido como una **instancia de una clase**.

Objetos

Al igual que un tipo de dato incorporado es utilizado para declarar múltiples variables, una clase puede ser usada para declarar múltiples **objetos**. Como analogía, en los preparativos para un nuevo edificio, el arquitecto diseña unos planos, los cuales son utilizados como una base para realmente construir la estructura. Ese mismo plano puede ser utilizado para crear varios edificios.
La programación funciona de la misma forma. Definimos (diseñamos) una clase que es el plano para crear objetos.
En programación, el término **tipo** es utilizado para referirse al **nombre** de una clase: Estamos creando un objeto de un **tipo** en particular.

Una vez que hemos escrito la clase, podemos crear objetos basados en esa clase. Crear un objeto es llamado **instanciar**.

Cada objeto tiene sus propias características. Así como una persona se distingue por su nombre, edad y género, un objeto tiene su propio conjunto de valores que lo diferencian de otro objeto del mismo tipo.
Las características de un objeto son llamadas **propiedades**.
Los valores de estas propiedades describen el estado actual de un objeto. Por ejemplo, una Persona (un objeto de la clase Persona) puede ser de 30 años, masculino y llamarse Antonio.

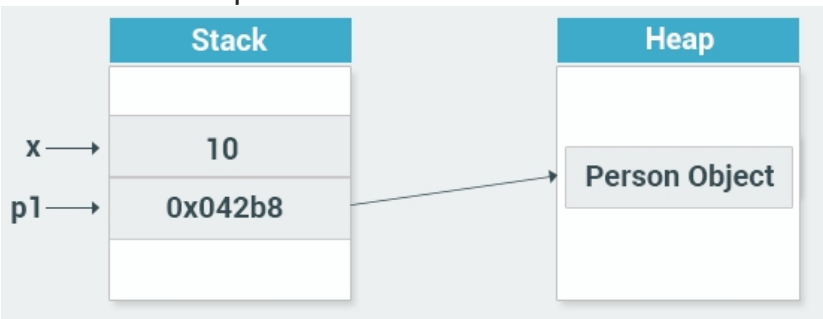
Los objetos no siempre representan sólo características físicas.
Por ejemplo, un objeto en programación puede representar una fecha, una hora y una cuenta bancaria. Una cuenta bancaria no es tangible; no puedes verla o tocarla, pero sigue siendo un objeto bien definido porque tiene sus propias propiedades.

Tipos por valor

C# tiene dos formas de almacenar datos: por **referencia** y por **valor**.
Los tipos de datos incorporados, como int y double, son utilizados para declarar variables que son tipos **de valor**. Su valor está almacenado en memoria, en una ubicación llamada el **stack**.
Por ejemplo, la declaración y asignación **int x = 10;** puede ser pensada como:

Tipos referenciados

Los tipos referenciados son utilizados para almacenar objetos. Por ejemplo, cuando creas un objeto de una clase, es almacenado como un tipo referenciado.
Los tipos referenciados son almacenados en una parte de la memoria llamada el **heap**.
Cuando instancias un objeto, los datos para ese objeto son almacenado en el heap, mientras que la ubicación de memoria del heap es almacenada en el stack.
Es por esto que es llamado un tipo referenciado - contiene una referencia (la dirección de memoria) al objeto actual en el heap.



Ejemplo de una clase

Vamos a crear una clase **Person**:

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

Copiar

El código anterior declara una clase llamada **Person**, la cual tiene campos denominados **age** (edad) y **name** (nombre) así como un método SayHi que despliega un saludo en la pantalla.
Puedes incluir un **modificador de acceso** para campos y métodos (también llamados **miembros**) de una clase. Los modificadores de acceso son palabras claves utilizadas para especificar la

accesibilidad de un miembro.
Un miembro que ha sido definido como **público** puede ser accedido desde fuera de la clase, siempre y cuando esté en cualquier parte dentro del alcance del objeto de la clase. Esta es la razón del por qué nuestro método **SayHi** está declarado **público**, ya que vamos a invocarlo desde afuera de la clase.

También puedes designar a los miembros de clase como **privados** o **protegidos**. Esto será discutido en mayor detalle posteriormente en el curso. Si ningún modificador de acceso está definido, el miembro es **privado** por defecto.

Fíjate que cuando tenemos nuestra clase Person definida, podemos instanciar un objeto de este tipo en Main.
El operador **new** instancia un objeto y retorna una referencia a su ubicación:

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

Puedes acceder a todos los miembros públicos de una clase utilizando el operador punto.
Además de invocar métodos, puedes utilizar el operador punto para hacer una asignación cuando sea válido.

```
class Program
    {
        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);
        }
    }
```

Encapsulación

Parte del significado de la palabra **encapsulación** es la idea de "rodear" una entidad, no sólo para mantener junto lo que está adentro, pero también para protegerlo.
En programación, la encapsulación significa más que simplemente combinar miembros dentro de una clase; también significa restringir acceso a las funciones internas de esa clase.

La encapsulación es implementada utilizando **modificadores de acceso**. Un modificador de acceso define el alcance y visibilidad de un miembro de clase.

La encapsulación es también llamado **ocultamiento de información**.

C# soporta los siguientes modificadores de acceso: "**public**", "**private**", "**protected**", "**internal**", "**protected internal**".
Como se ha visto en los ejemplos anteriores, el modificador de acceso "**public**" (público) hace que el miembro sea accesible desde fuera de la clase.
El modificador de acceso "**private**" (privado) hace que los miembros sean accesibles sólo desde dentro de la clase y los oculta del exterior.

"**protected**" (protegido) será discutido posteriormente en el curso.

Para mostrar la encapsulación en acción, vamos a considerar el siguiente ejemplo:

```
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;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            BankAccount b = new BankAccount();
            b.Deposit(199);
            b.Withdraw(42);
            Console.WriteLine(b.GetBalance());
        }
    }
```

Hemos utilizado encapsulación para ocultar el miembro **balance** del código externo. Luego hemos proveído acceso restringido al mismo utilizando métodos públicos. Los datos de clase pueden ser leídos a través del método **GetBalance** y modificado sólo a través de los métodos **Deposit** y **Withdraw**. No puedes directamente cambiar la variable **balance**. Sólo puedes ver su valor utilizando el método público. Esto ayuda a mantener la integridad de los datos.
Podemos añadir diferentes mecanismos de verificación y validación a los métodos para proveer seguridad adicional y prevenir errores.

En resumen, los beneficios de la encapsulación son:
- Controlar la manera en que los datos son accedidos o modificados.
- El código es más flexible y fácil de cambiar a partir de nuevos requerimientos.
- Poder modificar una parte del código sin afectar otras partes del mismo.

Constructores

Un constructor de clase en un miembro especial de una clase que es ejecutado cada vez que un nuevo objeto d esa clase es creado.
Un constructor tiene exactamente el mismo nombre que su clase, es público y no tiene ningún tipo de retorno.

Por ejemplo:

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

Ahora, al momento de creación de un objeto del tipo Person, el constructor es automáticamente invocado.

```
class Program
    {
        class Person
        {
            private int age;
            public Person()
            {
                Console.WriteLine("Hi there");
            }
        }
        static void Main(string[] args)
        {
            Person p = new Person();
        }
    }
```

Esto puede ser útil en un diverso número de situaciones. Por ejemplo, cuando se esté creando un objeto del tipo BankAccount, puedes enviar un correo de notificación al dueño.
La misma funcionalidad puede ser lograda utilizando un método público separado. La ventaja del constructor es que es invocado automáticamente.

Los constructores pueden ser muy útiles para fijar los valores iniciales para ciertas variables miembro. Un constructor predeterminado no tiene parámetros. Sin embargo, cuando es necesario, los parámetros pueden ser añadidos a un constructor. Esto permite asigna un valor inicial a un objeto cuando es creado, como se muestra en el siguiente ejemplo:

```
class Program
    {
        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());
        }
    }
```

Ahora, cuando el objeto es creado, podemos pasar un parámetro que será asignado a la variable name.

Los constructores pueden ser **sobrecargados** como cualquier método utilizando un número diferente de parámetros.

Propiedades

Como hemos visto en las lecciones anteriores, es una buena práctica encapsular los miembros de una clase y proveer acceso a ellos sólo a través de métodos públicos.
Una **propiedad** es un miembro que provee un mecanismo flexible para leer, escribir o computar el valor de un campo privado. Las propiedades pueden ser utilizadas como si fueran miembros públicos de datos, pero realmente incluyen métodos especiales llamados **descriptores de acceso (accessors)**.
El descriptor de acceso de una propiedad contiene las declaraciones ejecutables que ayudan a obtener (leer o computar) o fijar (escribir) un campo correspondiente. Las declaraciones del descriptor de acceso pueden incluir un descriptor de acceso **get**, un descriptor de acceso **set**, o ambas.
Por ejemplo:

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

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

Copiar

La clase Person tiene una propiedad **Name** que tiene tanto el descriptor de acceso **set** como el descriptor de acceso **get**.
El descriptor de acceso set es utilizado para asignar un valor a la variable name; mientras que get es utilizado para retornar su valor.

"**value**" es una palabra clave especial, la cual representa el valor que asignamos a una propiedad utilizando el descriptor de acceso **set**.
El nombre de la propiedad puede ser cualquier cosa que desees, pero las convenciones de codificación dictan que las propiedades tengan el mismo nombre que el campo privado con la primera letra en mayúscula.

Una vez que la propiedad está definida, podemos utilizarla para asignar y leer el miembro privado:

```
class Program
    {
        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);
        }
    }
```

La propiedad es accedida por su nombre, tal cual cualquier otro miembro público de la clase.

Cualquier descriptor de acceso de una propiedad puede ser omitido.
Por ejemplo, el siguiente código crea una propiedad que es sólo lectura:

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

Una propiedad puede también ser **privada**, por lo que sólo podrá ser invocada desde dentro de la clase.

Entonces, ¿para qué utilizamos propiedades? ¿Por qué no simplemente declaramos pública a la variable miembro y la accedemos directamente?
Con propiedades tienes la opción de controlar la lógica de acceso a la variable.
Por ejemplo, puedes validar si el valor de **age** es mayor que 0, antes de asignarlo a la variable:

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

Copiar

Puedes tener cualquier lógica personalizado con los descriptores **get** y **set**.

Propiedades auto-implementadas

Cuando no necesitas ninguna lógica personalizada, C# provee un mecanismo rápido y efectivo para declarar miembros privados a través de sus propiedades.
Por ejemplo, para crear un miembro privado que sólo pueda ser accedido a través de los descriptores de acceso **get** y **set** de la propiedad **Name**, utiliza la siguiente sintaxis:

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

Copiar

Como puedes ver, no necesitas declarar el campo privado "name" por separado - es creado por la propiedad automáticamente. **Name** es llamada una **propiedad auto implementada**. También llamadas propiedades automáticas, permiten una declaración fácil y reducida de miembros privados.
Podemos reescribir el código de nuestro ejemplo anterior utilizando una propiedad automática:

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