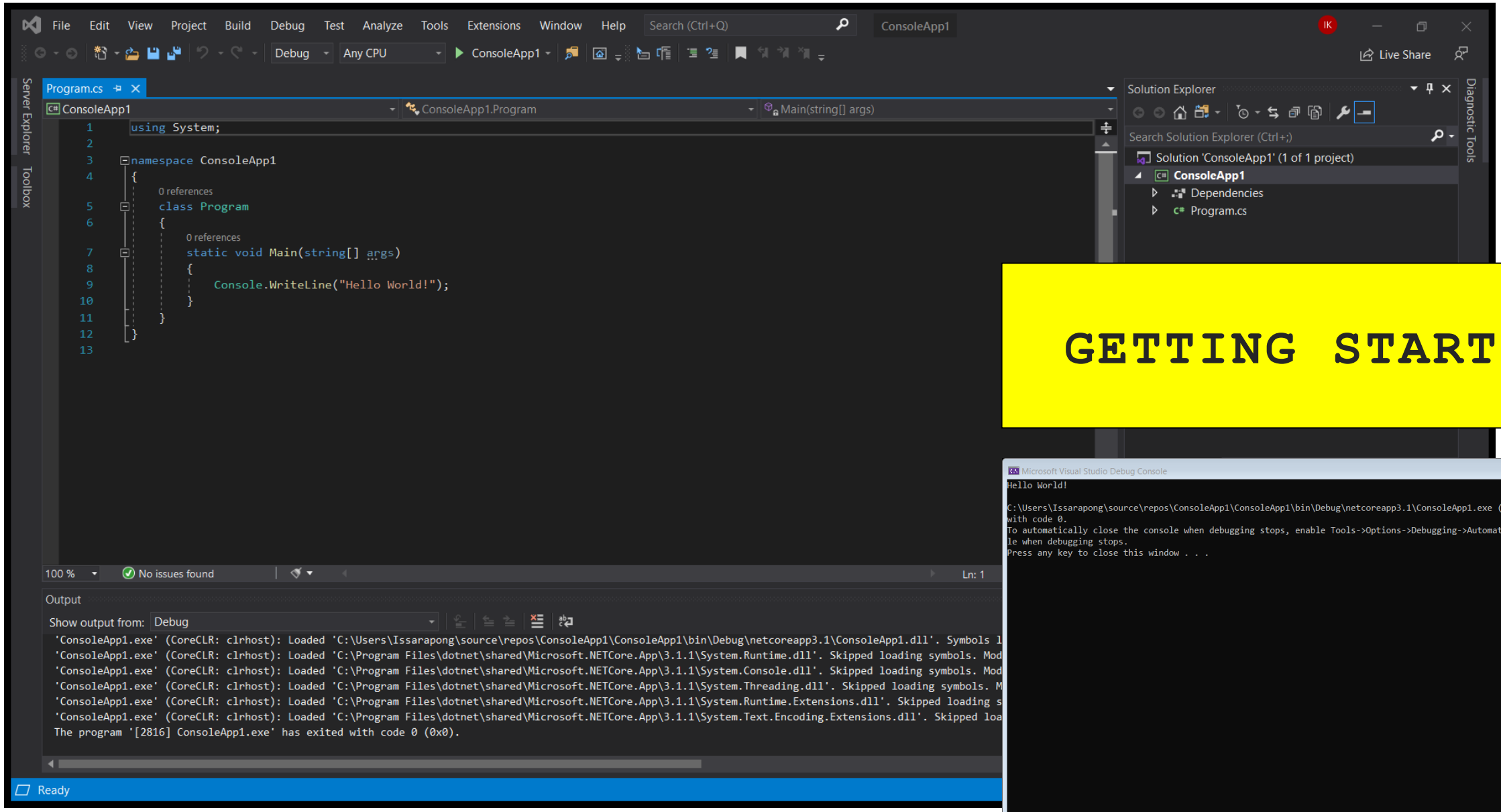
C# 101

Dr. Issarapong Khuankrue

King Mongkut's University of Technology Thonburi

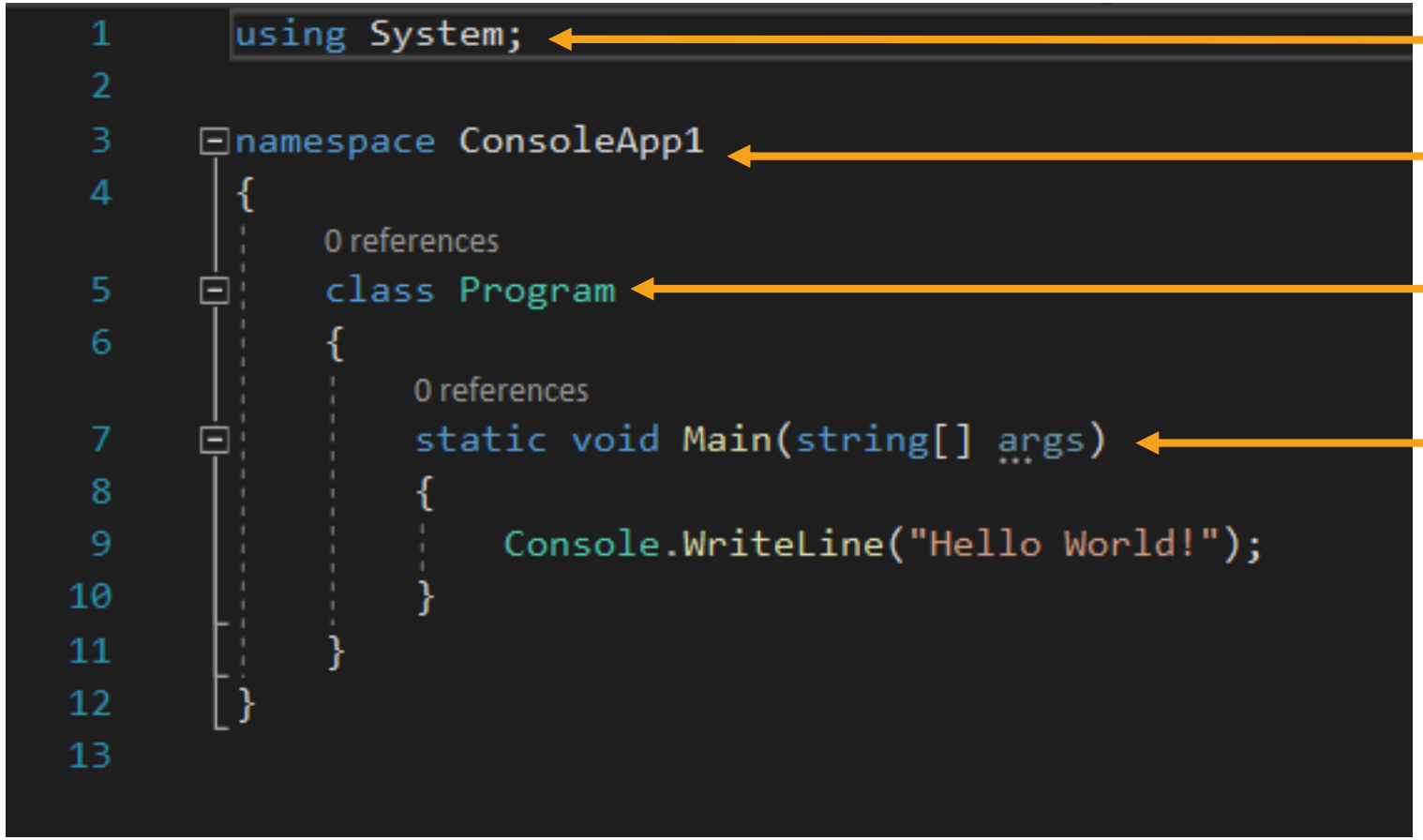
CONTENTS

- **Getting started**
- **From principle to code**
  - Encapsulation
  - Polymorphism
- **Class and Object**
  - Inheritance
  - Abstract class
  - Interface



GETTING STARTED

PROGRAM STRUCTURE



Using — or include

Namespace declaration — A collection of classes

Class declaration — contains the data & method definitions that your program uses

Entry point

Main Method — specifies its behavior with the statement **Console.WriteLine("Hello World");**

- C# is **case sensitive.**
- All statements and expression must end with a **semicolon (;).**
- The program execution starts at **the Main method.**
- **/*...*/** is ignored by the compiler and it is put to add **comments** in the program
- Unlike Java, program file name could be different from the class name.

VARIABLE

Type	Example
Integral types	sbyte, byte, short, ushort, int, uint, long, ulong, and char
Floating point types	float and double
Decimal types	decimal
Boolean types	true or false values, as assigned
Nullable types	Nullable data types

- **Defining Variables** : <*type*> <*name*>;
- **Assignment** : <name> = <expression>;

Ex.

```
int radius;
double area;
radius = 3+4*5;
```

- **Accepting Values from User**
- **ReadLine()** : accepting input from the user and store it into a variable

Ex.

```
int num;
num = Convert.ToInt32(Console.ReadLine());
```

Convert.ToInt32() converts the data entered by the user to int data type
Console.ReadLine() *accepts the data in string format.*

MEMBER VARIABLE/METHODS

- **Defining Methods**

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
   Method Body
}
```

- **Access Specifier** – determines the visibility of a variable or a method from another class.
- **Return type** – A method may return a value.
- **Method name** – unique identifier and case sensitive.
- **Parameter list** – the type, order, and number of the parameters of a method.
- **Method body** – set of instructions needed to complete the required activity.

```
using System;

namespace ConsoleApp1
{
    2 references
    class Rectangle
    {
        //member variables
        public double length;
        public double width;

        1 reference
        public double GetArea()
        {
            return length * width;
        }
        1 reference
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.length = 4.5;
            r.width = 3.5;
            r.Display();
            Console.ReadLine();
        }
    }
}
```

Member Variables

Member Methods

DECISION MAKING

```
if (<boolean expression>)
{
    statement1;
    :
}
else
{
    statement2;
    :
}
```

```
int number;
Console.WriteLine("Please enter a number between 0 and 10:");
number = int.Parse(Console.ReadLine());

if(number > 10)
    Console.WriteLine("Hey! The number should be 10 or less!");
else
if(number < 0)
    Console.WriteLine("Hey! The number should be 0 or more!");
else
    Console.WriteLine("Good job!");

Console.ReadLine();
```

ITERATION

```
while (<boolean exp>)
{
    :
}
```

```
while(number < 5)
        {
        Console.WriteLine(number);
        number = number + 1;
        }
```

```
do
{
    :
} while (<boolean exp>);
```

```
int number = 0;
do
{
    Console.WriteLine(number);
    number = number + 1;
} while(number < 5);
```

ITERATION

```
for (<init>; <condition>; <increment>)
{
    :
}
```

```
for(int i = 0; i < number; i++)
        Console.WriteLine(i);
```

```
foreach (<type> <var> in <array>)
{
    :
}
```

```
int[] a = new int[] {1,2,3,4,5};
int sum = 0;

foreach (int i in a)
{
    sum = sum + i;
}
```

ARRAY

One dimentional array

```
string[] weekDays;
weekDays = new string[] {"Sun","Sat","Mon","Tue","Wed","Thu","Fri"};
```

```
int[] numbers = { 4, 3, 8, 0, 5 };
Array.Sort(numbers);
foreach(int i in numbers)
Console.WriteLine(i);
```

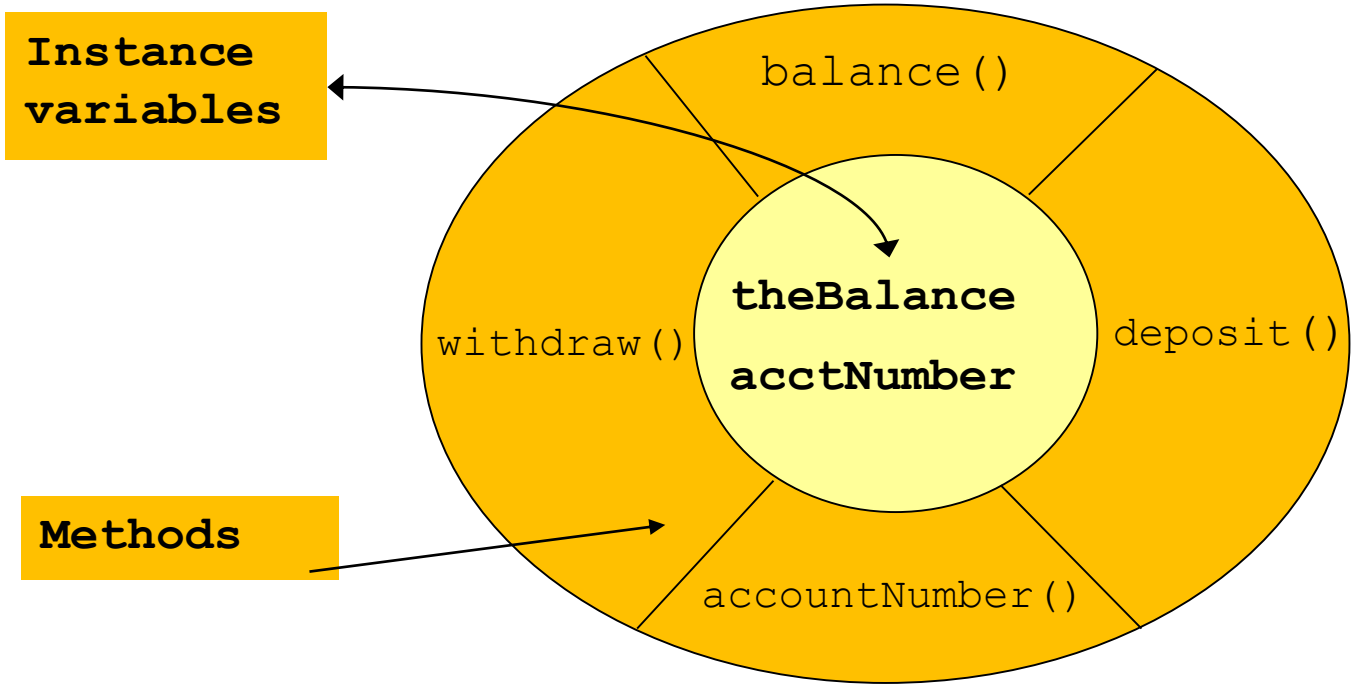
Two dimentional array

```
int[,] myArray; // 2-dimension

myArray = new int[,] {{1,2}, {3,4}, {5,6}, {7,8}};
```

ENCAPSULATION (INFORMING HIDING)

- **Encapsulation : Hide implementation from clients**
  - Clients depend on interface – only!
  - Clients do not need to know 'how' the server operates or provides the services!

- How does an object encapsulate?
- What does it encapsulate?



ENCAPSULATION

- **Visibility**
  - **Attributes** can be public or private:
    - **Private**: only be accessed by its own methods
    - **Public**: can be modified by methods associated with any class (violates encapsulation)
  - **Methods** can be public, private or protected:
    - **Public**: it is exposed to other objects.
    - **Private**: it can't be accessed by other objects, only internally
    - **Protected**: (special case) only subclasses that descend directly from a class that contains it, know and can use this method.

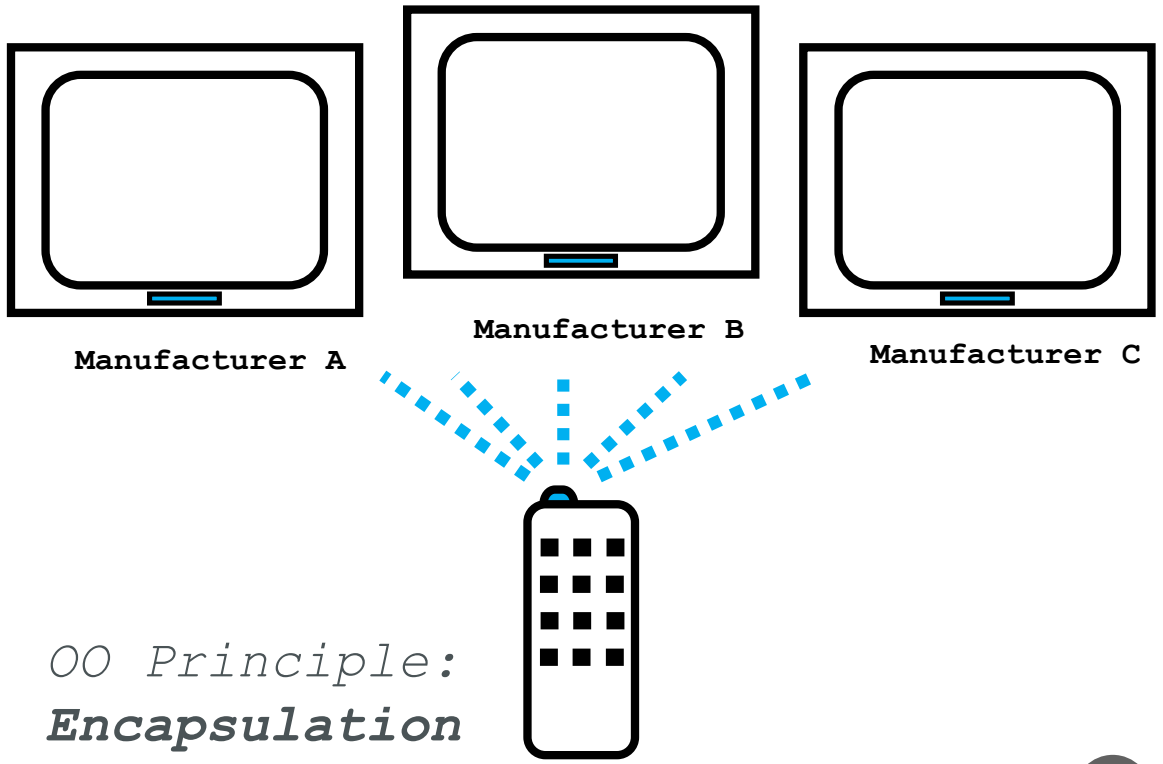
```
using System;

namespace ConsoleApp1
{
    2 references
    class Rectangle
    {
        //member variables
        private double length;
        private double width;

        1 reference
        public void Acceptdetails()
        {
            Console.WriteLine("Enter Length: ");
            length = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("Enter Width: ");
            width = Convert.ToDouble(Console.ReadLine());
        }
        1 reference
        public double GetArea()
        {
            return length * width;
        }
        2 references
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.Acceptdetails();
            r.Display();
            r.Display();
            Console.ReadLine();
        }
    }
```

POLYMORPHISM

- The ability to hide many **different implementations** behind a single interface.

The **same method** will behave differently when it is applied to the objects of **different classes**.

In the same way, the **different methods** associated with different classes can interpret the **same message** in different ways.



OO Principle:
Encapsulation

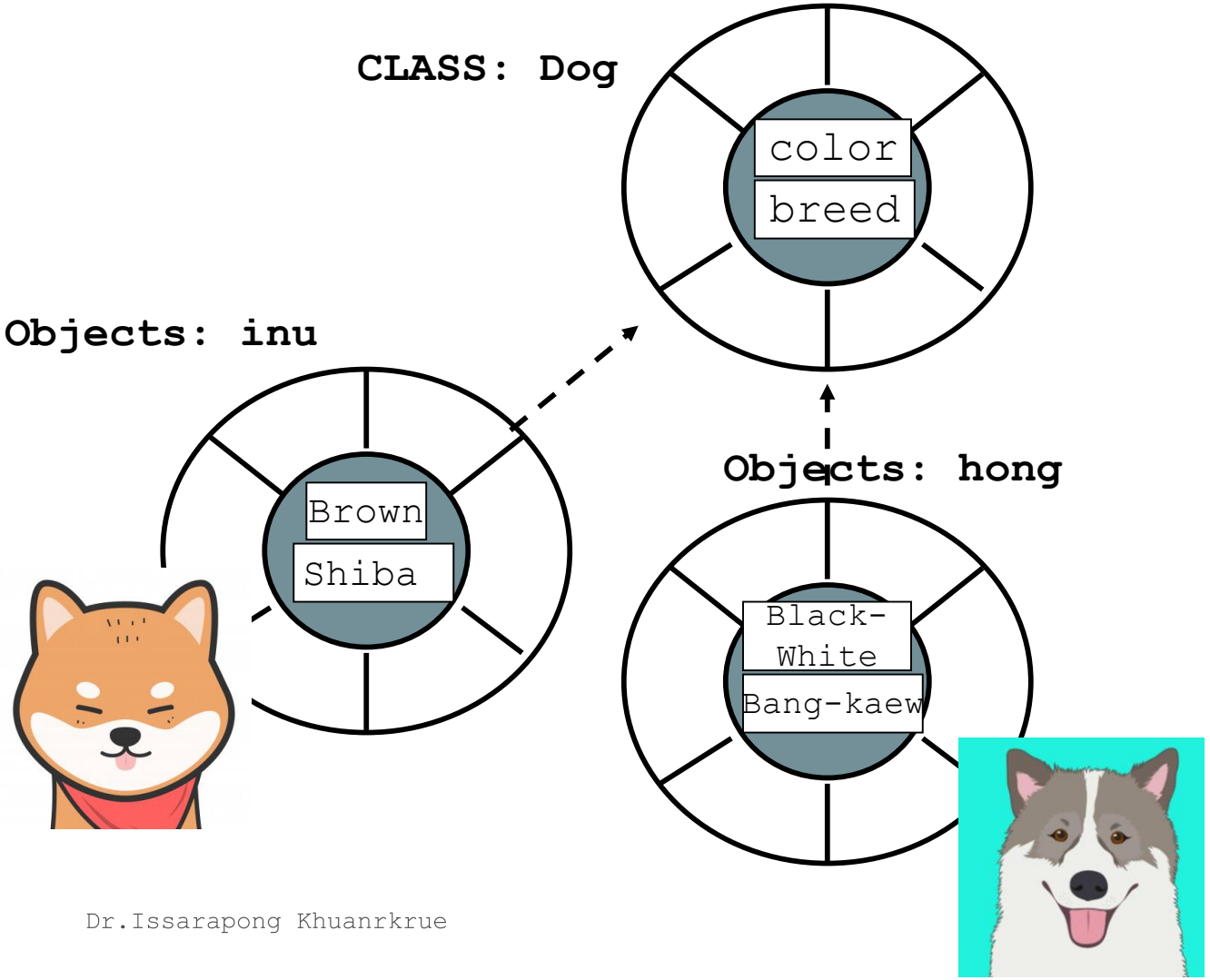
POLYMORPHISM

overload

```
class Printdata
{
    1 reference
    public void print(int i)
    {
        Console.WriteLine("Printing int: {0}", i);
    }
    1 reference
    public void print(double f)
    {
        Console.WriteLine("Printing float: {0}", f);
    }
    1 reference
    public void print(string s)
    {
        Console.WriteLine("Printing string: {0}", s);
    }
}
```

```
static void Main(string[] args)
{
    Printdata p = new Printdata();
    // Call print to print integer
    p.print(5);
    // Call print to print float
    p.print(500.263);
    // Call print to print string
    p.print("Hello C# world");
    Console.ReadKey();
}
```

CLASS, OBJECT AND "NEW" STATEMENT



- **Member variable** are the attributes of an object.
  - These variables can only be accessed using the public member functions.
  - Ex. Color, Breed, etc.
- Using **new** statement to create object from class

```
Ex.
dog inu = new dog;
```

- C# needs at least one class in any program.

- **Properties** : control the accessibility of a class's variables and is the recommended way to access variables from the outside in OOP.

```
private string color;

public string Color
{
    get { return color; }
    set { color = value; }
}
```

```
public string Color
{
    get
    {
        return color.ToUpper();
    }
    set
    {
        if(value == "Red")
            color = value;
        else
            Console.WriteLine("This car can only be red!");
    }
}
```

```
using System;

namespace ConsoleApp2
{
    4 references
    class Car
    {
        private string color;

        2 references
        public Car(string color)
        {
            this.color = color;
        }

        2 references
        public string Describe()
        {
            return "This car is " + Color;
        }

        1 reference
        public string Color
        {
            get { return color; }
            set { color = value; }
        }
    }
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Car car;

            car = new Car("Red");
            Console.WriteLine(car.Describe());

            car = new Car("Green");
            Console.WriteLine(car.Describe());

            Console.ReadLine();
        }
    }
```

- **Constructors and destructors:**
  - **Constructors** are special methods, used when instantiating a class.

A constructor can be defined like this:

```
public string Describe()
```

overload

```
public Car()
{
    Console.WriteLine("Constructor with no parameters called!");
}

public Car(string color) : this()
{
    this.color = color;
    Console.WriteLine("Constructor with color parameter called!");
}
```

```
using System;

namespace ConsoleApp2
{
    4 references
    class Car
    {
        private string color;

        2 references
        public Car(string color)
        {
            this.color = color;
        }

        2 references
        public string Describe()
        {
            return "This car is " + Color;
        }

        1 reference
        public string Color
        {
            get { return color; }
            set { color = value; }
        }
    }
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Car car;

            car = new Car("Red");
            Console.WriteLine(car.Describe());

            car = new Car("Green");
            Console.WriteLine(car.Describe());

            Console.ReadLine();
        }
    }
```

- **Constructors and destructors:**
  - If you call the constructor which takes 2 parameters, the first parameter will be used to invoke the constructor that takes 1 parameter.

```
public Car(string color) : this()
{
    this.color = color;
    Console.WriteLine("Constructor with color parameter called!");
}

public Car(string param1, string param2) : this(param1)
{

}
```

  - **Destructors** can be used to cleanup resources used by the object

```
~Car()
{
    Console.WriteLine("Out..");
}
```

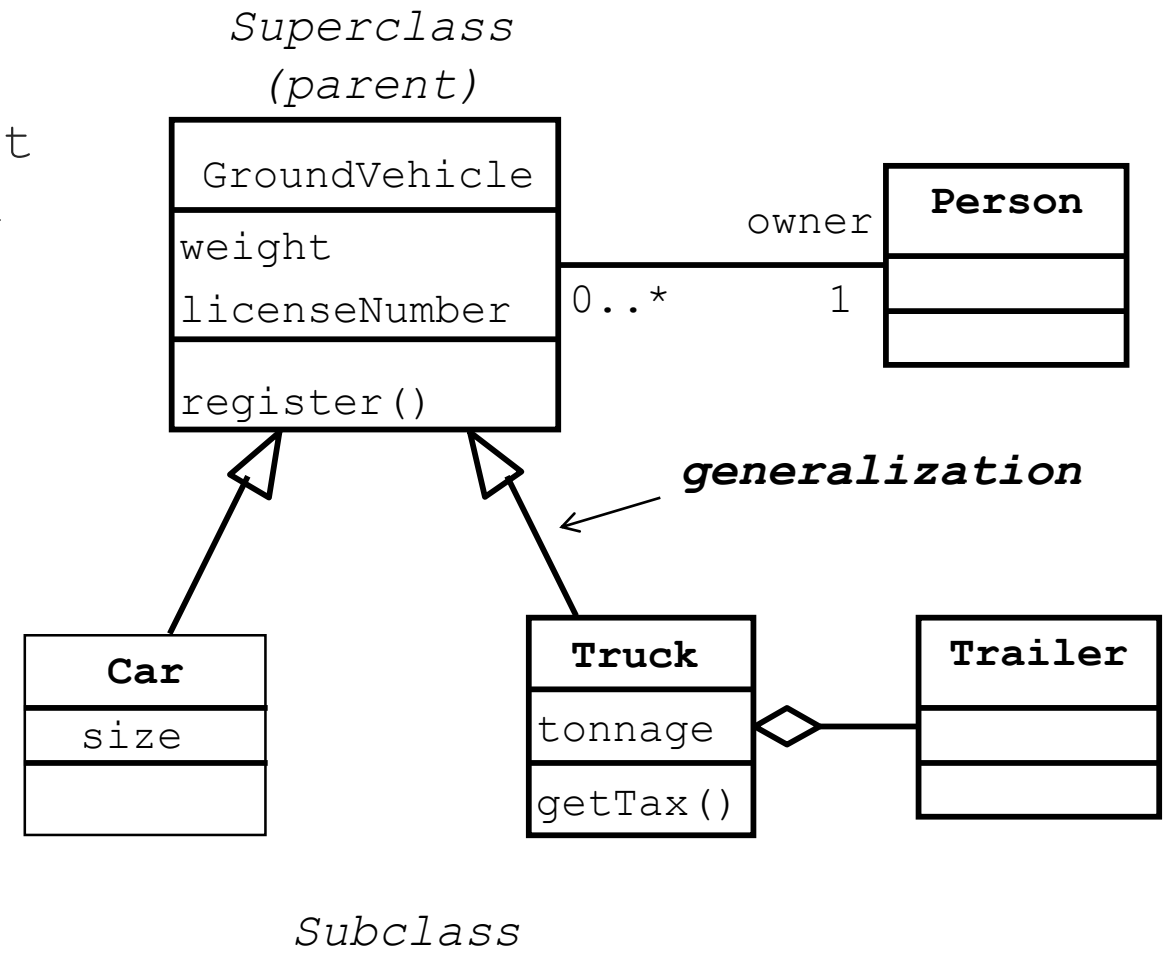
```
using System;

namespace ConsoleApp2
{
    4 references
    class Car
    {
        private string color;

        2 references
        public Car(string color)
        {
            this.color = color;
        }

        2 references
        public string Describe()
        {
            return "This car is " + Color;
        }

        1 reference
        public string Color
        {
            get { return color; }
            set { color = value; }
        }
    }
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Car car;

            car = new Car("Red");
            Console.WriteLine(car.Describe());

            car = new Car("Green");
            Console.WriteLine(car.Describe());

            Console.ReadLine();
        }
    }
```

STATIC MEMBERS

```
class Rectangle
    {
        private int width, height;

        public Rectangle(int width, int height)
        {
            this.width = width;
            this.height = height;
        }

        public void OutputArea()
        {
            Console.WriteLine("Area output: " + Rectangle.CalculateArea(this.width, this.height));
        }

        public static int CalculateArea(int width, int height)
        {
            return width * height;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle(10,20);
            r.OutputArea();
        }

    }
```

INHERITANCE (REUSE)

- **Inheritance** is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities.
  - The existing classes :
    - Base classes/ Parent classes/ Super-classes,
  - New classes :
    - Derived classes/ Child classes/ Subclasses.



INHERITANCE (REUSE)

Parent class

```
public class Animal
{
    2 references
    public void Greet()
    {
        Console.WriteLine("Hello, I'm some sort of animal!");
    }
}
```

Child class

```
2 references
public class Dog : Animal
{

}
```

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Animal animal = new Animal();
        animal.Greet();
        Dog dog = new Dog();
        dog.Greet();
    }
}
```

it **inherits** this method from the Animal class

INHERITANCE (REUSE)

```
public class Animal
{
    4 references
    public virtual void Greet()
    {
        Console.WriteLine("Hello, I'm some sort of animal!");
    }
}

2 references
public class Dog : Animal
{
    4 references
    public override void Greet()
    {
        base.Greet(); /*still access the inherited method*/
        Console.WriteLine("Hello, I'm a dog!");
    }
}
```

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Animal animal = new Animal();
        animal.Greet();
        Dog dog = new Dog();
        dog.Greet();
    }
}
```

ABSTRACT CLASSES

- Abstract classes, marked by the keyword abstract in the class definition, are typically used to define a base class in the hierarchy.

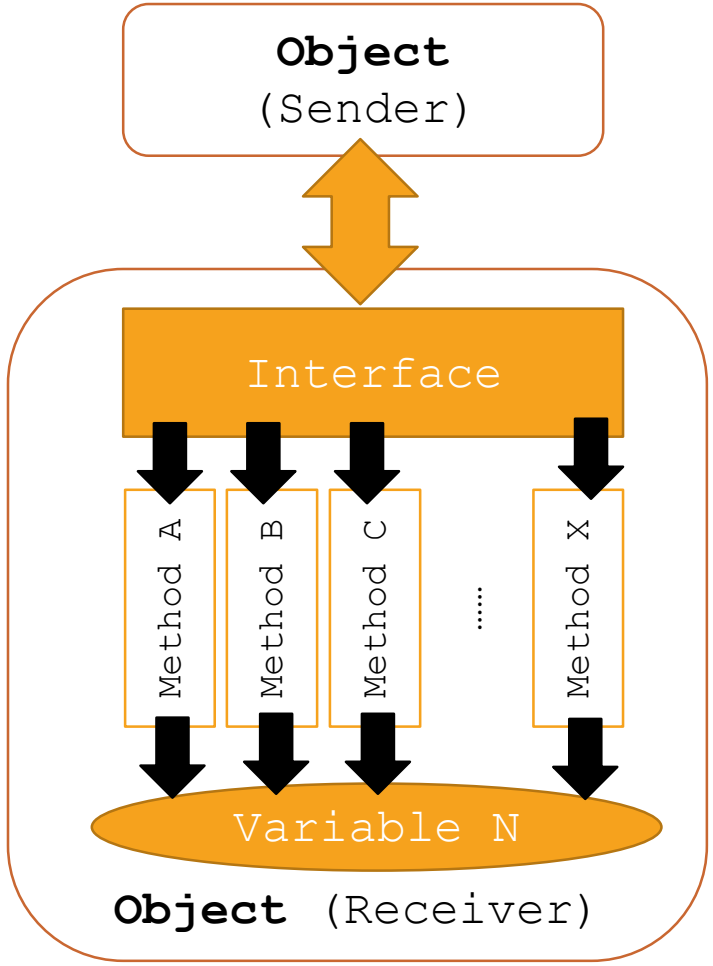
```
abstract class FourLeggedAnimal
{
    1 reference
    public virtual string Describe()
    {
        return "Not much is known about this four legged animal!";
    }
}

2 references
class Dog : FourLeggedAnimal
{

}
```

```
public override string Describe()
{
    string result = base.Describe();
    result += " In fact, it's a dog!";
    return result;
}
```

```
static void Main(string[] args)
{
    Dog dog = new Dog();
    Console.WriteLine(dog.Describe());
    Console.ReadKey();
}
```

We can go a long way without needing an abstract class, but they are great for specific things, like frameworks

INTERFACE



- **Visibility**
  - **Attributes** can be public or private:
    - **Private:** only be accessed by its own methods
    - **Public:** can be modified by methods associated with any class (violates encapsulation)
  - **Methods** can be public, private or protected:
    - **Public:** it is exposed to other objects.
    - **Private:** it can't be accessed by other objects, only internally
    - **Protected:** (special case) only subclasses that descend directly from a class that contains it, know and can use this method.

INTERFACE

```
interface IAnimal
{
    2 references
    string Describe();
    5 references
    string Name
    {
        get;
        set;
    }
}
```

C# doesn't allow multiple inheritance, where classes inherit more than a single base class, BUT it allow for multiple interfaces!

```
class Dog : IAnimal, IComparable
{
    private string name;
    3 references
    public Dog(string name)
    {
        this.Name = name;
    }
    2 references
    public string Describe()
    {
        return "Hello, I'm a dog and my name is " + this.Name;
    }
    0 references
    public int CompareTo(object obj)
    {
        if (obj is IAnimal)
            return this.Name.CompareTo((obj as IAnimal).Name);
        return 0;
    }
    5 references
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

The interface defines

- the **'what'** part of the syntactical contract.
- the deriving classes define the **'how'** part of the syntactical contract.

```
static void Main(string[] args)
{
    List<Dog> dogs = new List<Dog>();
    dogs.Add(new Dog("Fido"));
    dogs.Add(new Dog("Bob"));
    dogs.Add(new Dog("Adam"));
    dogs.Sort();
    foreach (Dog dog in dogs)
        Console.WriteLine(dog.Describe());
    Console.ReadKey();
}
```