Classes and Objects

Pillars of OOP

- Encapsulation
  - Encapsulate the inner details of implementation
  - Protect data
- Inheritance
  - Build new class based on existing class definitions
  - Embody the is-a relationship between types
- Polymorphism
  - Treat the related objects in the same way

Introduction to Classes

- A class is a group of related methods and variables.
- A **class** is the blueprint for an object.
  - It describes a particular type of object, yet it is not an object.
  - It specifies the fields and methods a particular type of object can have.
  - One or more object can be created from the class.
  - Each object created from a class is called an **instance** of the class.

Creating a Class

- You can create a class by writing a **class declaration**. A generic form is:

```
class ClassName  // class header
{
   Member declaration(s)…
}
```

- **Class headers** starts with the keyword *class*, followed by the name of the class.
- **Member declarations** are statements that define the class's fields, properties, and/or methods.
- A class may contains a **constructor**, which is special method automatically executed when an object is created.

Sample

```
class Coin
{
   private string sideUp; // field

   public Coin() // constructor
   {
      sideUp = “Heads”;
   }

   public void Toss() // a void method
   {
      MessageBox.Show(sideUp);
   }

   public string GetSideUp() // a value-returning method
   {
       return sideUp;
   }
}
```

Creating an Object

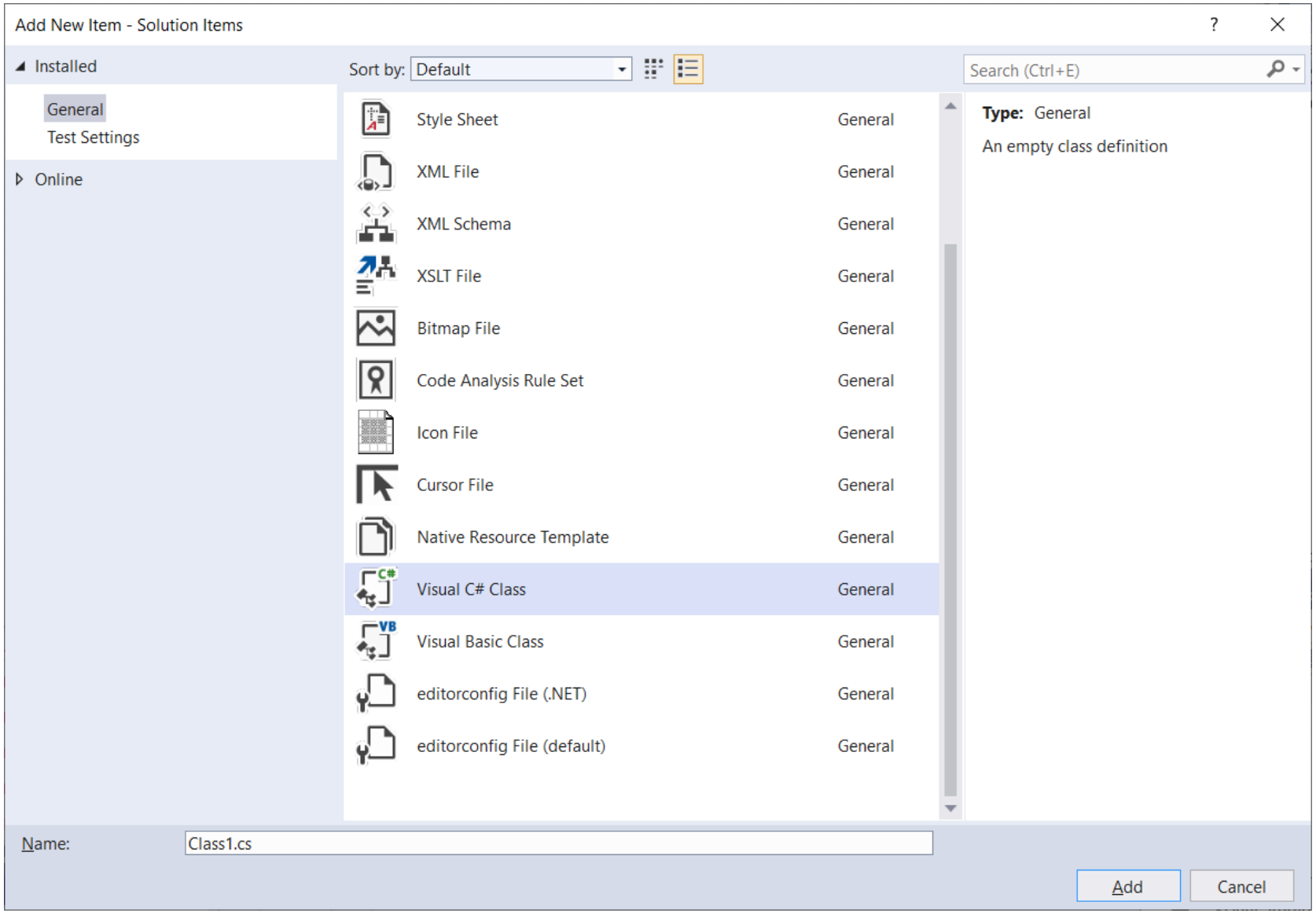
- Given a class named Coin, you can create a Coin object use:

```
Coin myCoin = new Coin();
```

- where,
  - •myCoin is a variable that references an object of the Coin class;
  - •the **new** keyword creates an instance of the Coin class; and
  - •the = operator assigns the reference that was returned from the new operator to the myCoin variable.
- Once a Coin object is created, you can access members of the class with it. E.g.

```
myCoin.Toss();
```

Where to Write Class Declarations



Passing an Object to a Method

- Objects of a class can be used as parameter of a method. E.g.

```
private void ShowCoinStatus(Coin coin)
{
   MessageBox.Show(“Side is “ + coin.GetSideUp());
}
```

- In this example, a method named ShowCoinStatus accepts a Coin object as an argument.
- To create a Coin object and pass it as an argument to the ShowCoinStatus method, use:

```
Coin myCoin = new Coin();
ShowCoinStatus(myCoin);
```

Properties

- A **property** is a class member that holds a piece of data about an object.
  - Properties can be implemented as special methods that set and get the value of corresponding fields.
  - Both **set** and **get** methods are known as **accessors**.
  - In the code, there is a private field (_name) which is a known as **field** and is used to hold any data assigned to the Name property.
  - The **value** parameter of set accessor is automatically created by the compiler.

```
class Pet
{
    private string _name; // field
    public Pet()
    {
        _name = “”;
    }

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

Field

- The **private field** is a variable that stores a value assigned to the property which the backing fields is associated with.
- It is declared to be private to protect it from accidental corruption.
- If a field is public, it can then be accessible directly by code outside the class without the need for accessors.

get vs set Accessors

- The **get** accessor, if not empty, is a method that returns the property's value because it has a **return** statement.
  - It is executed whenever the property is read.
- The **set** accessor, if not empty, gets the value and assigns the value to the property
  - It has an implicit parameter named **value**.
  - It is executed whenever a value is assigned to the property.

Read-Only Properties

- A read-only property can be read, but it cannot be modified.
  - To set a read-only property, simply do no write a set accessor for the property. E.g.

```
// read and write
public double Diameter
{
   get { return _diameter; }
   set { _diameter = value; }
}
```

```
// read
public double Diameter
{
   get { return _diameter; }
}
```

Parameterized Constructor & Overloading

- A constructor that accepts arguments is known as **parameterized constructor**. E.g.

```
public BankAccount(decimal startingBalance) { }
```

- A class can have multiple versions of the same method known as **overloaded methods**.
- How does the compiler know which method to call?
  - Binding relies on the **signature** of a method which consists of the method's name, the data type, and argument kind of the method's parameter. E.g.

```
public BankAccount(decimal startingBalance) { }
public BankAccount(double startingBalance) { }
```

  - The process of matching a method call with the correct method is known as **binding**.

Recall: Overloading Methods

- When a method is overloaded, it means that multiple methods in the same class have the same name but use different types of parameters.

```
public void Deposit(decimal amount) { }
public void Deposit(double amount) { } // overloaded
public void Deposit(int numbers) { } // overloaded
public void Deposit(string names) { } // overloaded
```

Overloading Constructors

- Constructors are special type of methods. They can also be overloaded.

```
public BankAccount() { } // parameterless constructor
public BankAccount(decimal startingBalance) { } // overloaded
public BankAccount(double startingBalance) { } // overloaded
```

  - The parameterless constructor is the default constructor

- Compiler will find the matching constructors automatically. E.g.

```
BankAccount account = new BankAccount();
BankAccount account = new BankAccount(500m);
```

Default Values of the Fields

- **When an object is created all of the fields are initialized with their respective default values in .NET, if they are not explicitly initialized with some other value.**

Type of the Field	Default Value
bool	false
byte	0
char	'\0'
decimal	0.0M
double	0.0D
float	0.0F
int	0
object reference	null

- **Unlike fields, local variables are not initialized with default values when they are declared.**

Field - constants

- The fields, declared as **const** or **readonly** are called **constants**.

```
public class ConstAndReadOnlyExample
{
    public const double PI = 3.14159265358979323385;
    public readonly double Size;
    public ConstAndReadOnlyExample(int size)
    {
          this.Size = size; // Cannot be further modified!
    }
    …
}
```

```
Console.WriteLine(ConstAndReadOnlyExample.PI);
ConstAndReadOnlyExample instance = new ConstAndReadOnlyExample(5);
Console.WriteLine(instance.Size);
```

Reusing Constructors

- Syntax

```
[<modifiers>] <class_name>([<parameters_list_1>])

: this([<parameters_list_2>])
```

```
public class Person
{
    public Person()
    {
    }

    public Person(String name)
        : this(name, 0, 0)
    {
    }

    public Person(String name, int age)
        : this(name, age, 0)
    {
    }

    public Person(String name, int age, float salary)
    {
        this.Name = name;
        this.Age = age;
        this.Salary = salary;
    }


    public String Name { get; set; }
    public int Age { get; set; }
    public float Salary { get; set; }

}
```

Static Members

- Static elements of the class can be used without creating an object of the given class.
- What are they used for, i.e.
    - Math method
    - Instance counter

Naming and Accessibility

- The following recommendations are reasonably common; however, C# does not enforce these rules:
- Identifiers that are *public* should start with a capital letter. This system is known as the *PascalCase* naming scheme (because it was first used in the Pascal language).
- Identifiers that are not *public* (which include local variables) should start with a lowercase letter. This system is known as the *camelCase* naming scheme.

*Some organizations use camelCase only for methods and adopt the convention that private fields are named starting with an initial underscore character, such as *_radius*.

* one exception to this rule: class names should start with a capital letter, and constructors must match the name of their class exactly; therefore, a *private* constructor must start with a capital letter.