Object-Oriented Programming in C#

Tan Cher Wah
cherwah@nus.edu.sg

Classes and Objects

Object Oriented Programming

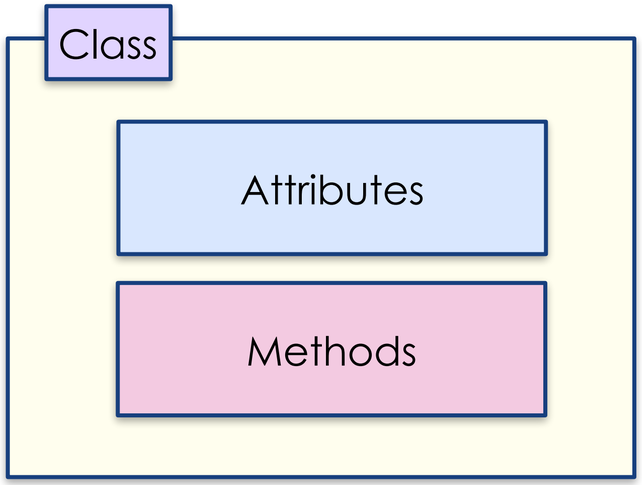
- OOP is a programming paradigm built on the concept of **objects**, instead of functions
- Objects in a OOP program mimic real-world objects
- An **object** comprises of **data** that define its **state** and **methods** that define its **behaviour**
- OOP was first used in writing simulation programs (Simula, 1967)

Class

- Before we can create an Object, we need a Class
- A Class is a **Blueprint** that **models** a real-world entity
- A Class **defines** the

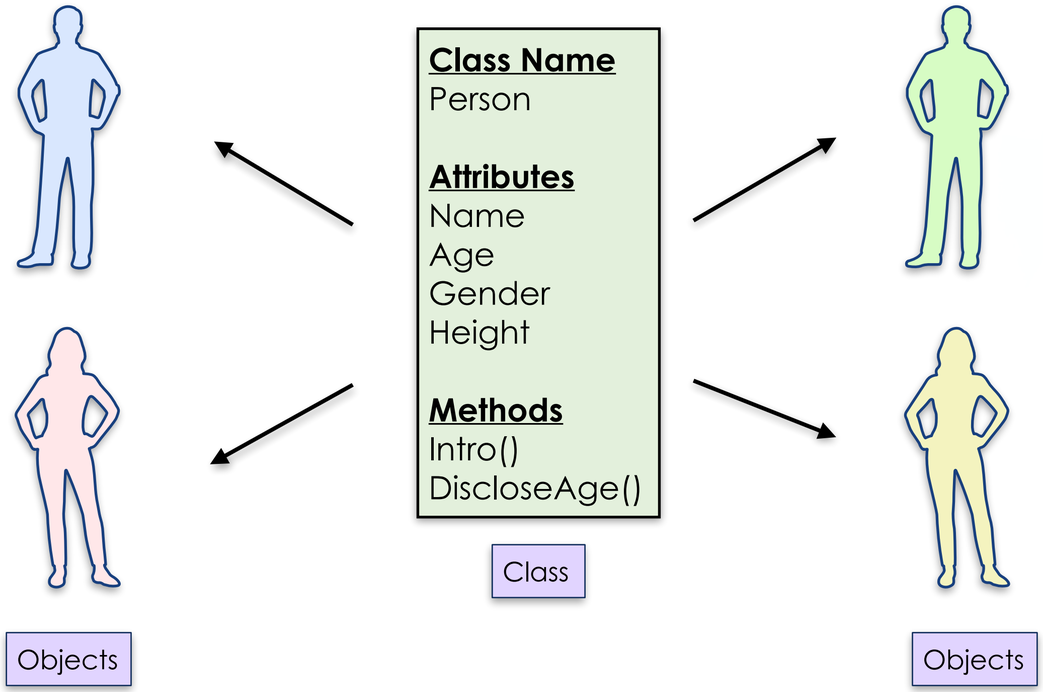
  Attributes (data) of an Object

  Methods (logic) to manipulate those attributes



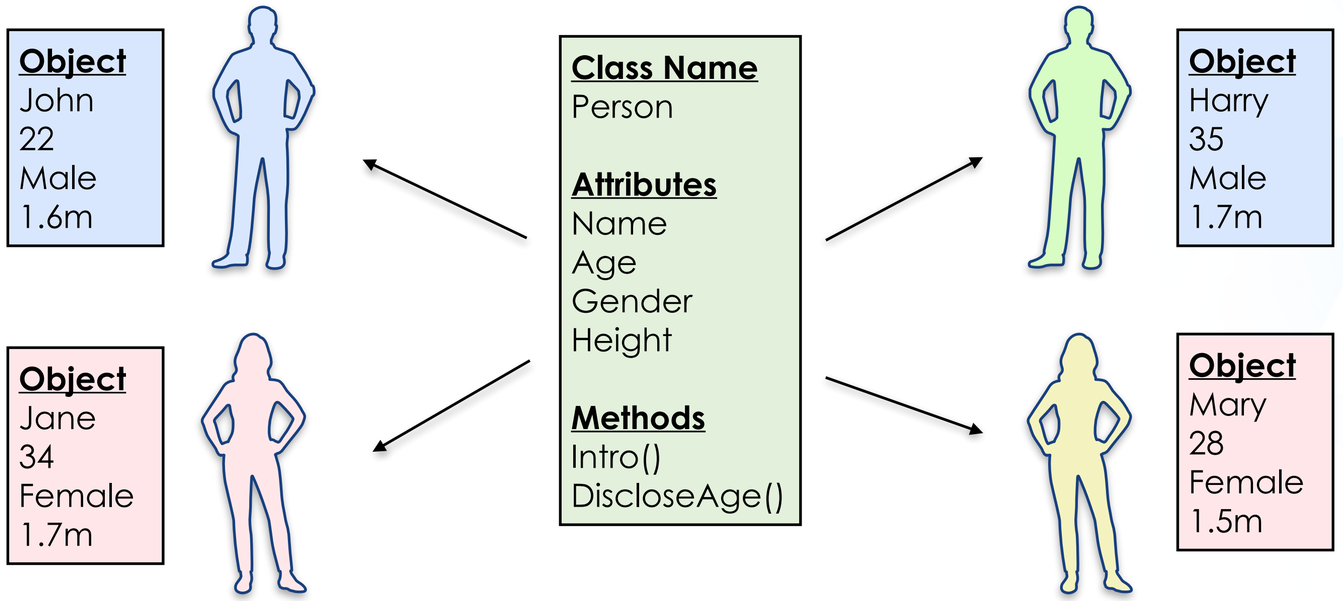
Class as a Blueprint

The Person class defines the **attributes** and **methods** to model a Person



Objects

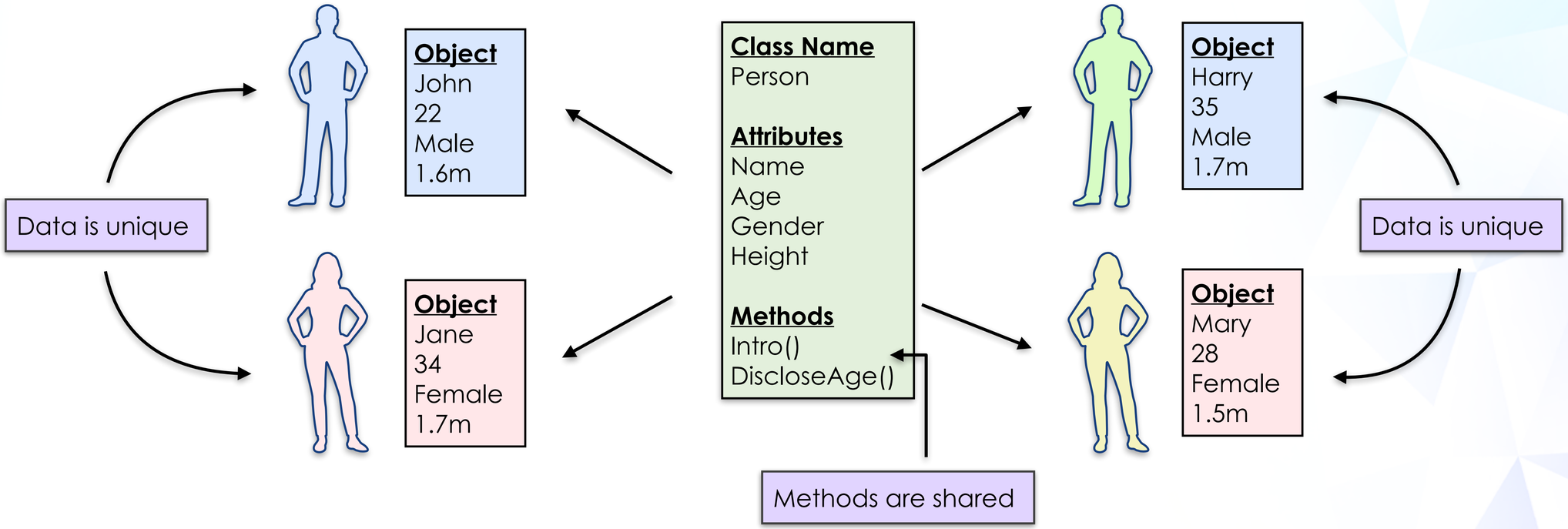
An Object is an **instance** of a Class



John, Jane, Harry and Mary are **objects** of the Person class

Objects

Objects store their own **unique data** but **share** a class's **methods** with other objects



Person class

Creating a Person **class** in C#

```
public class Person {
    private string _name;
    private int _age;

    public Person(string name, int age) {
        _name = name;
        _age = age;
    }

    public void Intro() {
        Console.WriteLine("My name is " + _name + ".");
    }

    public void DiscloseAge() {
        Console.WriteLine("I'm " + _age + " years old.");
    }
}
```

Instance Variables

Code to initialize our person object

Input provided by the user of our class

Methods of our class

Attributes used by the methods of our class

Constructor

The **Constructor** has the **same name** as its **class name** and is used for **initialization of attributes**

A **constructor** has the same name as the **class name**

A constructor **does not return any value**

A constructor is **automatically executed** when an object is **created** from a class

```
public class Person {
    private string _name;
    private int _age;

    public Person(string name, int age) {
        _name = name;
        _age = age;
    }

    . . .
}
```

If no constructor is specified, a public **default constructor**, that **accepts no parameters**, will be automatically created

Create Objects from Class

A non-static Class has to be **instantiated**, via the **new** keyword, before use

The keyword **new** is used to **instantiate** a **class** into an **object**

```
Person harry = new Person("Harry", 23);

harry.Intro();
harry.DiscloseAge();
```

Calling methods in the **created object**

Output

My name is Harry.
I'm 23 years old.

Calling Methods

Calling the same methods on different objects results in different outputs

```
Person harry = new Person("Harry", 23);
harry.Intro();
harry.DiscloseAge();

Person mary = new Person("Mary", 36);
mary.Intro();
mary.DiscloseAge();
```

Creating two different objects from the same class

Same methods called on different objects

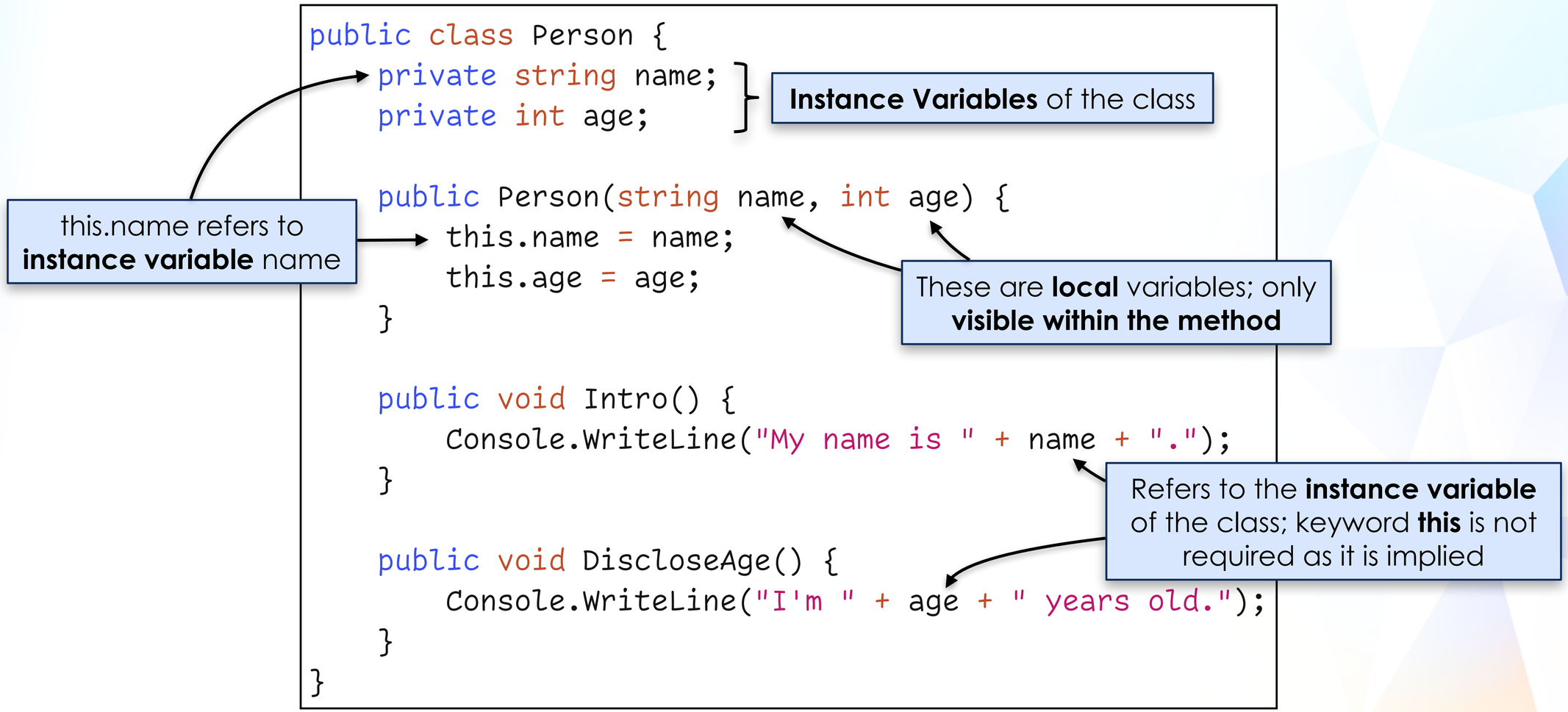
Output

My name is Harry.
I'm 23 years old.

My name is Mary.
I'm 36 years old.

Concept of 'this'

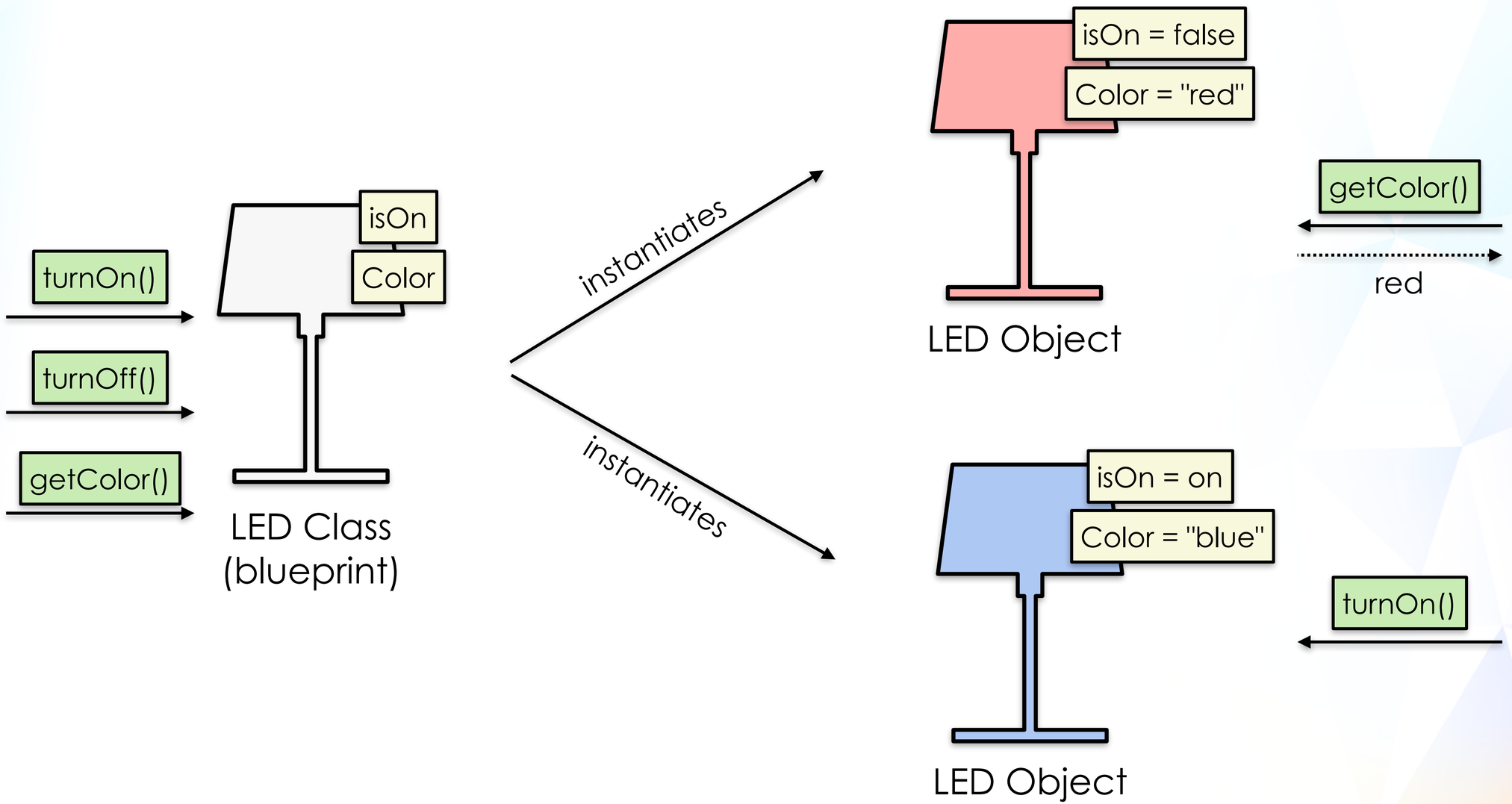
The keyword **this** references the **current object**



Example: Model a LED

- Let's create a C# class to model a LED
- The C# class accepts a string (e.g. “red”) as its Color in its constructor
- The C# class has two attributes
  - Color
  - IsOn (a flag to determine if it is currently ON or OFF)
- It has methods to
  - Turn itself ON or OFF
  - Return its color (as a string)
  - Return a Boolean status to indicate if it is currently ON

Example: Model a LED



Creating the LED class

Designing a LED class

```
namespace LEDProj;

public class LED
{
    private bool isOn;
    public string color;

    public LED(string color) {
        isOn = false;
        this.color = color;
    }

    public string getColor() {
        return color;
    }

    public bool isLEDOn() {
        return isOn;
    }

    public void turnOn() {
        if (!isLEDOn())
            isOn = true;
    }

    public void turnOff() {
        if (isLEDOn())
            isOn = false;
    }
}
```

Constructor initializes its attributes

Method to return its color

Method to indicate if it is currently on

Methods to turn itself ON and OFF

Using the LED class

Making use of our LED class

Instantiate the **class** to create an **object**

```
using LEDProj;

LED redLED = new LED("red");
Console.WriteLine("LED Color: " + redLED.getColor());

redLED.turnOn();
Console.WriteLine("LED is " + (redLED.isLEDOn() ? "ON" : "OFF"));

redLED.turnOff();
Console.WriteLine("LED is " + (redLED.isLEDOn() ? "ON" : "OFF"));
```

Query for the current color of our LED object

Turn our LED object **ON** or **OFF**

If the **expression** is **true**, then "ON" is selected, else "OFF" is selected

Output

```
LED Color: red
LED is ON
LED is OFF
```

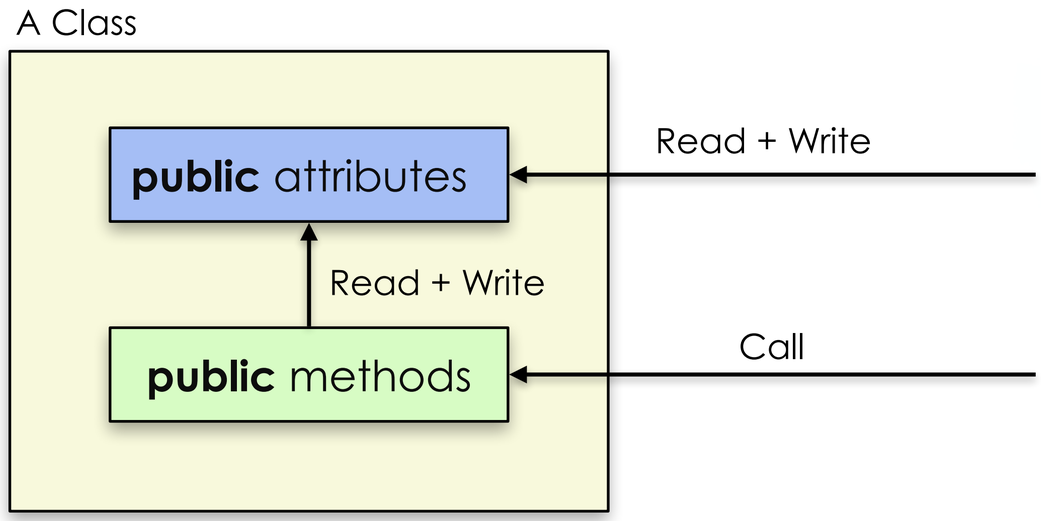
Access Modifiers

Access Modifiers

- Access Modifiers **control external access** to the **attributes** and **methods** of a Class
- Access Modifiers **has no effects within** a Class
- Access Modifiers
  - **Public**
  - **Private**
  - **Protected**

Public Access Modifiers

A **public** modifier allows external access with no restrictions



Public Attributes

A **public** modifier on a class's **attribute** denotes that it can be **read** and **updated** by code that is **outside the class**

The **attribute** *count* is directly **written** from outside the class because it has a **public** modifier

The **modifier** is **public**, hence the **attribute** can be accessed by all

```
public class MyClass
{
    public int count;

    public MyClass(int count)
    {
        this.count = count;
    }
}
```

What is the **value** of count **member attribute** before executing the line of code?

```
class Program
{
    static void Main(string[] args)
    {
        MyClass mc = new MyClass(10);
        mc.count = 20;

        Console.WriteLine("Count = " + mc.count);
    }
}
```

Output

Count = 20

Public Methods

A **public** modifier on a class's **methods** denotes that they can be **invoked** by code that is **outside the class**

```
public class MyClass
{
    public int count;

    public MyClass(int count)
    {
        this.count = count;
    }

    public int GetCount()
    {
        return count;
    }
}
```

Uses a **public** modifier, hence the **method** can be accessed by all

```
class Program
{
    static void Main(string[] args)
    {
        MyClass mc = new MyClass(10);
        mc.count = 20;

        Console.WriteLine("Count = " + mc.GetCount());
    }
}
```

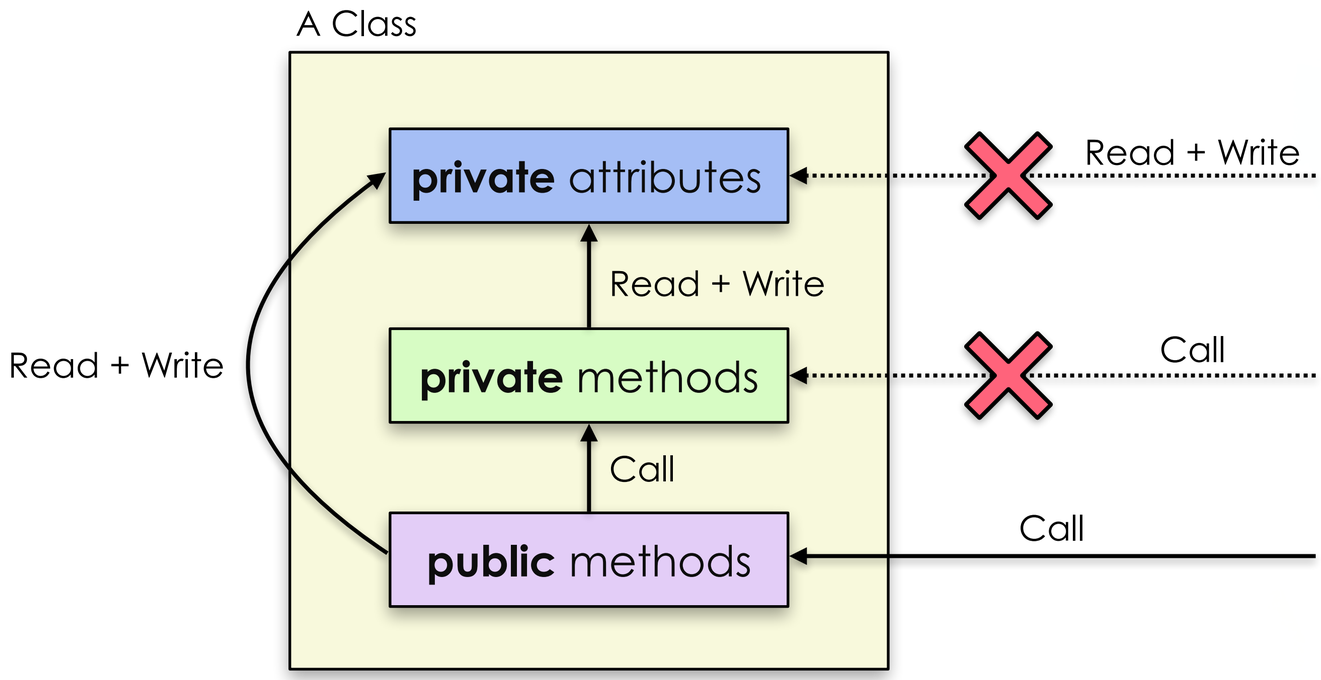
The **method** can be called from **outside the class**

Output

Count = 20

Private Access Modifiers

A **private** modifier disallows any external access



Private Attributes

A **private** modifier on a class's **attributes** denotes that they can only be accessed by code **within the class**

```
public class MyClass
{
    private int count;

    public MyClass(int count)
    {
        this.count = count;
    }

    public int GetCount()
    {
        return count;
    }

    public void SetCount(int count)
    {
        this.count = count;
    }
}
```

The **attribute** *count* is no longer accessible from outside the class due to the **private** modifier

What is the effect of changing this **modifier** to **private**?

```
class Program
{
    static void Main(string[] args)
    {
        MyClass mc = new MyClass(10);
        mc.SetCount(20);

        Console.WriteLine("Count = " + mc.GetCount());
    }
}
```

Output

Count = 20

Use a **public** method to access the **private field** of a class instead

This gives us **control** over how the **field** can be **updated**

Private Methods

A **private** modifier on a **method** denotes that it can only be invoked by code **within the class**

```
public class MyClass {
    private int count;

    public MyClass(int count) {
        this.count = count;
    }

    public int GetCount() {
        return count;
    }

     public void SetCount(int count) {
        if (count == 0)
            Reset();
        else
            this.count = count;
    }

    private void Reset() {
        this.count = 0;
    }
}
```

Even though the method *Reset()* has been declared **private**, it can be accessed by code **within the class** itself

The method *Reset()* is **private**, hence it cannot be invoked/ called from the outside

```
class Program
{
    static void Main(string[] args)
    {
        MyClass mc = new MyClass(10);
        mc.SetCount(0);

        Console.WriteLine("Count = " + mc.GetCount());
    }
}
```

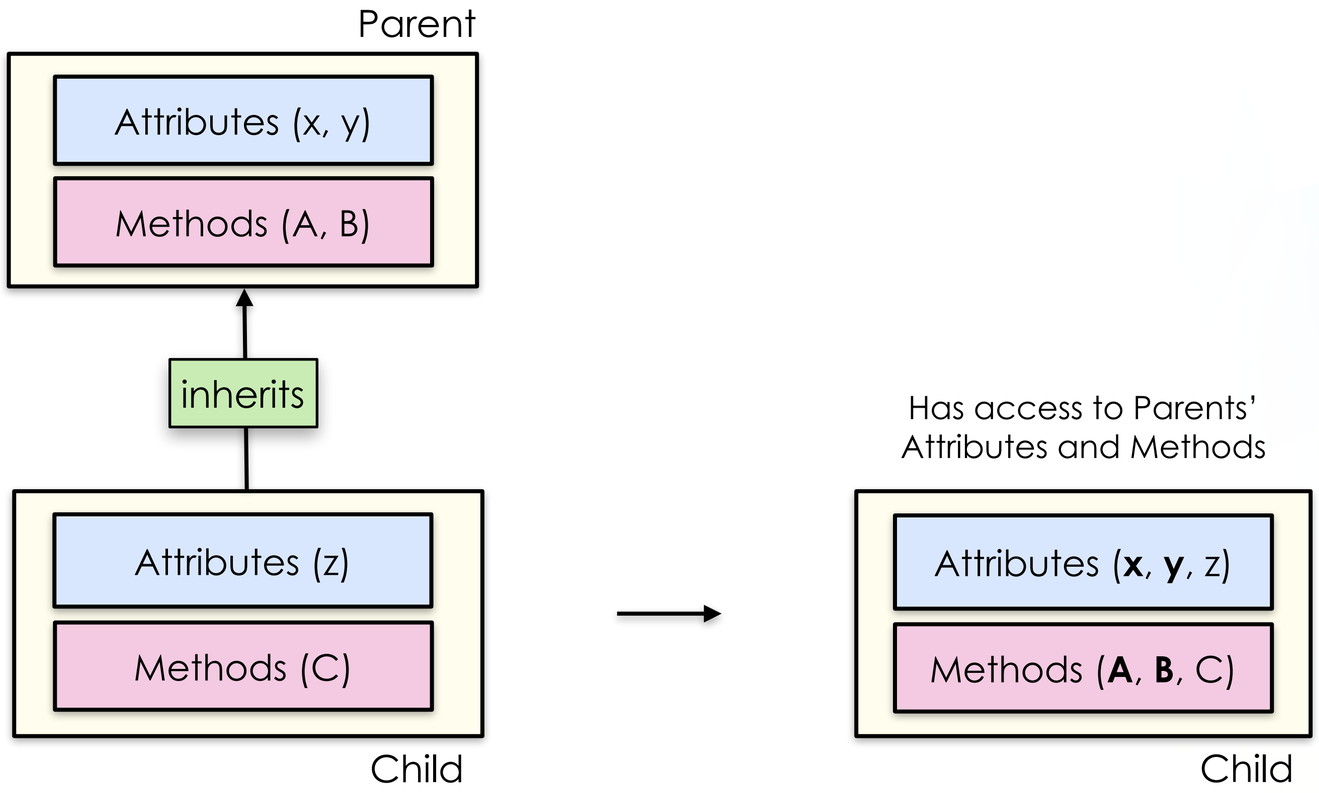
The code in Main(…) can only invoke **public methods**

Output

Count = 0

Protected Access Modifier (only applies for Inheritance)

A Class can inherit another class's **attributes** and **methods** in Object-Oriented Programming



Inheritance in code

AChildClass now has **fields** x, y, z and **methods** A(), B(), C()

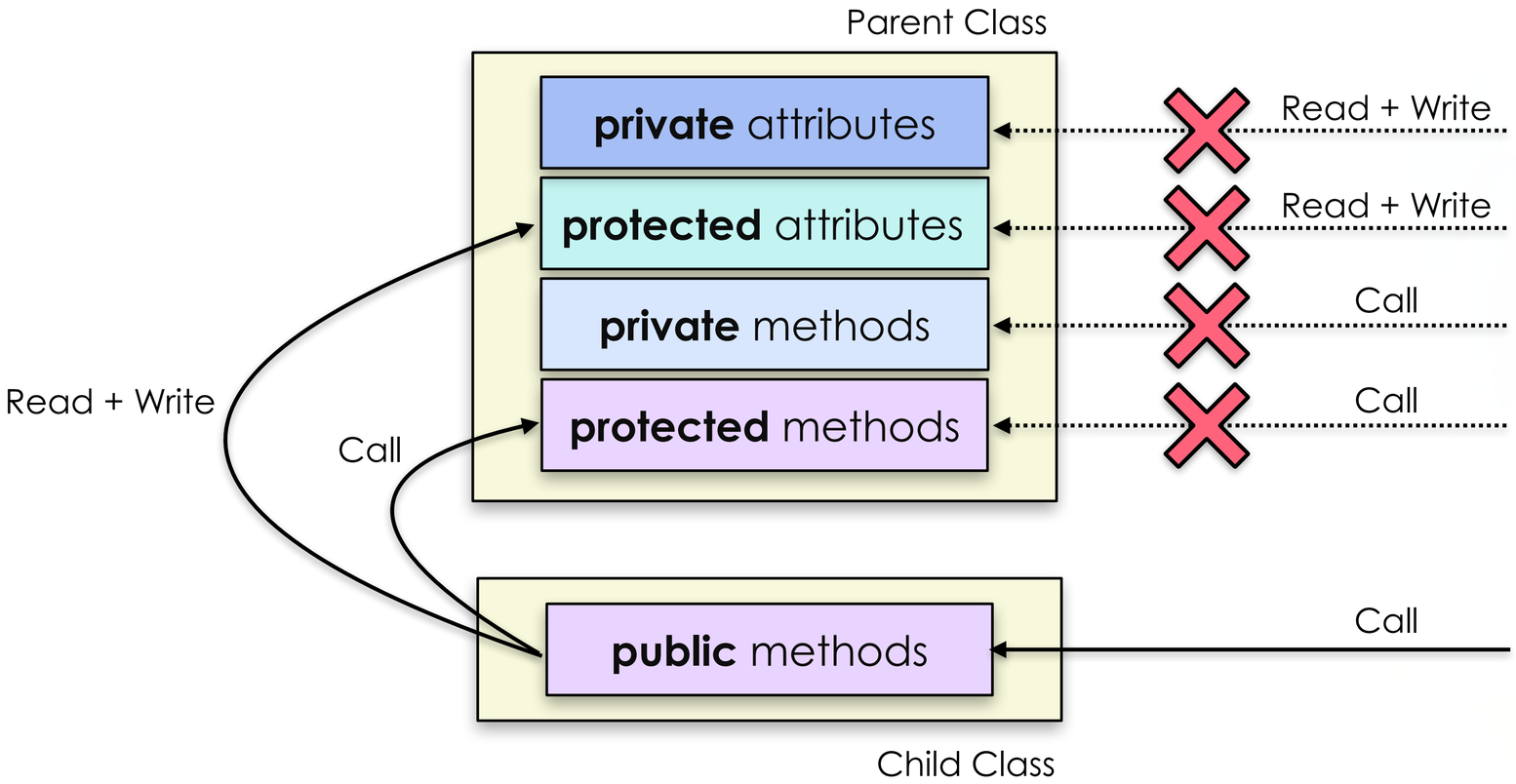
AChildClass inherits **attributes** and **methods** of AParentClass

```
public class AChildClass : AParentClass
{
    public int z;

    public AChildClass()
    {
    }

    public void C()
    {
    }
}
```

```
public class AParentClass
{
    public int x, y;

    public AParentClass()
    {
    }

    public void A()
    {
    }

    public void B()
    {
    }
}
```

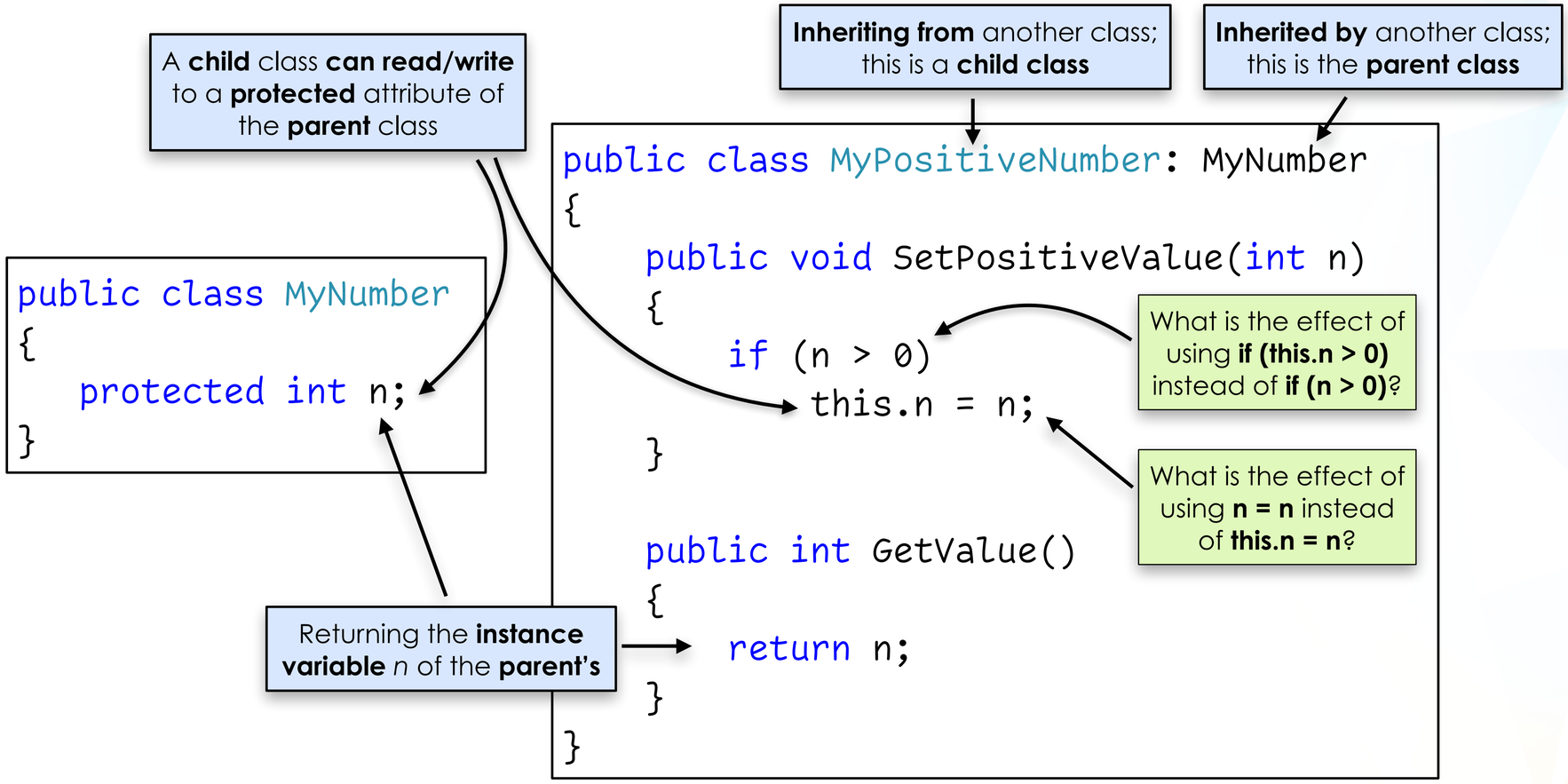
Protected Access Modifiers

A **protected** modifier allows access by its childs, but disallows external access



Protected Attributes

A child class can access its parents' non-private (i.e. **public** and **protected**) attributes and methods



Protected Methods

Protected methods of a class can be invoked by **itself** or its **child classes**

A **child** class **cannot read/write** to a **private** attribute of the **parent** class; read/write a **private attribute** via **protected method(s)** instead

```
class MyNumber
{
    private int n;

    protected void SetValue(int n)
    {
        this.n = n;
    }

    protected int GetValue()
    {
        return n;
    }
}
```

```
class MyPositiveNumber: MyNumber
{
    public void SetPositiveValue(int n)
    {
        if (n > 0)
            SetValue(n);
    }
}
```

The **variable** n is **local** to this method

Protected Modifier

Protected attributes and methods of a class are **not accessible** by the **external** world

```
class MyPositiveNumber: MyNumber {
    public void SetPositiveValue(int n) {
        if (n > 0)
            SetValue(n);
    }

    public int GetPositiveValue() {
        return GetValue();
    }
}
```

Only the child class can call the parent's SetValue(...) method

```
class MyNumber {
    private int n;

    protected void SetValue(int n) {
        this.n = n;
    }

    protected int GetValue() {
        return n;
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        MyPositiveNumber pn = new MyPositiveNumber();
        pn.SetPositiveValue(10);
        pn.SetPositiveValue(-10);

        Console.WriteLine("Count = " + pn.GetPositiveValue());
    }
}
```

A value of -10 is set here, **why** is the output 10 instead of -10?

Why did we not call GetValue() directly?

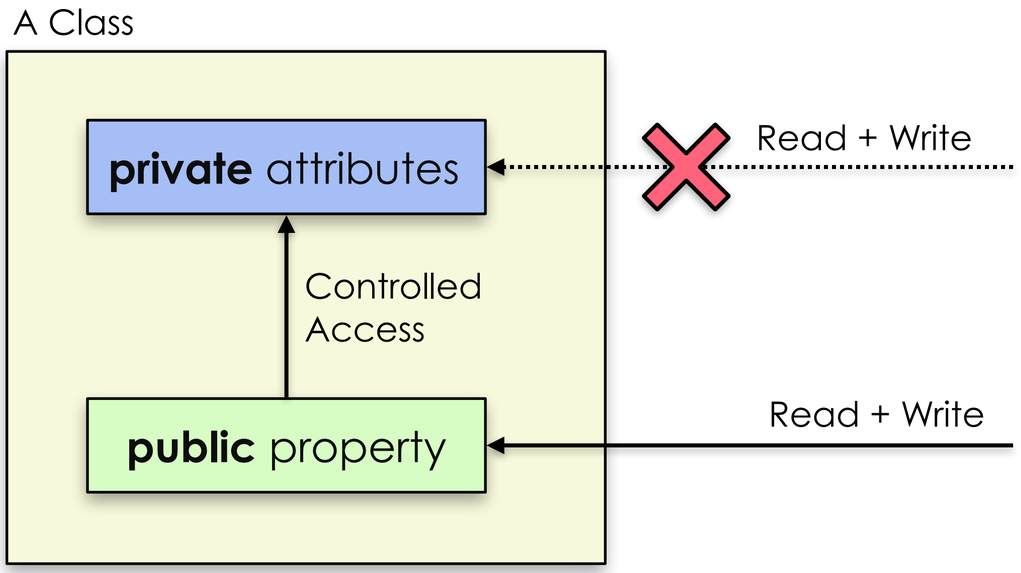
Output

Count = 10

C# Property

Property

A C# **property** is a **public** member of a class that provides controlled access to read, write, validate and compute the value of a **private field**



Without using Property

An (private) **attribute** is often updated via public **getter** and **setter** methods

The **attribute** *n* is read and updated via the class's Read_n and Write_n methods

The class's **getter** for the attribute *n*

The class's **setter** for the attribute *n*

```
class MyNumber
{
    private int n;

    public int Read_n()
    {
        return n;
    }

    public void Write_n(int n)
    {
        this.n = n;
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        MyNumber num = new MyNumber();
        num.Write_n(10);

        Console.WriteLine("Value = " + num.Read_n());
    }
}
```

Use of **setter** to set value to 10

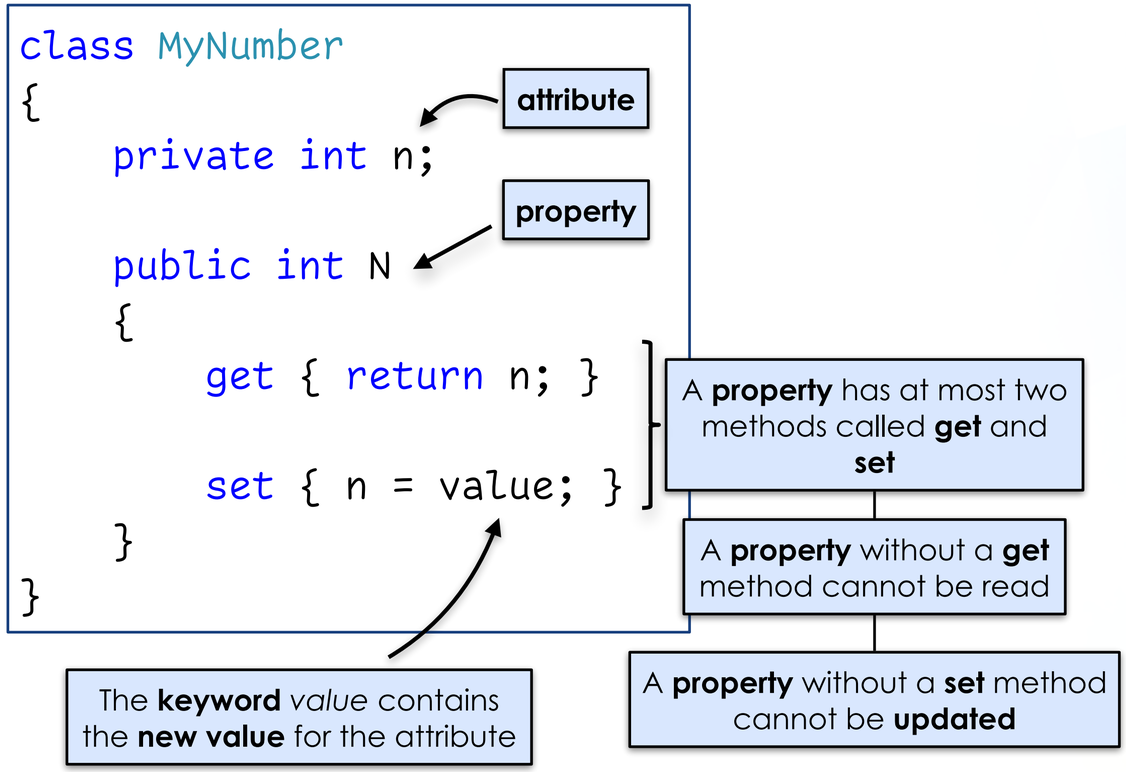
Use of **getter** to get back value

Output

Count = 10

Property

A **Property** exposes an **attribute** to the outside world; a **private attribute** is updated or retrieved via a **(public) property**



Property

A Property can be accessed via the **dot notation**

```
class Program
{
    static void Main(string[] args)
    {
        MyNumber num = new MyNumber();
        num.N = 10;

        Console.WriteLine("Value = " + num.N);
    }
}
```

Calling the property's **set** method to update **private** field n

Calling the property's **get** method to read **private** attribute *n*

Output

Value = 10

Property

The **get** and **set** methods can hold logic to **safeguard the integrity** of our objects

```
class MyEvenNumber
{
    private int n;

    public int N
    {
        get { return n; }

        set
        {
            if (value % 2 == 0)
                n = value;
        }
    }
}
```

The **keyword** *value* holds the new value **assigned** to the **property**

Ability to enforce **logic** to **protect the integrity** of our **fields** (or states)

```
class Program
{
    static void Main(string[] args)
    {
        MyEvenNumber even = new MyEvenNumber();
        even.N = 2;
        even.N = 1;

        Console.WriteLine("Value = " + even.N);
    }
}
```

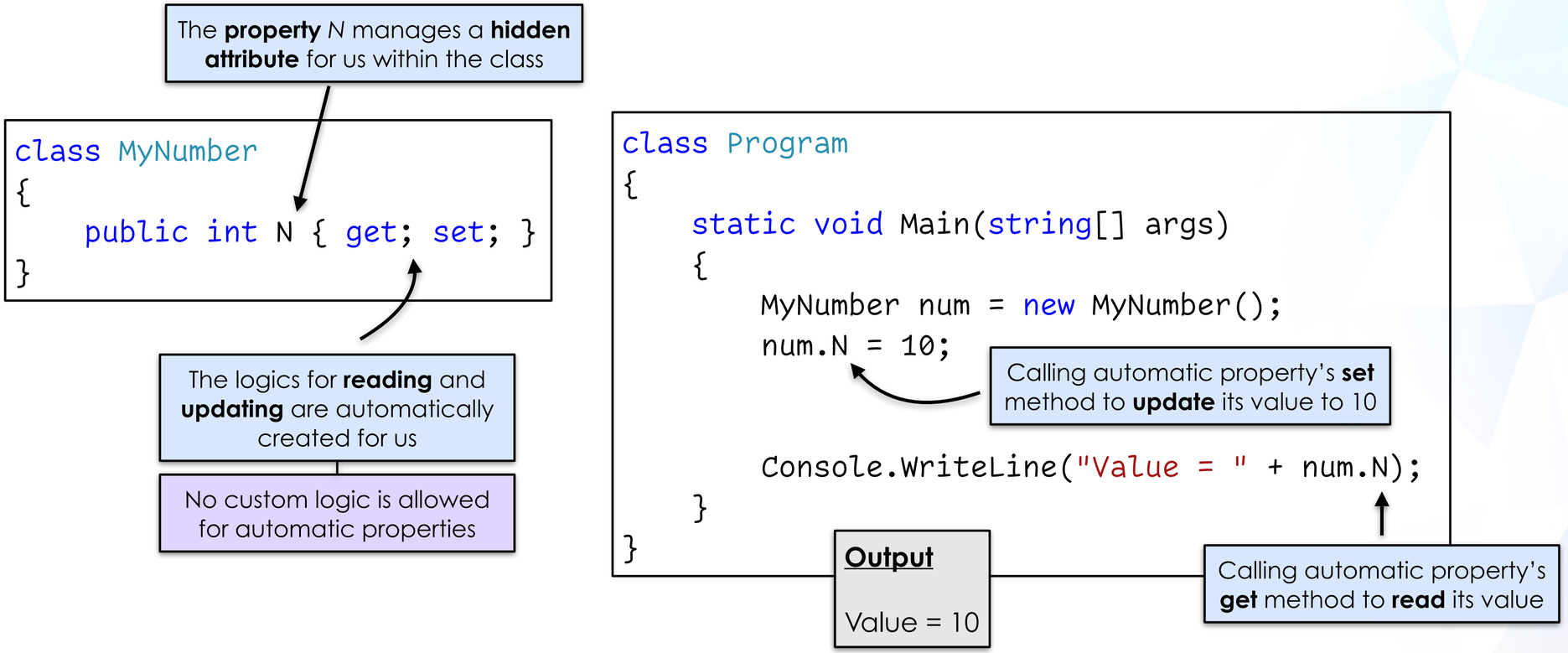
We set 1 here, **why** is the output 2?

Output

Value = 2

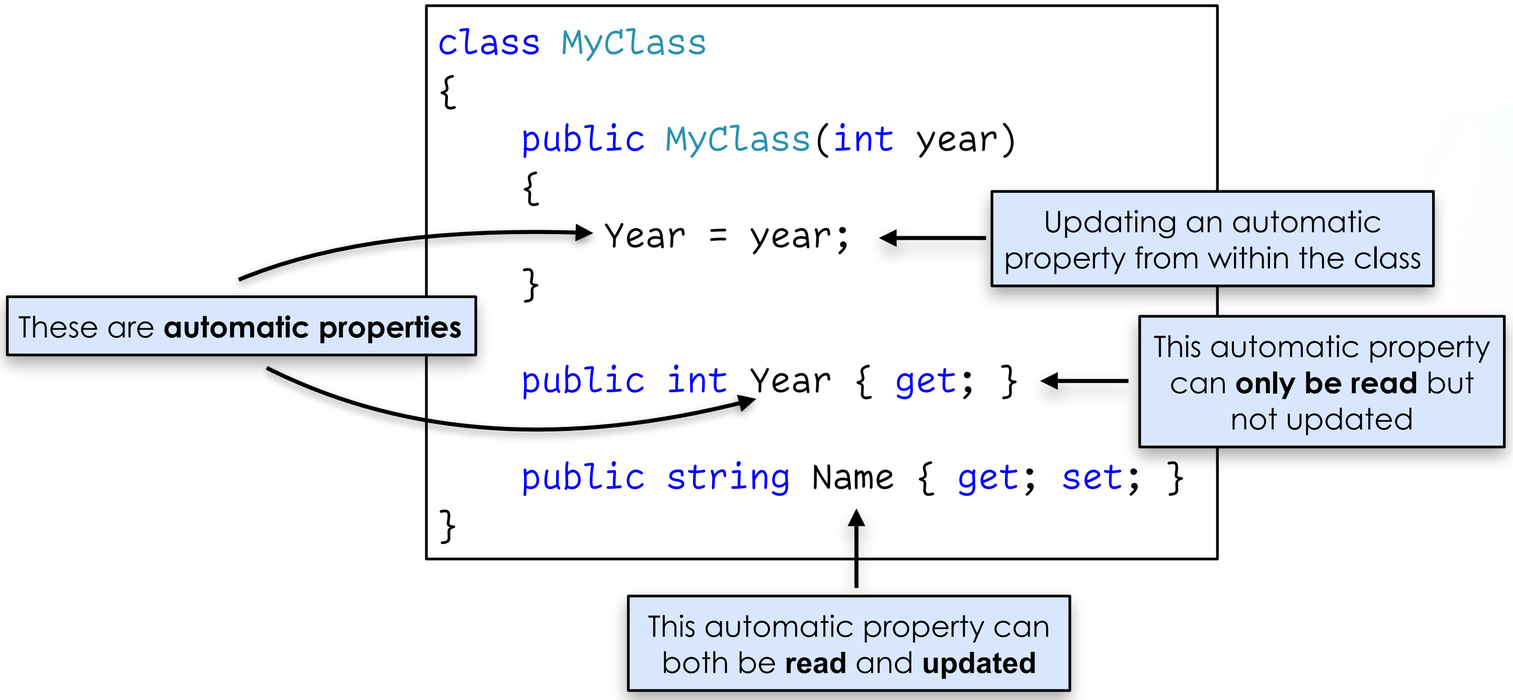
Automatic Properties

Properties can be **automatically created** via the **{get; set;}** language construct



Automatic Properties

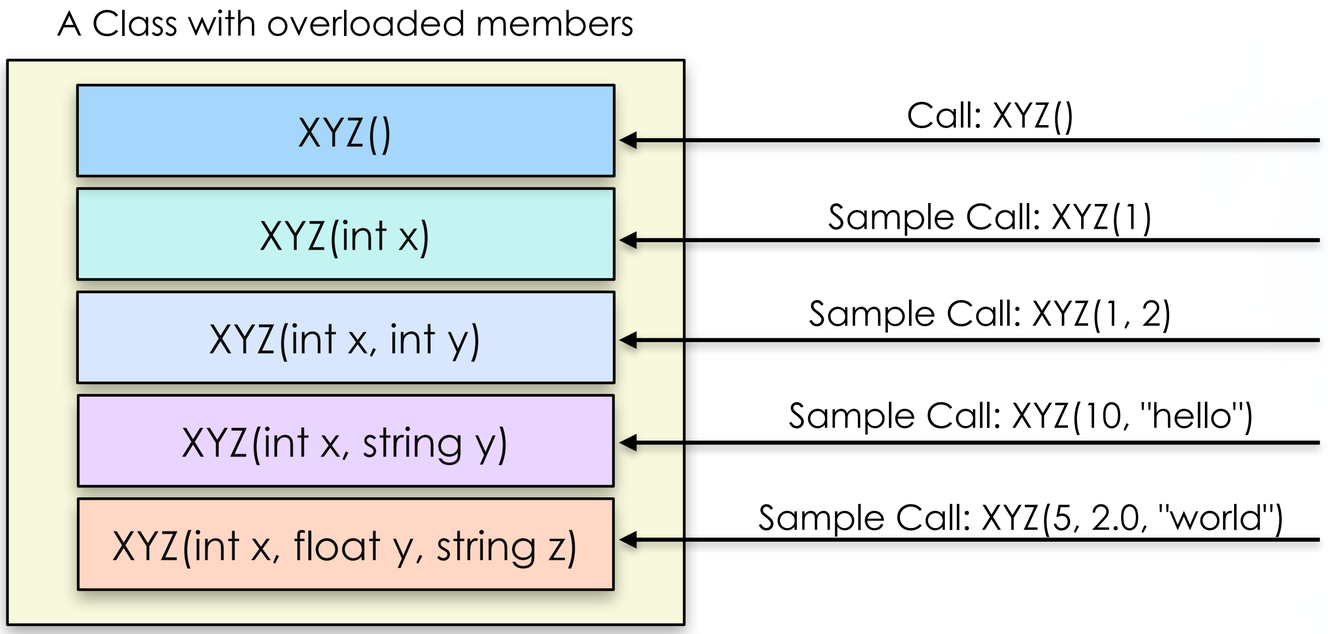
Automatic Properties must have **get** accessors, but **set** accessors are **optional**



Overloading

Overloading

Overloading allows constructors or methods to have the **same name**, but with **different signatures** (different number of parameters and/or parameter types)



Constructor Overloading

Constructor Overloading allows us to define multiple constructors, each with different input parameters

```
class MyClass
{
    protected int n1, n2;

    public MyClass()
    {
        n1 = n2 = 0;
    }

    public MyClass(int n1, int n2)
    {
        this.n1 = n1;
        this.n2 = n2;
    }

    public int GetSum()
    {
        return n1 + n2;
    }
}
```

Default values provided in the constructor

```
class Program
{
    static void Main(string[] args)
    {
        MyClass c1 = new MyClass();
        int sum1 = c1.GetSum();

        MyClass c2 = new MyClass(1, 2);
        int sum2 = c2.GetSum();

        Console.WriteLine("sum1 = " + sum1);
        Console.WriteLine("sum2 = " + sum2);
    }
}
```

We can use **different constructors** to create our objects during instantiation

Output

sum1 = 0
sum2 = 3

Method Overloading

Method Overloading allows methods within a class to have **identical names** as long as their **method signatures** are different

A **method signature** is defined by
(a) the number of parameters
(b) the data types of the parameters

```
class MyClass
{
    public void Greet()
    {
        Console.WriteLine("Hello World!");
    }

    public void Greet(string greet)
    {
        Console.WriteLine(greet);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        MyClass myclass = new MyClass();

        myclass.Greet();
        myclass.Greet("Hi, there!");
    }
}
```

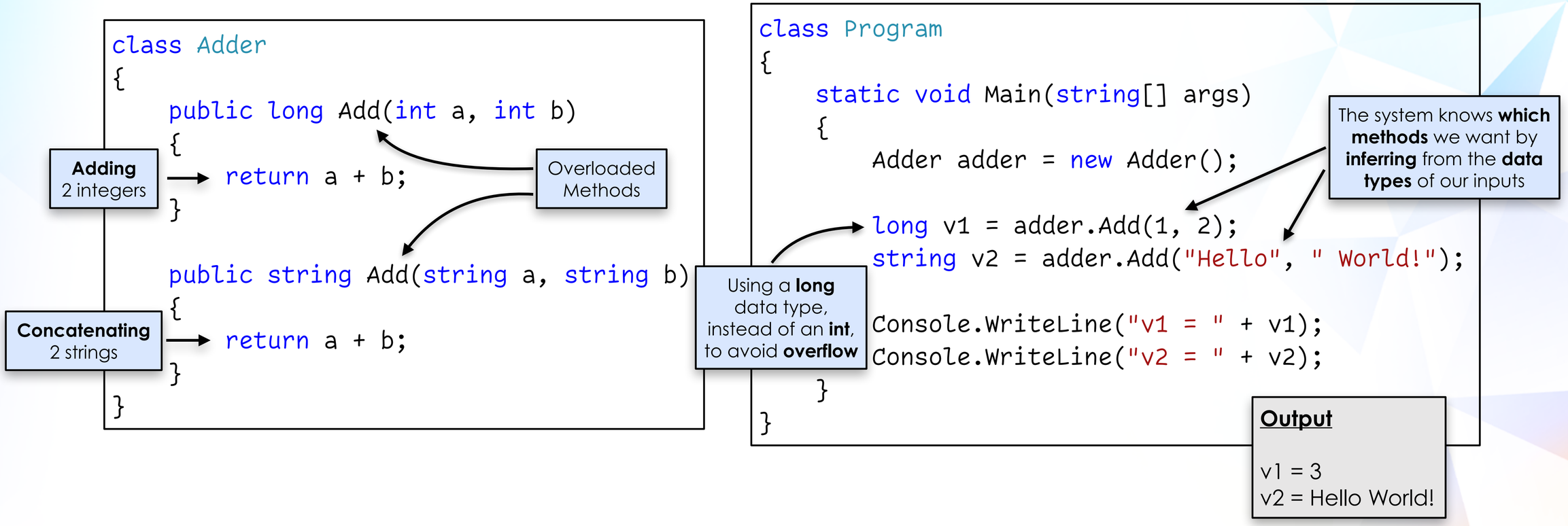
Call our **desired methods** by supplying **different parameters**

Output

Hello World!
Hi, there!

Method Overloading

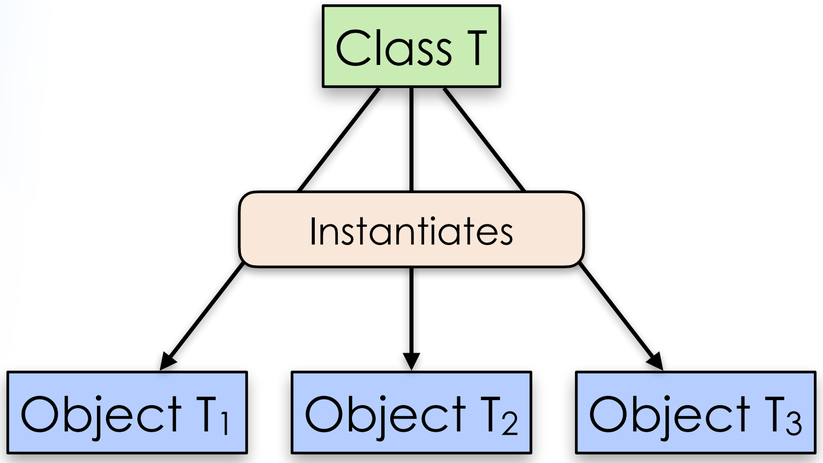
Using Method Overload to provide Add operations on **different data types**

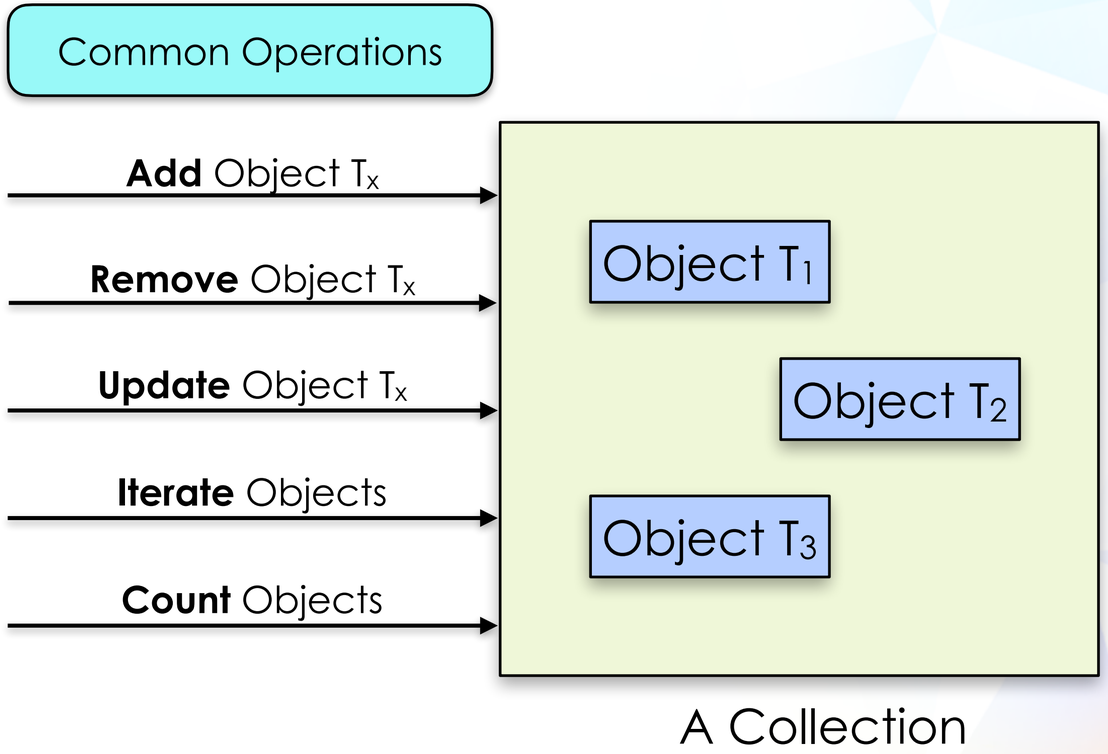


C# Collections

Collections

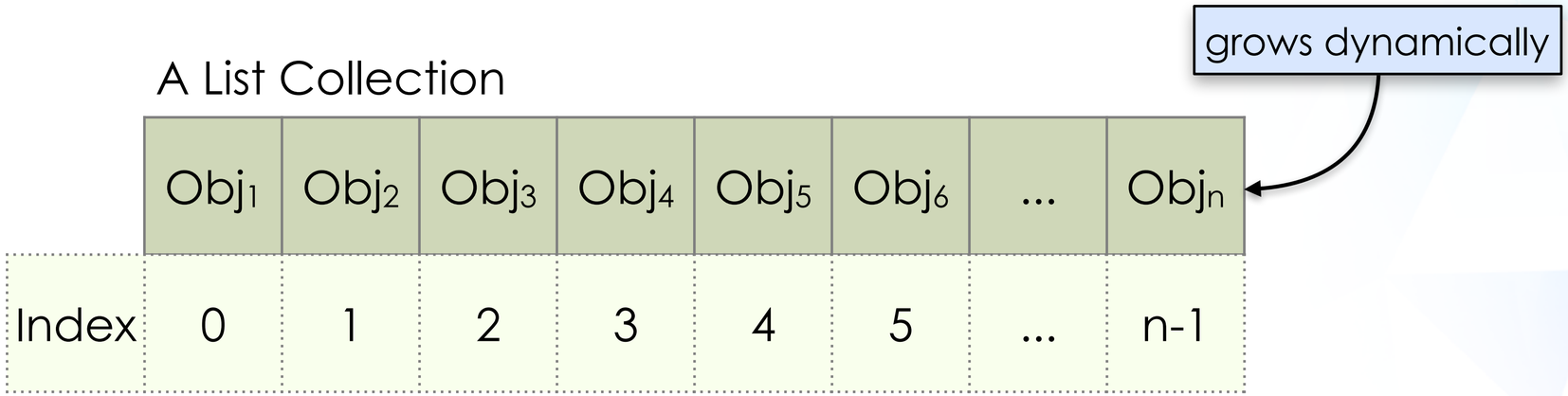
A C# **Collection** represents a group of objects (of the **same type**), and is used for data storage and retrieval





List<T>

A List<**T**> collection acts as a **dynamic array** that stores data (of type **T**) in an **ordered sequence**



Obj_1 to Obj_n are of type **T**, where T is a C# **data type** (e.g. int, string, Class_XYZ)

Adding to and Retrieving from a List Collection

```
class Program
{
    static void Main(string[] args)
    {
        List<int> intList = new List<int>();
        intList.Add(32);
        intList.Add(100);

        Console.WriteLine(intList[0]);
        Console.WriteLine(intList[1]);
    }
}
```

Output

32
100

Remember to add "**using System.Collections.Generic;**" to use List<T>

Using List<T>

Counting, Iterating and Removing from a List Collection

```
class Program
{
    static void Main(string[] args)
    {
        List<string> strList = new List<string>();
        strList.Add("one");
        strList.Add("two");
        strList.Add("three");

        Console.WriteLine("Len: " + strList.Count);

        strList.RemoveAt(0);
        Console.WriteLine("Len: " + strList.Count);

        foreach (string item in strList)
            Console.WriteLine("Item: " + item);

        strList.Clear();
        Console.WriteLine("Len: " + strList.Count);
    }
}
```

Output

Len: 3
Len: 2
Item: two
Item: three
Len: 0

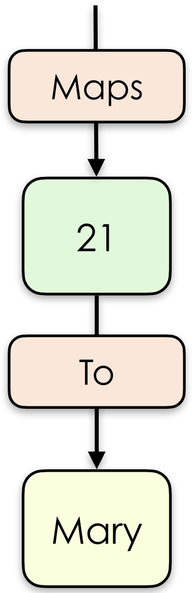
Dictionary<TKey, TValue>

A Dictionary<**TKey**, **TValue**> collection stores **mappings** of **unique keys** (of type **TKey**) to **values** (of type **TValue**)

A Dictionary Collection

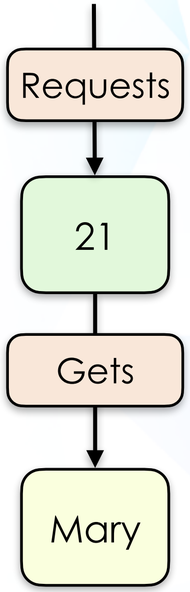
Key	Values
K_1	V_1
K_2	V_2
...	...
K_n	V_n

grows dynamically



Sample Values

Key	Values
8	John
21	**Mary**
11	Jane
44	Larry
55	Ivan



K_1 to K_n are type **TKey** and V1 to Vn are type **TValue**; TKey and TValue are C# **data types**

Dictionary<TKey, TValue>

Adding to and Existence Check on a Dictionary Collection

```
class Program
{
    static void Main(string[] args)
    {
        Dictionary<string, int> strToInt = new Dictionary<string, int>();

        // add key/value pairs
        strToInt["one"] = 1;
        strToInt["two"] = 2;

        // access "value" using "key"
        if (strToInt.ContainsKey("one")) {
            int val = strToInt["one"];
            Console.WriteLine("Value is " + val);
        }
        else {
            Console.WriteLine("Key not found.");
        }
    }
}
```

Output

Value is 1

Using Dictionary<TKey, TValue>

Iterating through a Dictionary collection

```
class Program
{
    static void Main(string[] args)
    {
        Dictionary<string, int> strToInt = new Dictionary<string, int>();

        // add key/value pairs
        strToInt["one"] = 1;
        strToInt["two"] = 2;

        foreach (KeyValuePair<string, int> kv in strToInt) {
            Console.WriteLine("{0} is mapped to {1}", kv.Key, kv.Value);
        }
    }
}
```

Output

one is mapped to 1
two is mapped to 2

Remember to add "**using System.Collections.Generic;**" to use Dictionary<TKey, TValue>

The End