7SENG011W Object Oriented Programming

Introduction to Classes and Objects

Dr Francesco Tusa

Readings

The topics we will discuss today can be found in the books

- Hands-On Object-Oriented Programming with C#
    - Chapter: Hello OOP – Classes and Objects
- Object-Oriented Thought Process
    - Chapter: Introduction to Object-Oriented Concepts

Outline

- Summary of last week
- Overview of Classes and Objects
- Our first OOP program: Point Class
- Object Oriented Programs Design

Outline

- Summary of last week
- Overview of Classes and Objects
- Our first OOP program: Point Class
- Object Oriented Programs Design

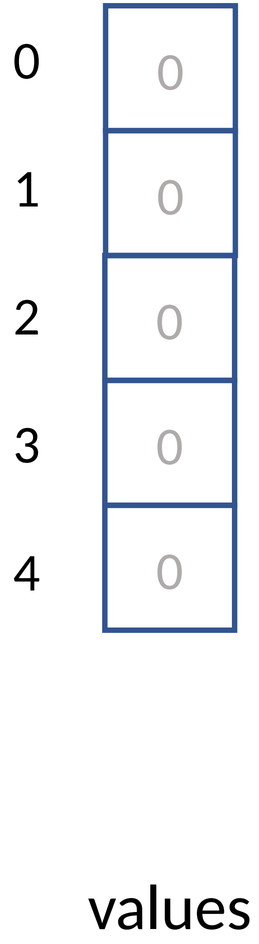
Arrays

```
double [] values = new double[5];
```

size: fixed number of elements

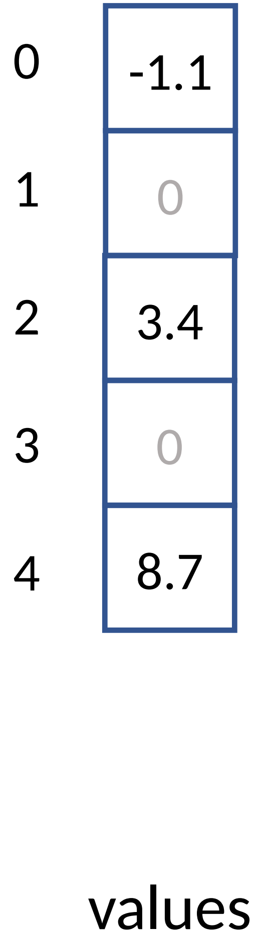
The index starts at 0

ends at n-1



Arrays

```
double [] values = new double[5];

// as lvalue
values[2] = 3.4;
values[0] = -1.1;
values[4] = 8.7;

// as rvalue
int sum = values[4] + 1
```

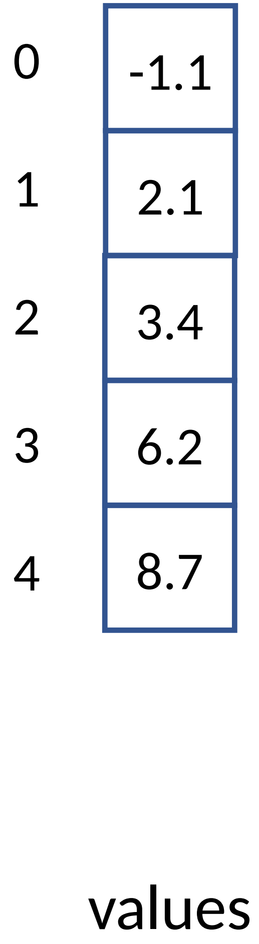


0	-1.1
1	0
2	3.4
3	0
4	8.7

values

Arrays

```
double [] values = { -1.1, 2.1, 3.4, 6.2, 8.7 };
```



0 -1.1

1 2.1

2 3.4

3 6.2

4 8.7

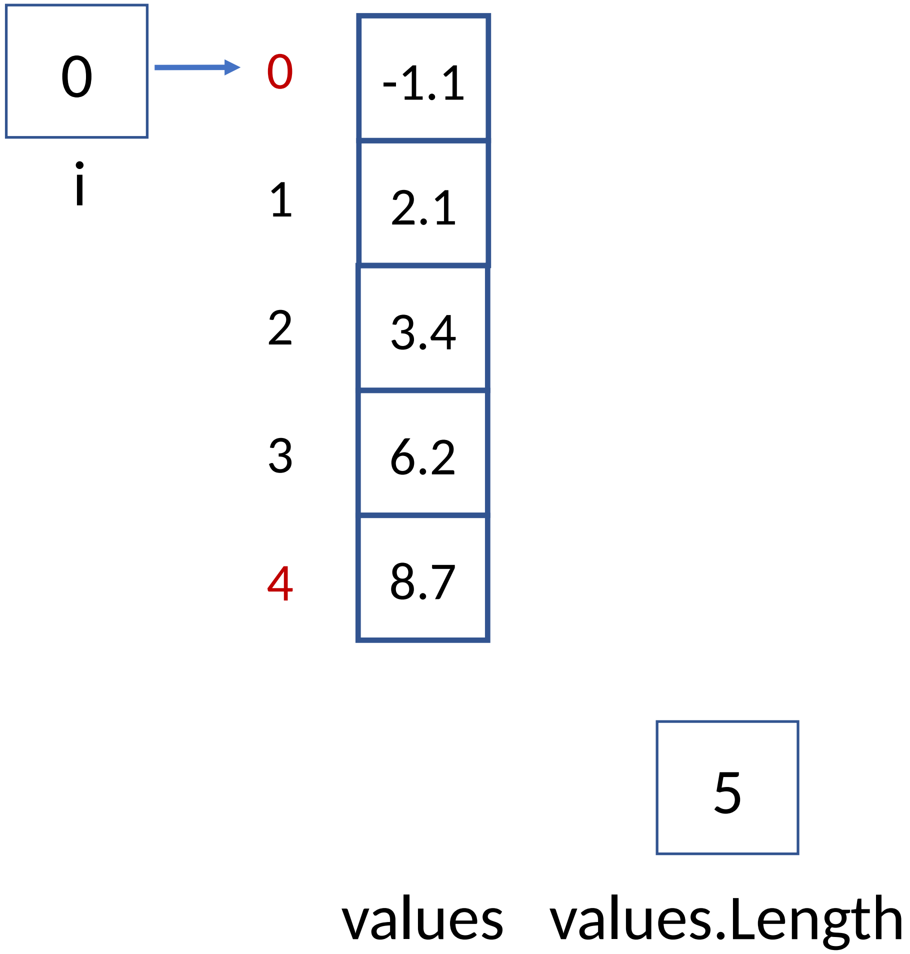
values

Array size

- When an array is declared its **size must be explicitly specified**
- Once the array is allocated its **size cannot be changed—fixed**
- *ArrayList* is a *Collection* that supports *dynamic allocation* (later)

Arrays: for loop

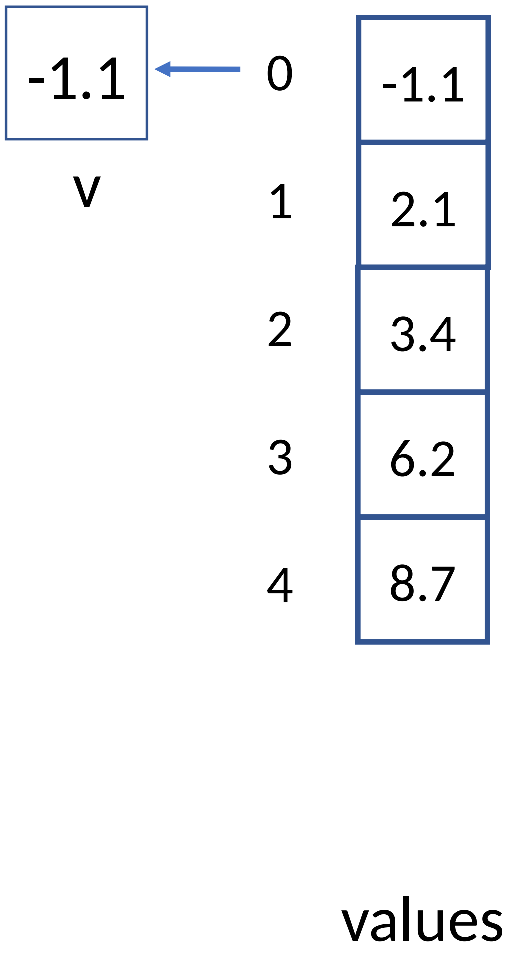
```
double [] values = { -1.1, 2.1, 3.4, 6.2, 8.7 };

for (int i = 0; i < values.Length; i++)
{
    Console.WriteLine(values[i]);
}
```



0	-1.1
1	2.1
2	3.4
3	6.2
4	8.7

Arrays: foreach loop

```
double [] values = { -1.1, 2.1, 3.4, 6.2, 8.7 };

foreach (int v in values)
{
     Console.WriteLine(v);
}
```



-1.1

v

0 -1.1

1 2.1

2 3.4

3 6.2

4 8.7

values

Arrays: program arguments

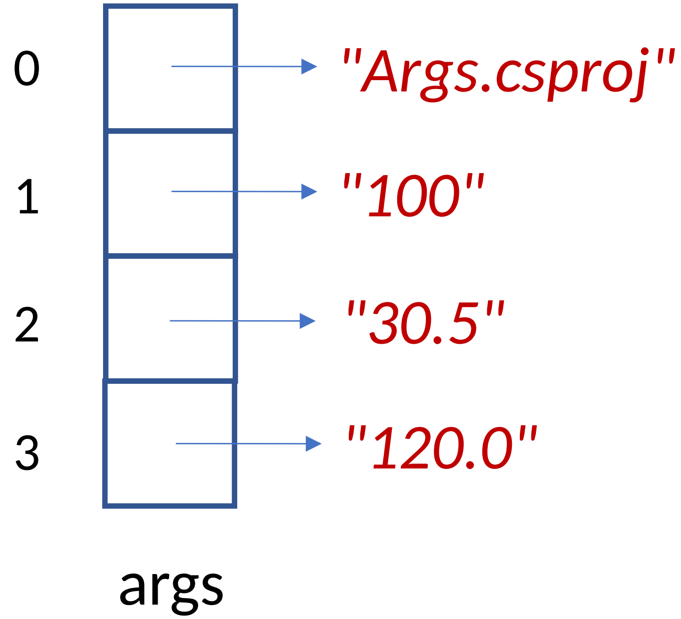
```
$ program arg0 arg1 arg2 … argn
```

Example:

```
$ dotnet run Args.csproj 100 30.5 120.0
```

(behind the scenes…)

```
string[] args = {"Args.csproj", "100", "30.5", "120.0"};
```



0 "Args.csproj"
1 "100"
2 "30.5"
3 "120.0"
args

Demo

Outline

- Summary of last week
- Overview of Classes and Objects
- Our first OOP program: Point Class
- Object Oriented Programs Design

Our Programs so far can manipulate…

- Data stored inside **variables**
  - *Primitive* data types (int, double, etc.) and string type

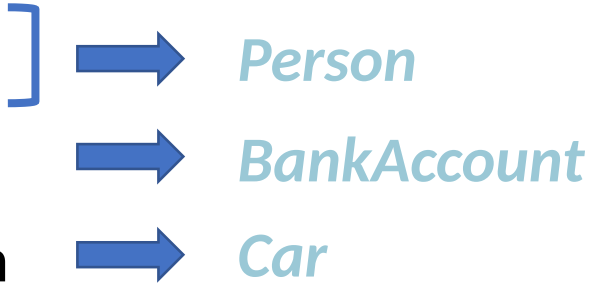
- **Multiple** data elements of the **same** type grouped as **array**s
  - int[] values, double[] timings, string[] args, etc.

What are Objects?

- *Different types of data* can be grouped to model **entities** of a **problem** we would like to solve
    - "James", "04/08/1962", 67.5 Kg, 175 cm
    - "Tom", "05/03/1992", 85.7 Kg, 185 cm
    - "A0123", "21/09/2022", £300.50
    - "AB72CDE", "500", "Fiat", 55 HP, 95 km/h

What are Objects?

- *Different types of data* can be grouped to model **entities** of a **problem** we would like to solve
  - "James", "04/08/1962", 67.5 Kg, 175 cm
  - "Tom", "05/03/1992", 85.7 Kg, 185 cm
  - "A0123", "21/09/2022", £300.50
  - "AB72CDE", "500", "Fiat", 55 HP, 95 km/h



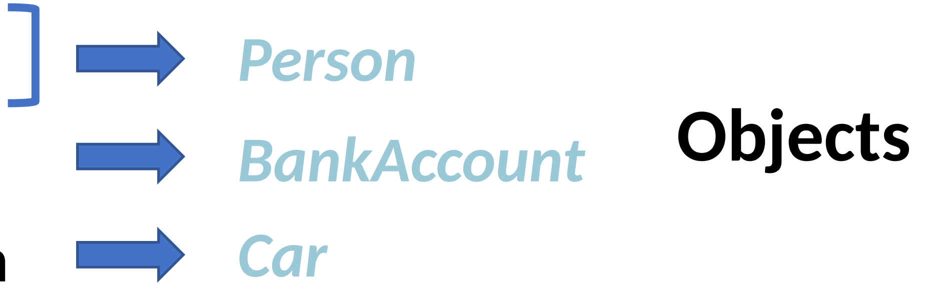
Person

BankAccount

Car

What are Objects?

- *Different types of data* can be grouped to model **entities** of a **problem** we would like to solve
  - "James", "04/08/1962", 67.5 Kg, 175 cm
  - "Tom", "05/03/1992", 85.7 Kg, 185 cm
  - "A0123", "21/09/2022", £300.50
  - "AB72CDE", "500", "Fiat", 55 HP, 95 km/h



Person

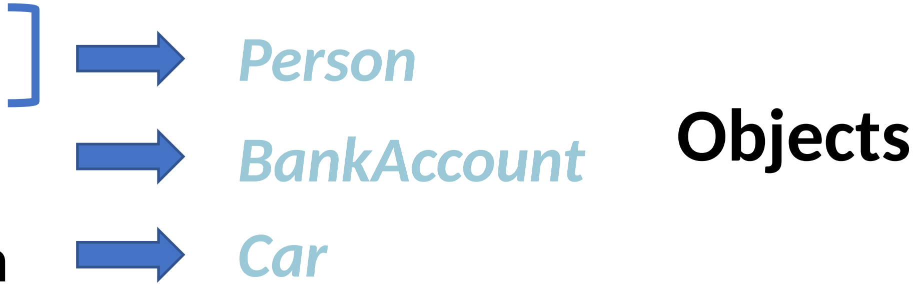
BankAccount

Car

Objects

Object attributes

- *Different types of data* can be grouped to model **entities** of a **problem** we would like to solve
  - "James", "04/08/1962", 67.5 Kg, 175 cm
  - "Tom", "05/03/1992", 85.7 Kg, 185 cm
  - "A0123", "21/09/2022", £300.50
  - "AB72CDE", "500", "Fiat", 55 HP, 95 km/h



Person

BankAccount

Car

Objects

- Each of the above sets of **data** represents the **state** of each **Object**
- They are called **attributes** of the Object

Object methods

- Objects have **behaviours** – *operations* they can perform:
  - *Person* Objects can *SayName*, *Eat*, *GetAge*, …
  - *BankAccount* Objects can *Deposit*, *Withdraw*, *GetBalance*, …
  - *Car* Objects can *SetEngineOn*, *GetSpeed*, *GetModel*, …

- Those behaviours are the **methods** of each Object

Classes

- Defines a set of **attributes** (data) and **methods** (behaviours) that are common for some Objects

class Person

attributes: *string name, string dateOb, double weight, int height*

methods*: SayName, Eat, GetAge*, …

- It can be used as a **blueprint** (template) to create Objects that **will have** those specific *attributes* and *methods*

Classes

- Defines a set of **attributes** (data) and **methods** (behaviours) that are common for some Objects

class Person

attributes: *string name, string dateOb, double weight, int height*

methods: *SayName*, *Eat*, *GetAge*, …

- Why **those** particular ***attributes*** and ***methods*** above?

Object-Oriented Programming (OOP) Principles

- **Abstraction**
- **Encapsulation**
- **Inheritance**
- **Polymorphism**

The process of **generalising concrete details** away from the study of objects and systems to focus on details of greater **importance for the problem to be solved**

Classes

- Defines a set of **attributes** (data) and **methods** (behaviours) that are common for some Objects

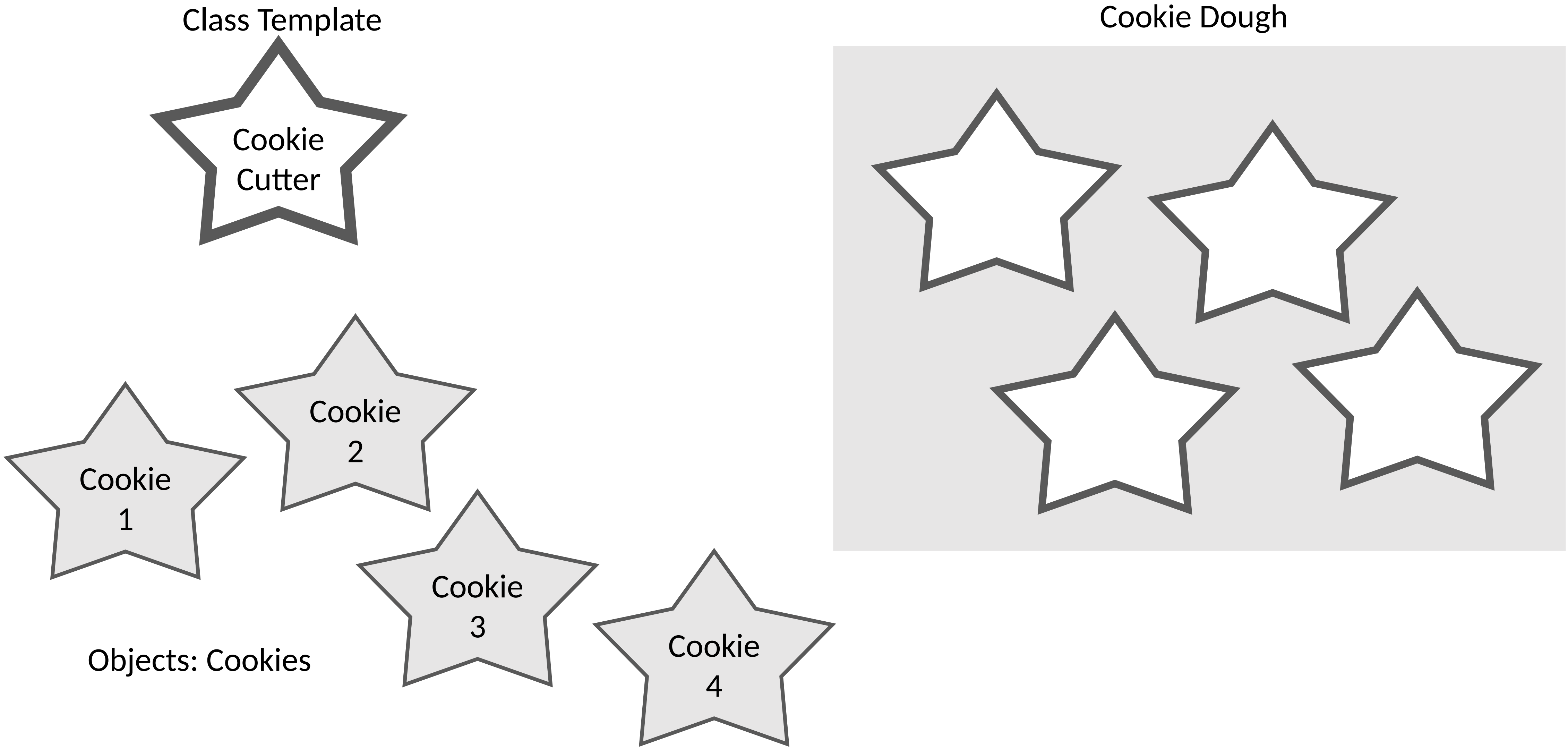
class Person

attributes: *string name, string dateOb, double weight, int height*

***methods**: SayName, Eat, GetAge, …*

- The **blueprint** is defined **once**—**multiple Objects** with similar features can be created

Classes and Objects



Class Template

Cookie Dough

Cookie Cutter

Cookie 1

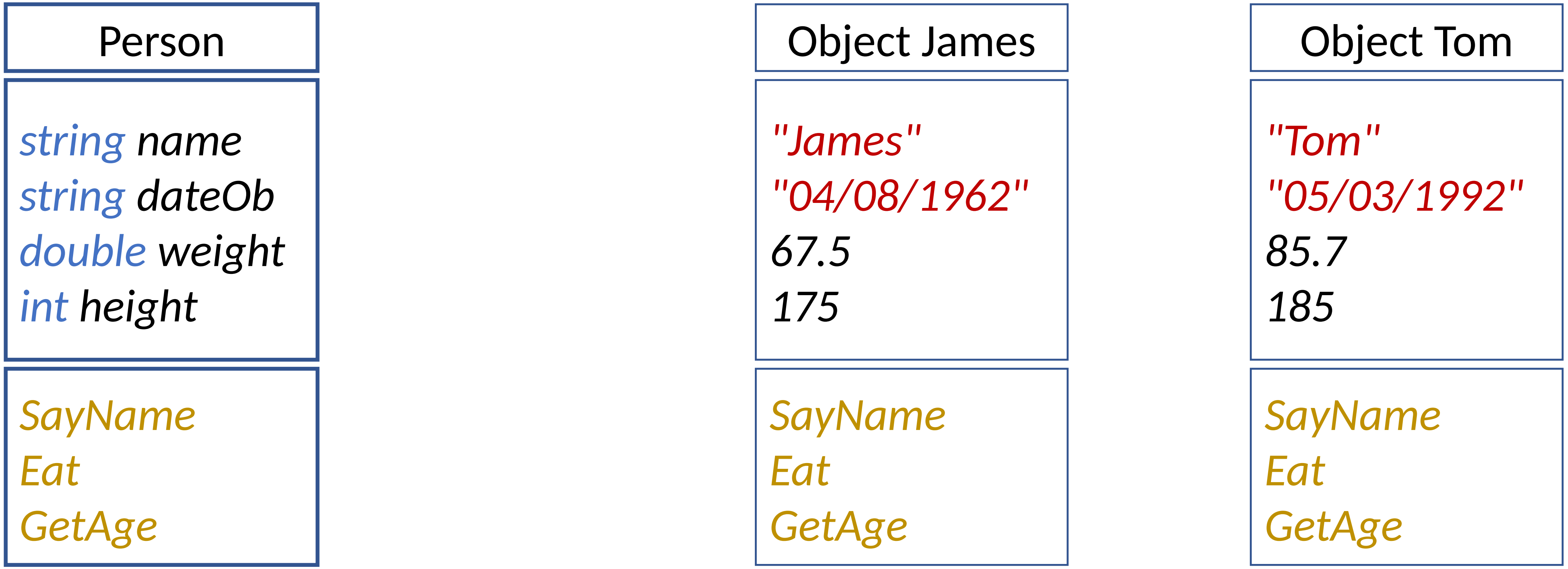
Cookie 2

Cookie 3

Cookie 4

Objects: Cookies

Classes and Objects



Person

string name
string dateOb
double weight
int height

SayName
Eat
GetAge

Object James

"James"
"04/08/1962"
67.5
175

SayName
Eat
GetAge

Object Tom

"Tom"
"05/03/1992"
85.7
185

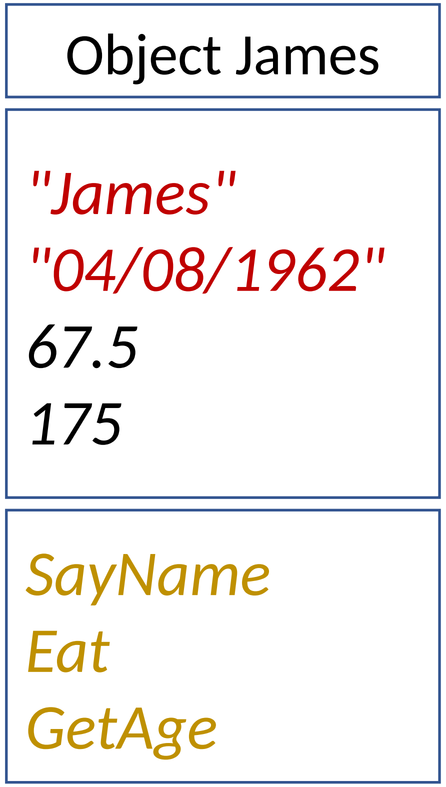
SayName
Eat
GetAge

Class Template

Objects: class instances

In OOP programs, the code we write defines classes

Objects Relationships



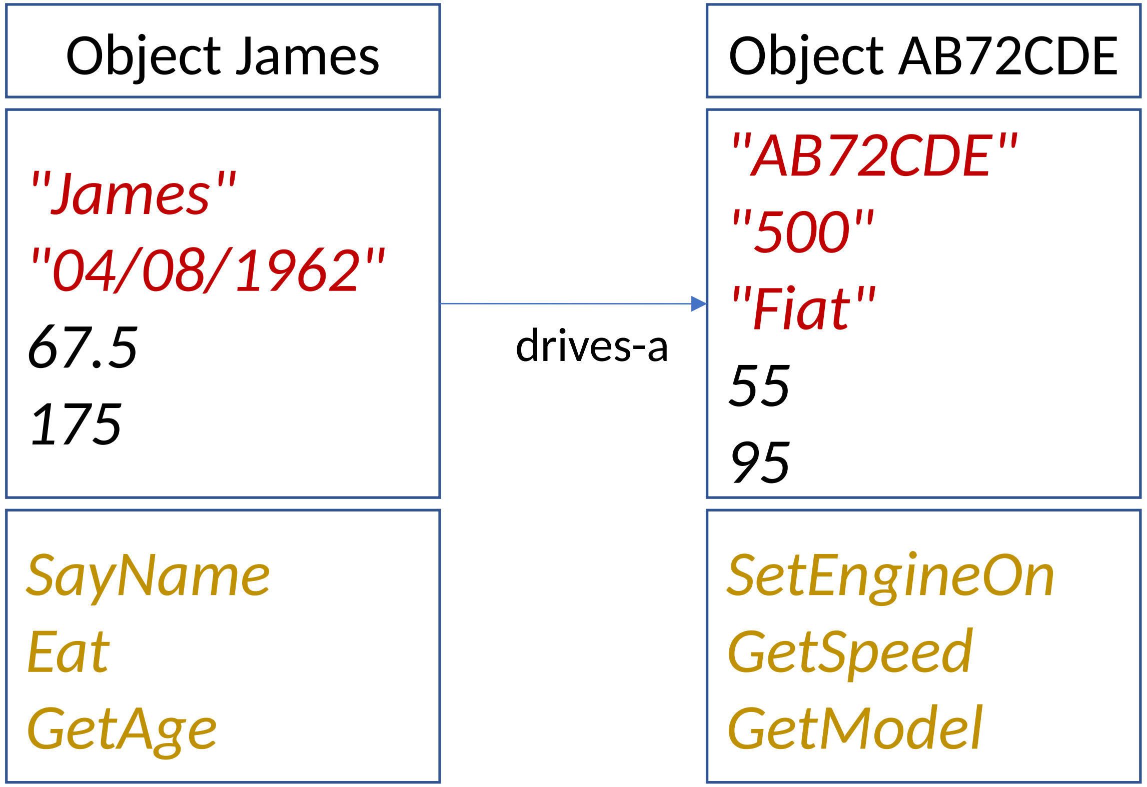
Object James

"James"
"04/08/1962"
67.5
175

SayName
Eat
GetAge

Person Object

Objects Relationships



Object James

"James"

"04/08/1962"

67.5

175

SayName

Eat

GetAge

drives-a

Object AB72CDE

"AB72CDE"

"500"

"Fiat"

55

95

SetEngineOn

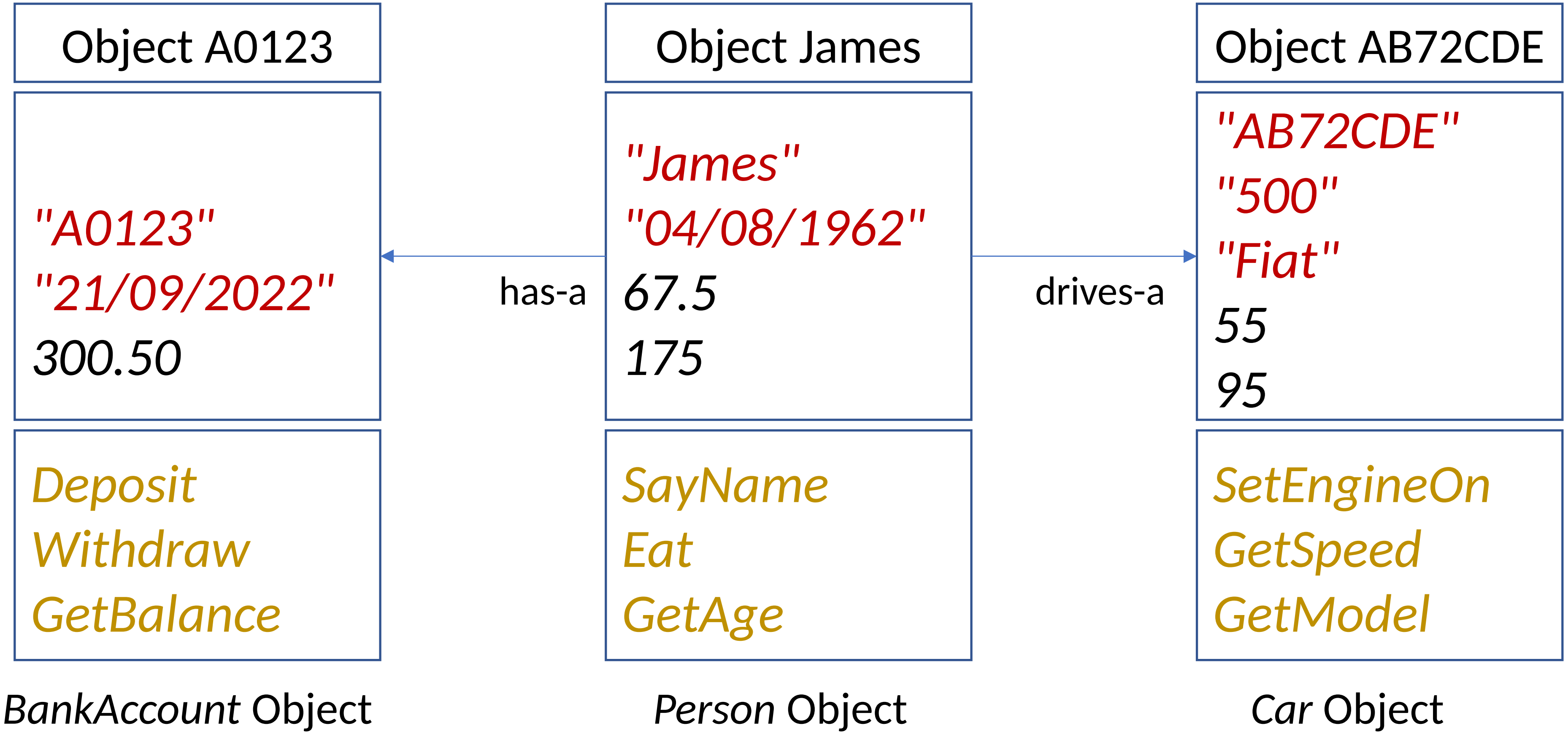
GetSpeed

GetModel

Person Object

Car Object

Objects Relationships



Object A0123

"A0123"
"21/09/2022"
300.50

Deposit
Withdraw
GetBalance

BankAccount Object

has-a

Object James

"James"
"04/08/1962"
67.5
175

SayName
Eat
GetAge

Person Object

drives-a

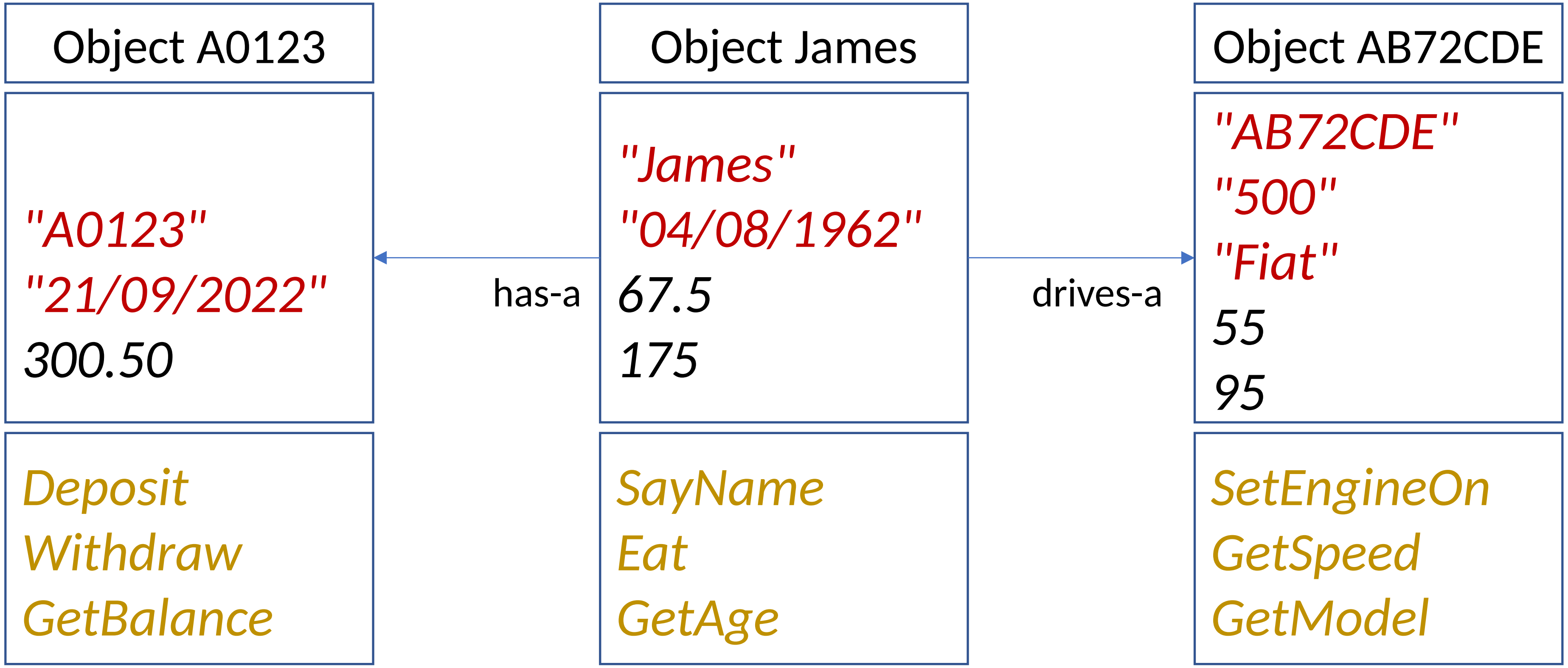
Object AB72CDE

"AB72CDE"
"500"
"Fiat"
55
95

SetEngineOn
GetSpeed
GetModel

Car Object

Objects Relationships



Object A0123

"A0123"
"21/09/2022"
300.50

Deposit
Withdraw
GetBalance

has-a

Object James

"James"
"04/08/1962"
67.5
175

SayName
Eat
GetAge

drives-a

Object AB72CDE

"AB72CDE"
"500"
"Fiat"
55
95

SetEngineOn
GetSpeed
GetModel

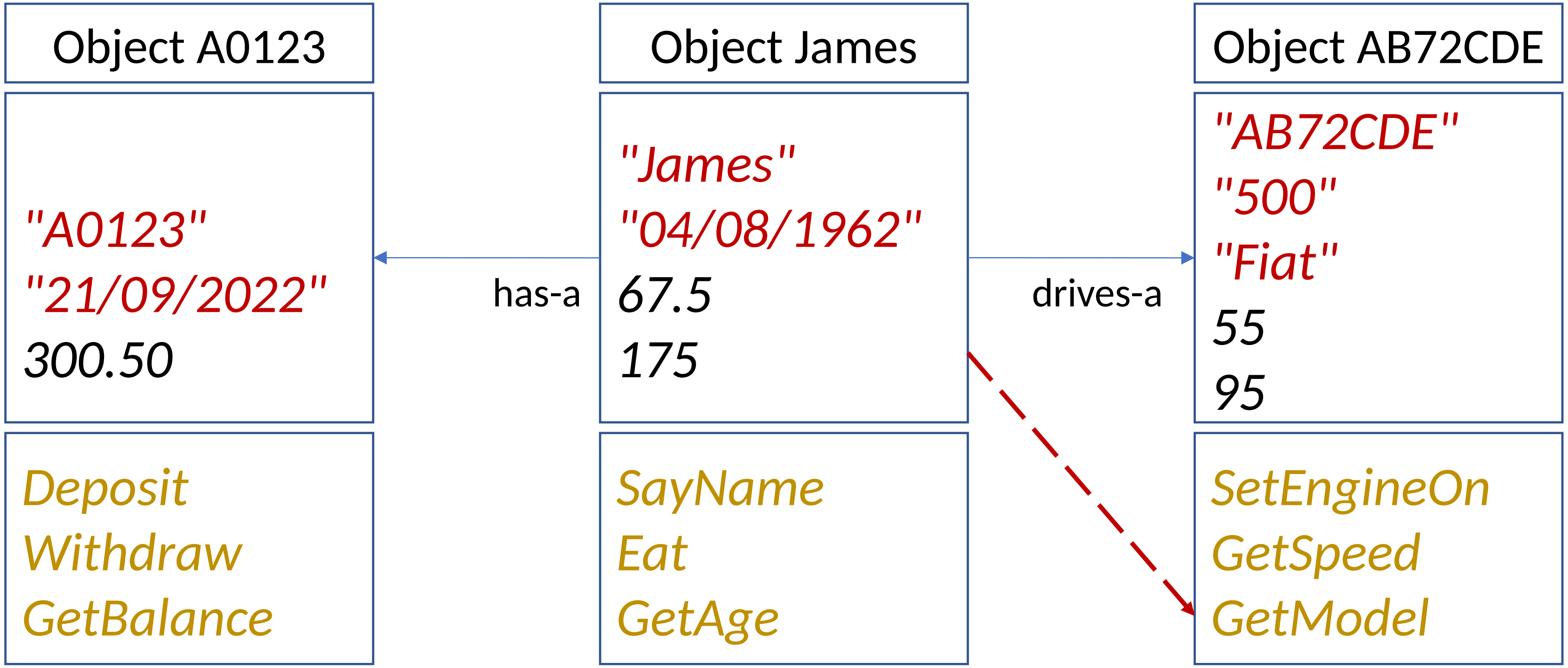
BankAccount Object

Person Object

Car Object

Each Object has its ***private*** attributes – they are **not shared** directly with other objects

Objects Relationships



Object A0123

"A0123"
"21/09/2022"
300.50

Deposit
Withdraw
GetBalance

BankAccount Object

has-a

Object James

"James"
"04/08/1962"
67.5
175

SayName
Eat
GetAge

Person Object

drives-a

Object AB72CDE

"AB72CDE"
"500"
"Fiat"
55
95

SetEngineOn
GetSpeed
GetModel

Car Object

An object interacts with another object by *sending a message* that triggers a **behaviour – method**

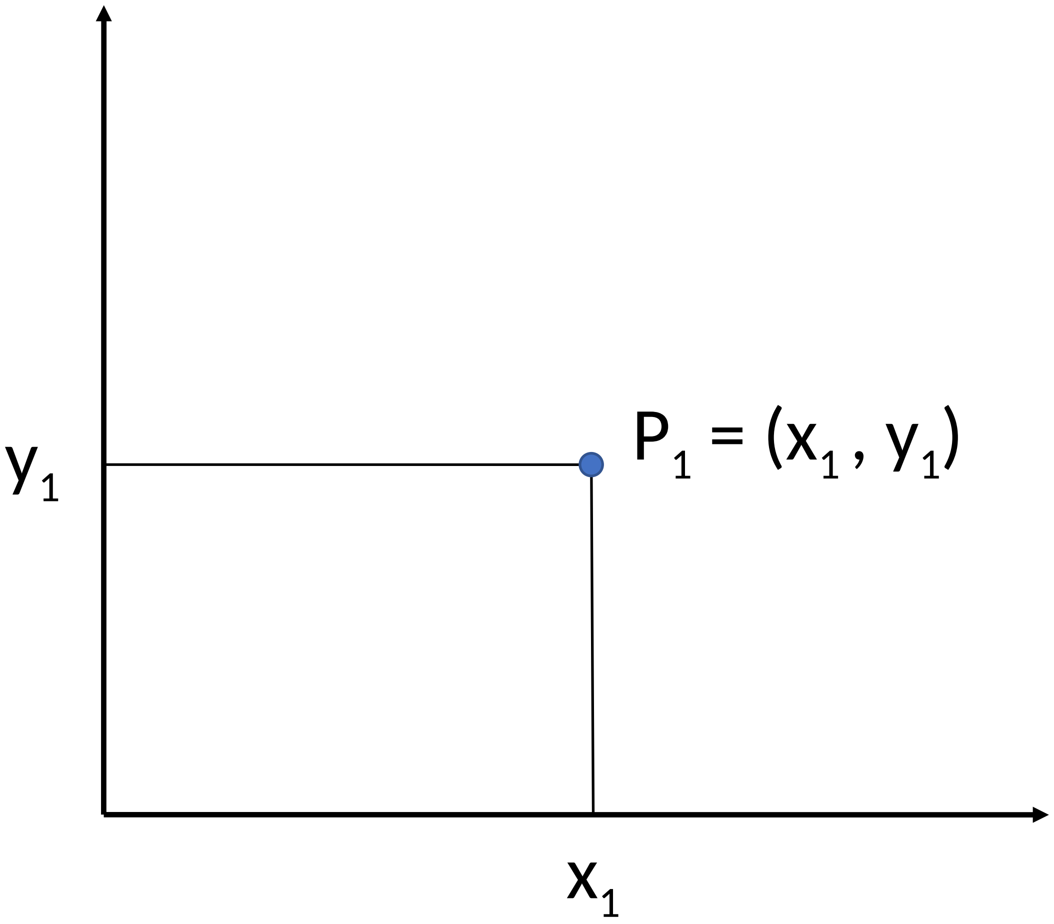
Outline

- Summary of last week
- Overview of Classes and Objects
- Our first OOP program: Point Class
- Object Oriented Programs Design

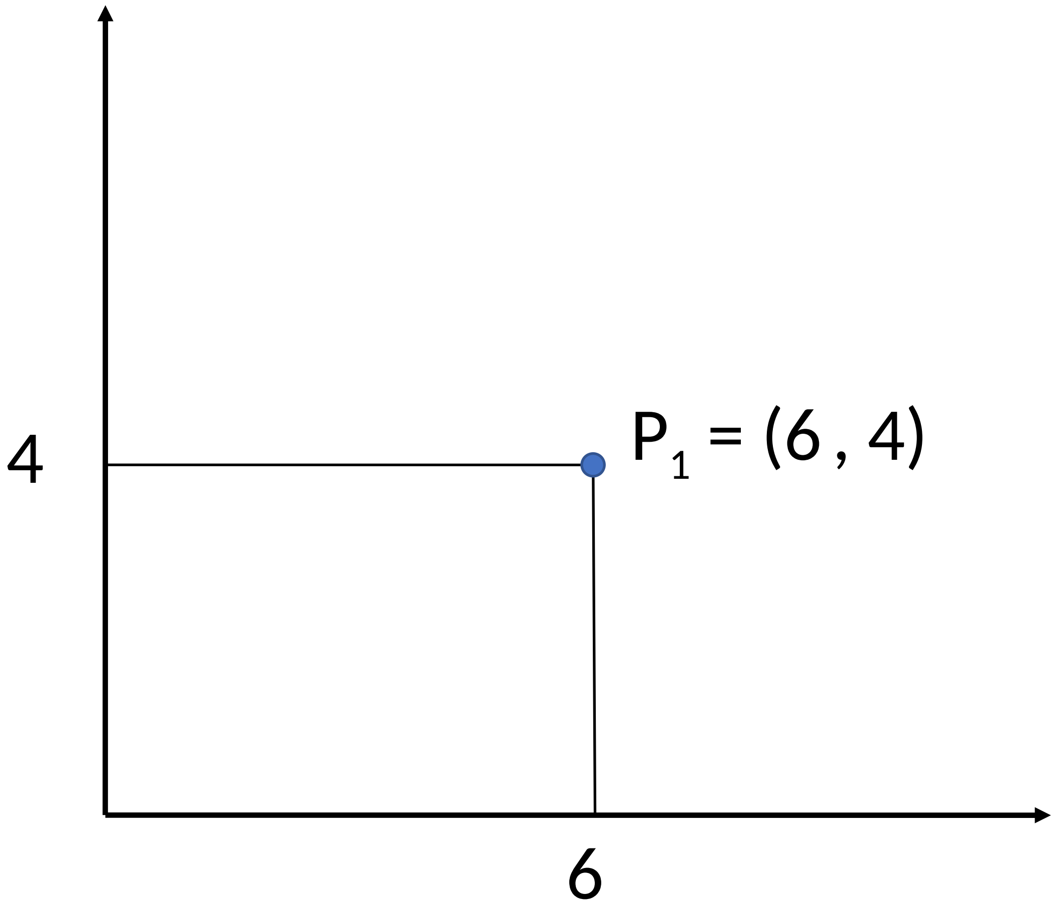
Class design: geometric shapes

- We understand classes, objects and relationships
- How do we use them inside a program?
- Let's try it!

Representing 2D points

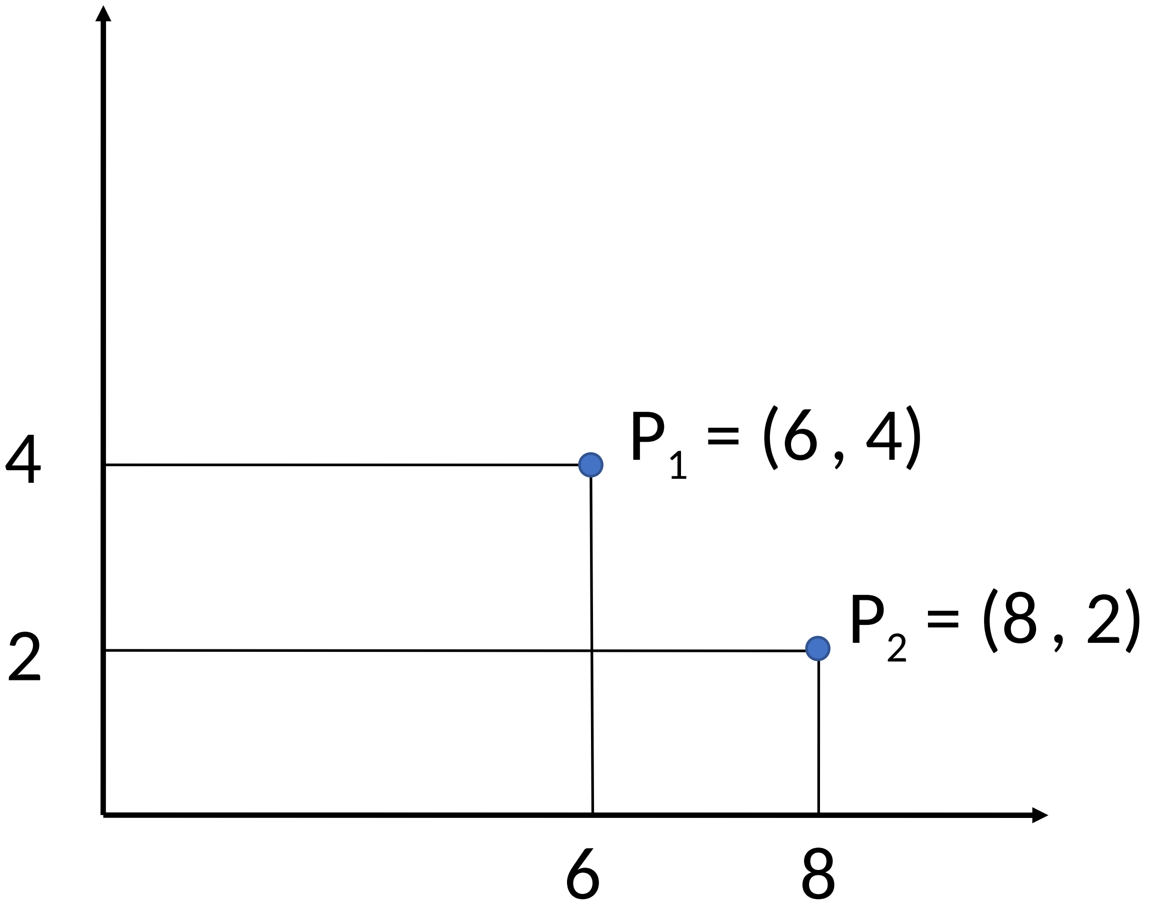


Representing 2D points: primitive data types



```
static void Main(string[] args)
{
        int p1X = 6;
        int p1Y = 4;



}
```

Representing 2D points: primitive data types



```
static void Main(string[] args)
{
        int p1X = 6;
        int p1Y = 4;

        int p2X = 8;
        int p2Y = 2;
}
```

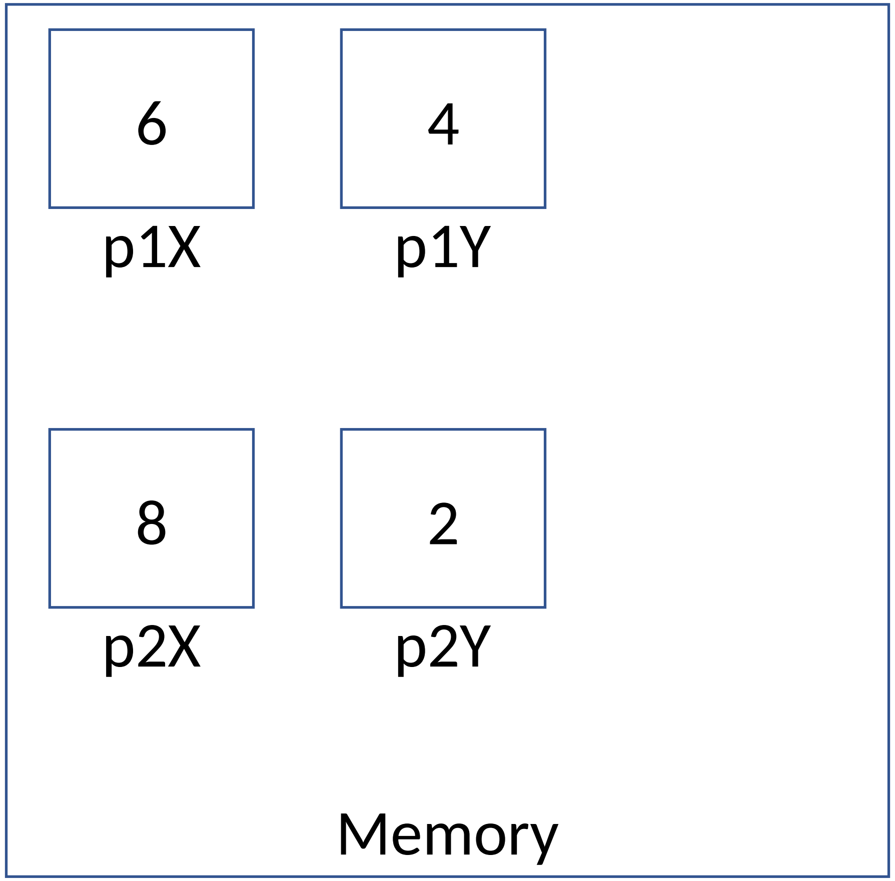
Representing 2D points: primitive data types

```
static void Main(string[] args)
{
        int p1X = 6;
        int p1Y = 4;

        int p2X = 8;
        int p2Y = 2;

        Console.WriteLine($"p1 = ({p1X} , {p1Y})");
        Console.WriteLine($"p2 = ({p2X} , {p2Y})");
}
```

Memory state and output

```
p1 = (6, 4)
p2 = (8, 2)
```

Console Output



Representing 2D points: primitive data types

```
static void Main(string[] args)
{
        int p1X = 6;
        int p1Y = 4;
        int p2X = 8;
        int p2Y = 2;
        // 497 more points ...
        int p500X = 3;
        int p500Y = 2;

                Console.WriteLine($"p1 = ({p1X} , {p1Y})");
                Console.WriteLine($"p2 = ({p2X} , {p2Y})");
                // 497 more WriteLine
        Console.WriteLine($"p500 = ({p500X} , {p500Y})");
}
```

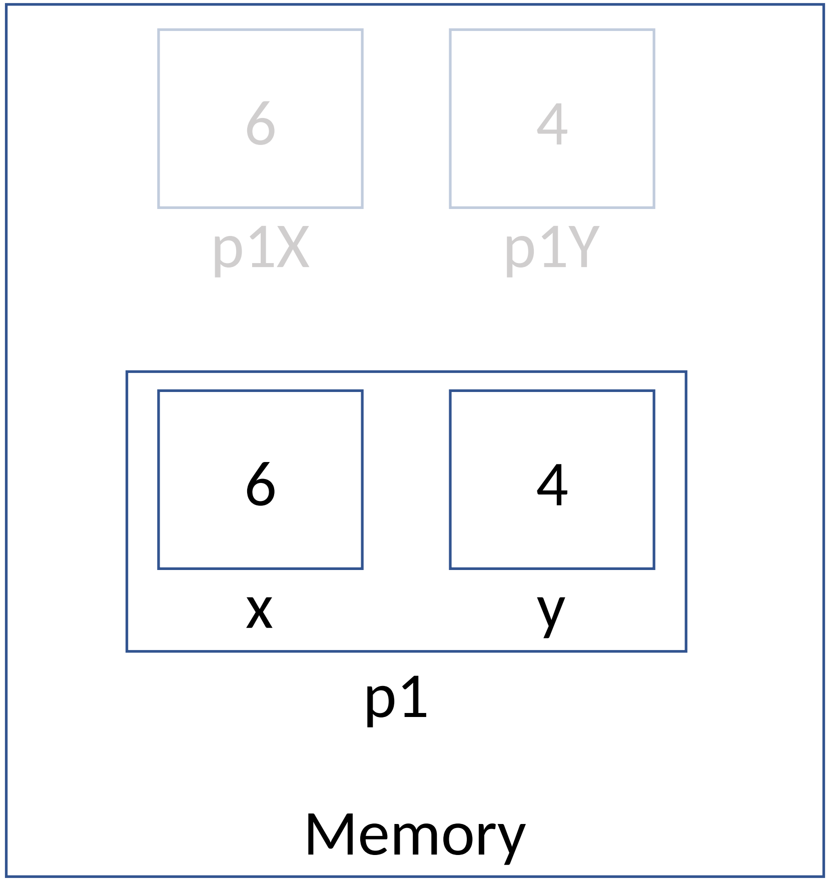
500 points?!

Representing 2D points: primitive data types

```
static void Main(string[] args)
{
        int p1X = 6;
        int p1Y = 4;
        int p2X = 8;
        int p2Y = 2;
        // 497 more points ...
        int p500X = 3;
        int p500Y = 2;

                Console.WriteLine($"p1 = ({p1X} , {p1Y})");
                Console.WriteLine($"p2 = ({p2X} , {p2Y})");
                // 497 more WriteLine
        Console.WriteLine($"p500 = ({p500X} , {p500Y})");
}
```

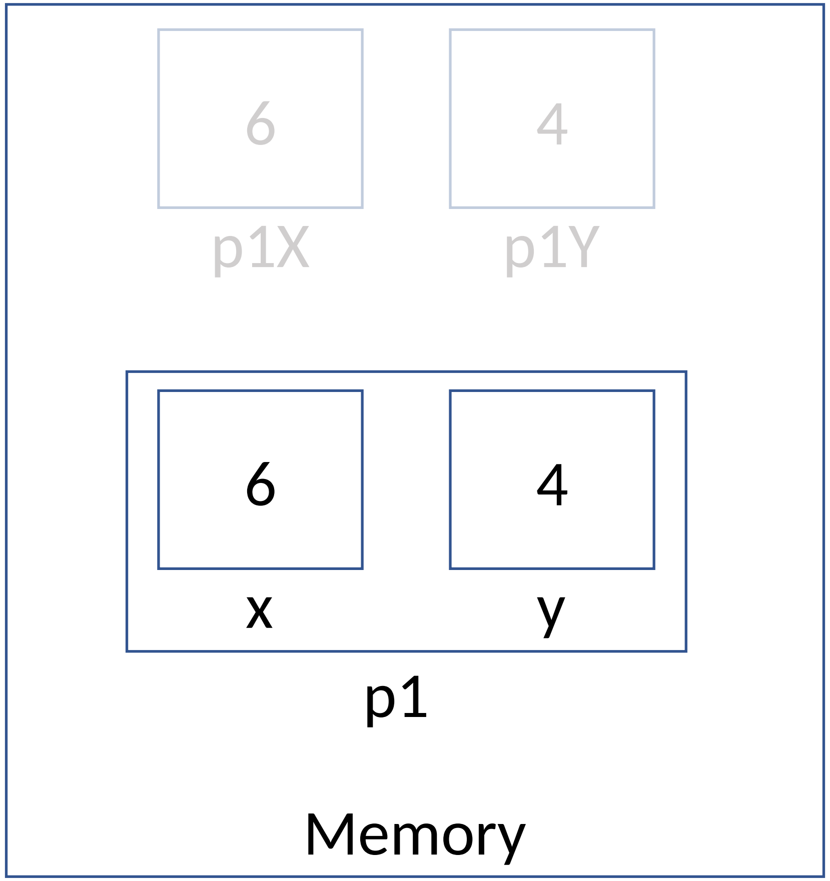
What if instead of using *int* variables we allocate and use objects of a class *Point* in our program?

Object of the Point class



Instead of using two separate int variables for the *x* and *y* coordinates, we can **aggregate** them as part of a single Point object

Object of the Point class



every Point object will have these two *attributes* *x* and *y*

and a behaviour *Display* to print the coordinates

Representing 2D points: with Objects

```
static void Main(string[] args)
{
        Point p1 = new Point(6, 4);          // setting state
        Point p2 = new Point(8, 2);

        p1.Display();                        // using behaviour
        p2.Display();


}
```

What if instead of using *int* variables, we allocate and use objects of a class *Point* in our program? Good idea!

Representing 2D points: with Objects

```
static void Main(string[] args)
{
        Point[] points = new Point[500];
        …

}
```

500 points?! Done!

Representing 2D points: with Objects

```
static void Main(string[] args)
{
        Point[] points = new Point[500];
        ...

}
```

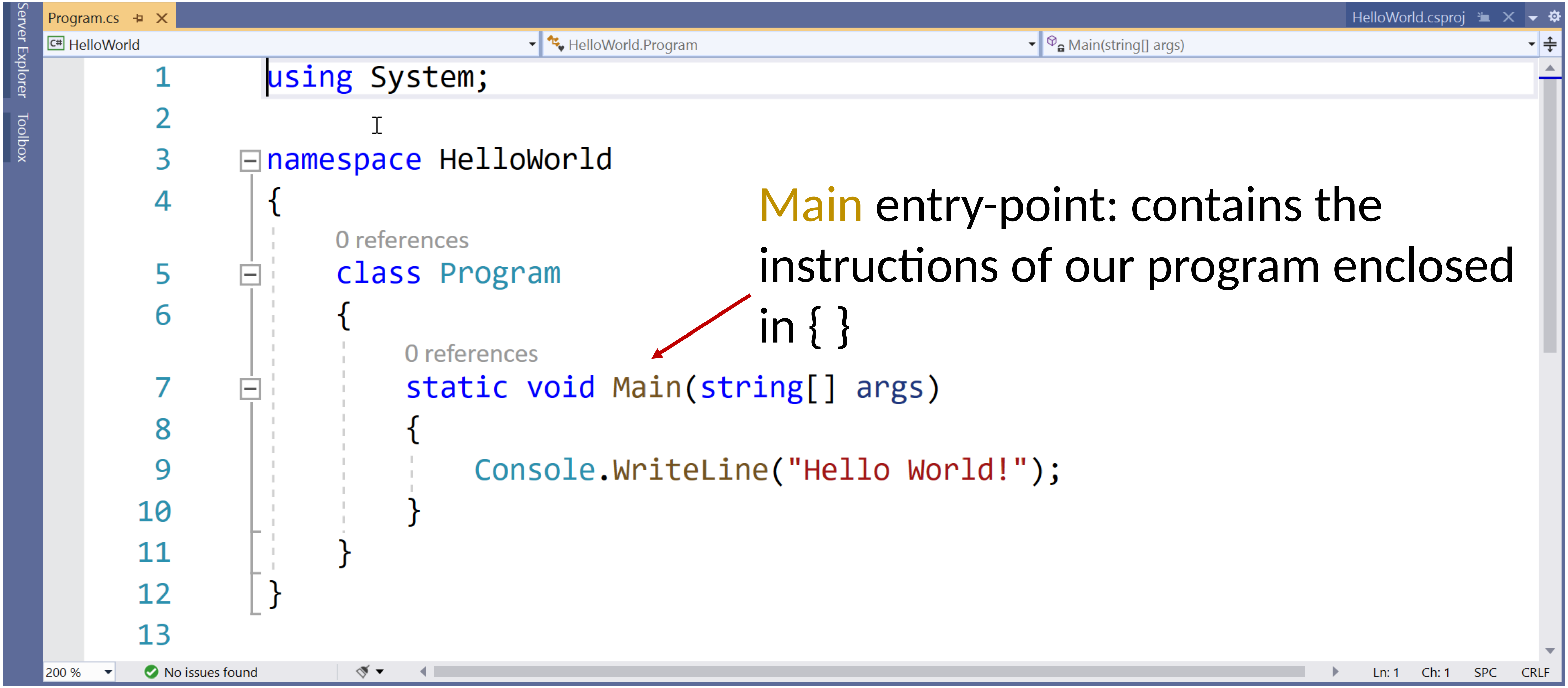
Let's write the Point class code!

classes and Objects

- A *class* is a **template** for **Objects**
- A *class* allows the definition of a **custom type**
- An **Object** is a *"variable"* of that custom type

- ***How can we define a class in C#?***

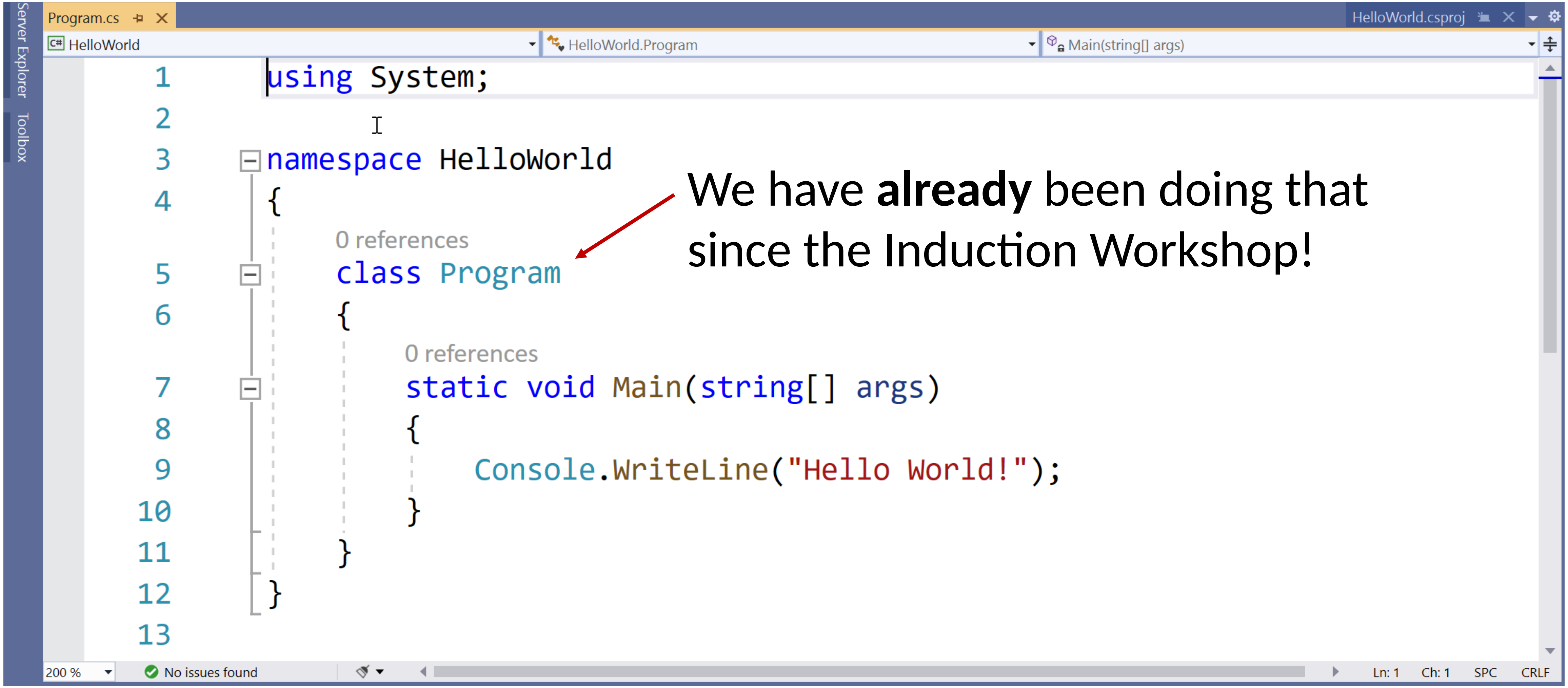
classes and Objects



```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}

```

Main entry-point: contains the instructions of our program enclosed in { }

classes and Objects



```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}

```

We have **already** been doing that since the Induction Workshop!

Creating a new class

- Let’s assume we still use the usual program template
- But we create an additional file inside our project
- This file will contain the definition of the new class

Point class: definition

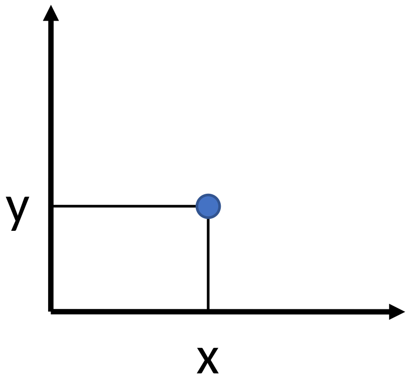
```
class Point
{
        int x;
        int y;




}
```

keyword class followed by the *name* of the class we are defining

Point class: attributes

```
class Point
{
        int x;
        int y;



}
```

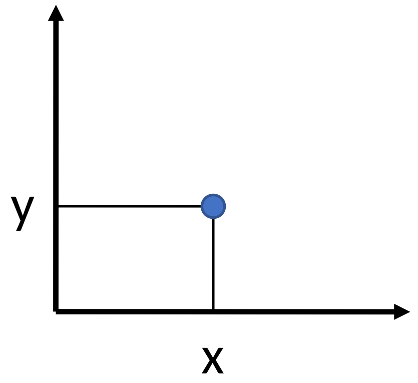
attributes: variables that define the characteristics of a Point
in this case *x* and *y* are the *cartesian coordinates* of a Point



Point class: attributes

```
class Point
{
        int x;
        int y;


}
```

attributes: variables that define the characteristics of a Point
in this case *x* and *y* are the *cartesian coordinates* of a Point



What **objects** should be able to **access** the **attributes directly**?

Point class: attributes

```
class Point
{
        private int x;
        private int y;




}
```

private: those **attributes** will **only** be **directly accessible** from methods of **objects** of the Point **class**

class operations: methods

- A class not only defines data as *attributes*
- It also defines the *operations* that are performed on those attributes

- These *operations* (*behaviours*) are called ***methods***

Point class: methods

```
class Point
{
        private int x;
        private int y;

        public void Display()
        {
          Console.WriteLine($"({x}, {y})");
        }
}
```

The method defines a block of instructions enclosed in { }

In this example, *Display*() prints on the screen the content of the attributes *x* and *y* **defined in the same class**

Point class: methods

```
class Point
{
        private int x;
        private int y;

        public void Display()
        {
          Console.WriteLine($"({x}, {y})");
        }
}
```

defined **once** in the class—
available in **all** the *Point* objects

Point class: methods

```
class Point
{
        private int x;
        private int y;

        public void Display()
        {
          Console.WriteLine($"({x}, {y})");
        }
}
```

public: makes the method accessible to (i.e., can be "called") objects of classes other than *Point*

Point class: methods

```
class Point
{
        private int x;
        private int y;

        public void Display()
        {
          Console.WriteLine($"({x}, {y})");
        }
}
```

public: makes the method accessible to (i.e., can be "called") objects of classes other than *Point*

How can a new object of the *Point* class be created?

Point class: attributes initialisation

```
class Point
{
   private int x;
   private int y;




   public void Display()
   {
      Console.WriteLine($"({x}, {y})");
   }
}
```

No values were assigned to *x* and *y*.

0 is assigned by default by the C# compiler to uninitialised integer **attributes**

Point class: attributes initialisation

```
class Point
{
  private int x;
  private int y;

  public Point()
  {
    x = 6;
    y = 4;
  }

  public void Display()
  {
    Console.WriteLine($"({x}, {y})");
  }
}
```

A ***constructor*** method is used specifically for attribute *initialisation*

It must have the *same name* as the class

Point class: attributes initialisation

```
class Point
{
   private int x;
   private int y;

   public Point()
   {
      x = 6;
      y = 4;
   }

   public void Display()
   {
     Console.WriteLine($"({x}, {y})");
   }
}
```

It is used to create a new **object** instance of that class

Creating Point Objects

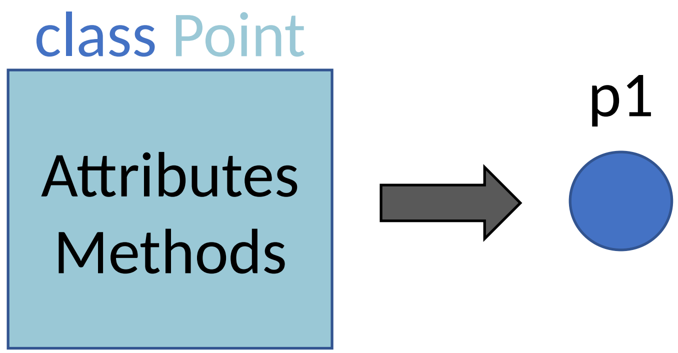
```
static void Main(string[] args)
{
    Point p1 = new Point();
}
```

The ***constructor*** is "called" after the **new** operator

```
public Point()
{
    x = 6;
    y = 4;
}
```

Creating Point Objects

```
static void Main(string[] args)
{
    Point p1 = new Point();



}
```



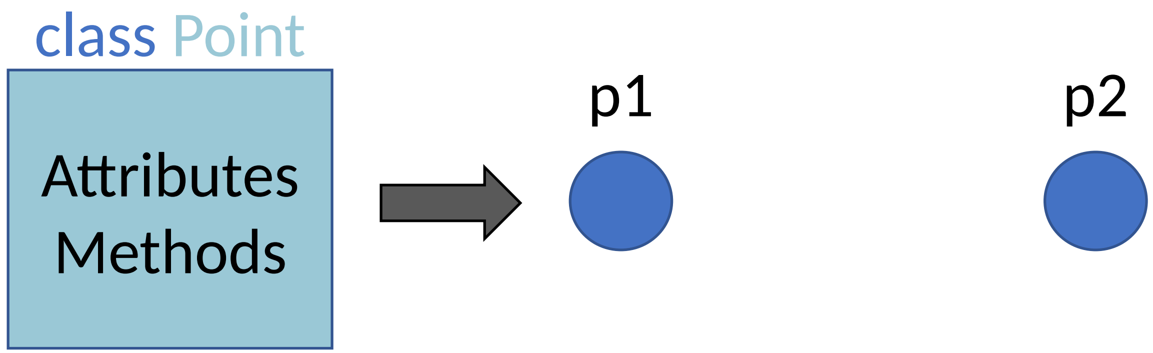
class Point

Attributes
Methods

Object
Instances

Creating Point Objects

```
static void Main(string[] args)
{
    Point p1 = new Point();
    Point p2 = new Point();



}
```



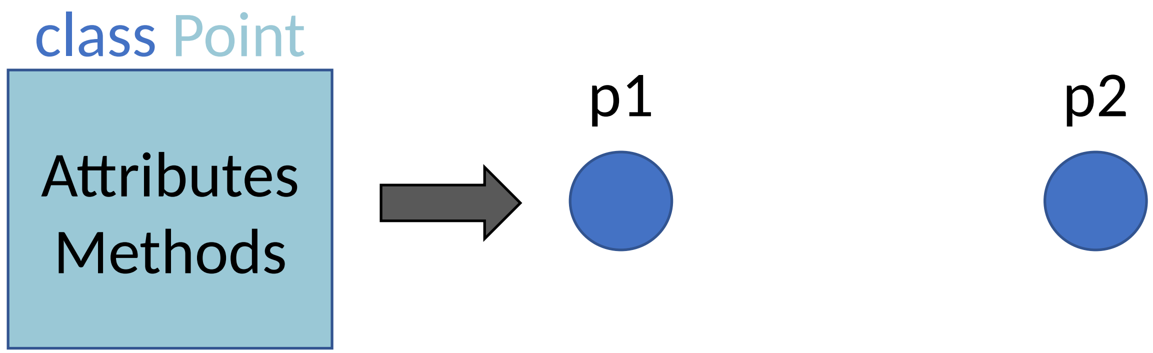
class Point

Attributes
Methods

Object
Instances

Creating Point Objects

```
static void Main(string[] args)
{
    Point p1 = new Point();
    Point p2 = new Point();



}
```

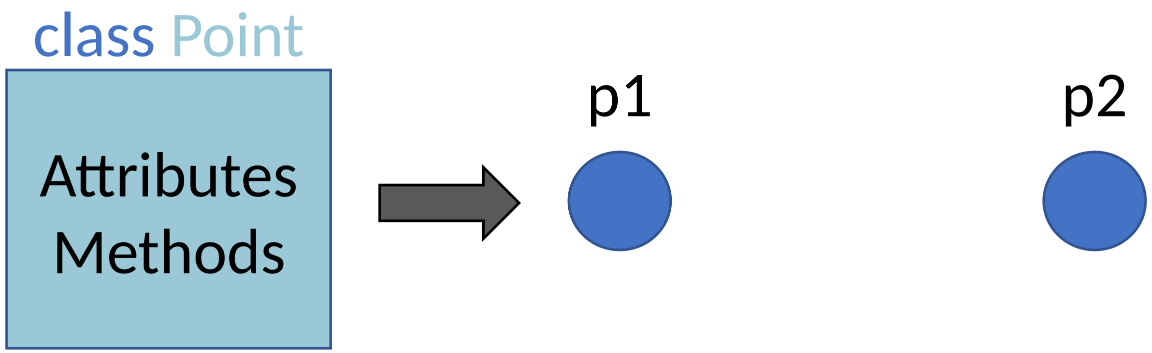


There is now a *relationship* between (the Main of) the *Program* class and the *Point* objects *p1* and *p2*

Object Instances

Creating Point Objects

```
static void Main(string[] args)
{
    Point p1 = new Point();
    Point p2 = new Point();



}
```



class Point

Attributes
Methods

p1

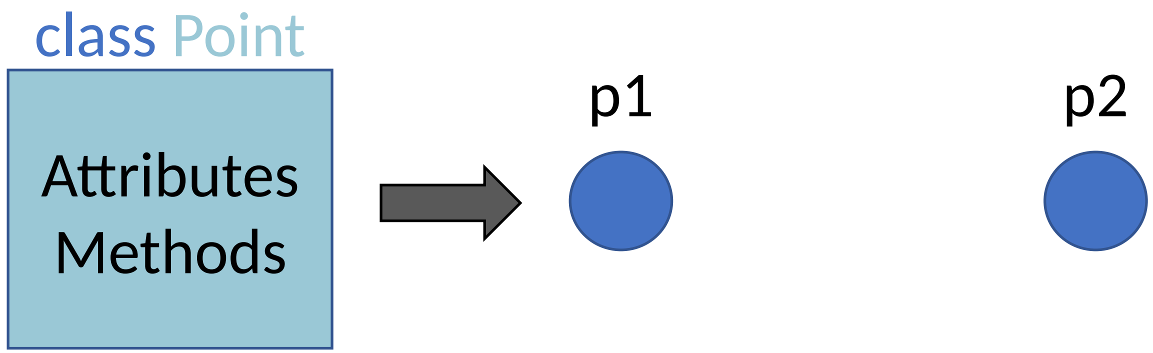
p2

Program can *send a message* to those objects to trigger one of their public *behaviours*

Object
Instances

Creating Point Objects

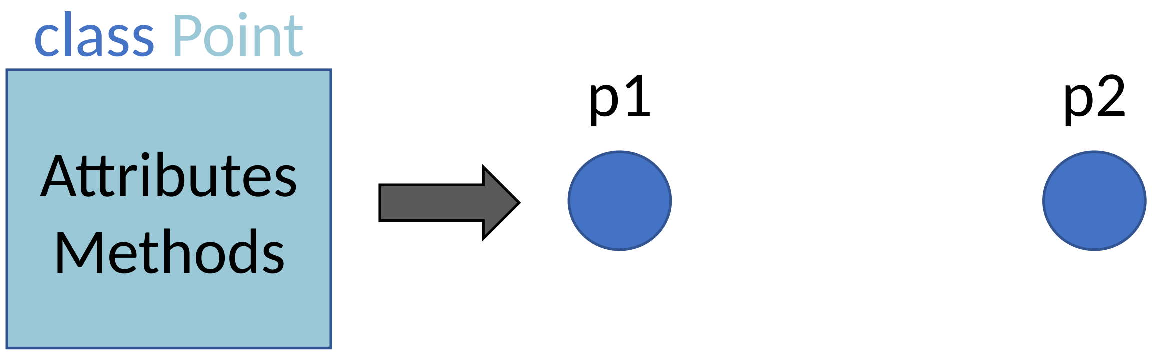
```
static void Main(string[] args)
{
    Point p1 = new Point();
    Point p2 = new Point();



}
```



The *Point* class defines the public behaviour *Display*()

Creating Point Objects

```
static void Main(string[] args)
{
    Point p1 = new Point();
    Point p2 = new Point();

    p1.Display();
    p2.Display();
}
```

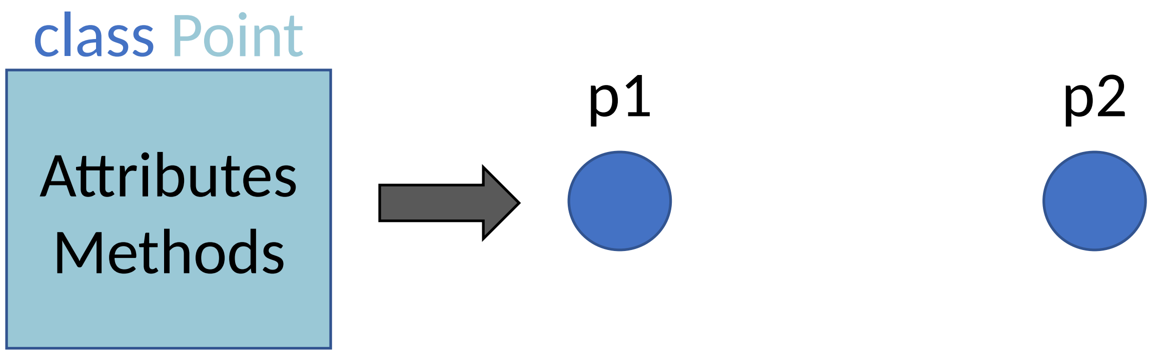


The **dot notation (.)** is used to *send a message* to those objects and **invoke** a method—*Display*()

Object Instances

Creating Point Objects

```
static void Main(string[] args)
{
    Point p1 = new Point();
    Point p2 = new Point();

    p1.Display();
    p2.Display();
}
```



??

```
public void Display()
{
   Console.WriteLine($"({x}, {y})");
}
```

```
public Point()
{
     x = 6;
     y = 4;
}
```

Object
Instances

Point class: constructor

- Multiple *Object instances* of the Point class
- Each instance is **independent** of the other ones
- It is allocated in a **separate** *area of memory*
- Each instance has **its values** for the *attributes x* and *y*

Method's Parameters

Any method can receive some *values* as *input*:

public Point(int xarg, int yarg)

- **Parameters** are like *placeholders* (variables) defined in a method to receive *input values*
- Each parameter has a *type* and an *identifier*
- *xarg* and *yarg* above, but any valid identifier can be used
- We saw this for the *Main* last week – *Main*(string[] args)

Method's Parameters

```
class Point
{
    private int x;
    private int y;

    public Point(int xarg, int yarg )
    {
        x = xarg;
        y = yarg;
    }

    public void Display()
    {
        Console.WriteLine($"({x}, {y})");
    }
}
```

Parameters become *local variables* that a method can access and use

Method's Arguments

```
class Point
{
    private int x;
    private int y;

    public Point(int xarg, int yarg )
    {
        x = xarg;
        y = yarg;
    }

    public void Display()
    {
        Console.WriteLine($"({x}, {y})");
    }
}
```

```
class Program {
    static void Main(string[] args)
    {
        int a = 6, b = 4;
        Point p1 = new Point(a, b);
    }
}
```

When a caller invokes the *method*, it provides *concrete values*, called **arguments**, for each **parameter**.

The arguments must be compatible with the parameter type

Method's Arguments

```
class Point
{
    private int x;
    private int y;

    public Point(int xarg, int yarg )
    {
        x = xarg; // 6
        y = yarg;
    }

    public void Display()
    {
        Console.WriteLine($"({x}, {y})");
    }
}
```

6

```
class Program {
    static void Main(string[] args)
    {
        int a = 6, b = 4;
        Point p1 = new Point(a, b);
    }
}
```

A **copy** of the arguments, the content of the variables *a* and *b*, is passed to the method—the Point constructor

The behaviour of the constructor now depends on the provided arguments

Method's Arguments

```
class Point
{
    private int x;
    private int y;

    public Point(int xarg, int yarg )
    {
        x = xarg; // 6
        y = yarg; // 4
    }

    public void Display()
    {
        Console.WriteLine($"({x}, {y})");
    }
}
```

4

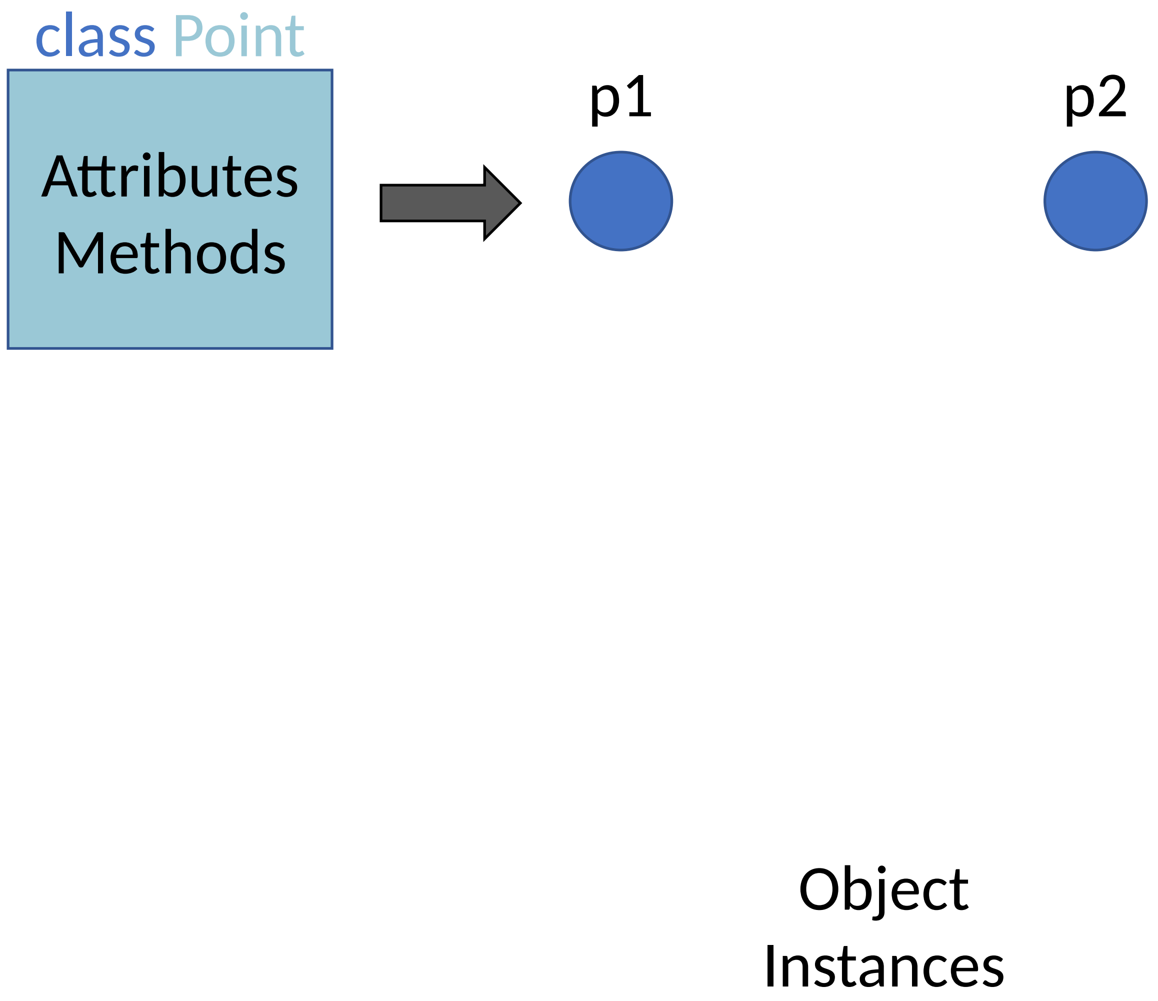
```
class Program {
    static void Main(string[] args)
    {
        int a = 6, b = 4;
        Point p1 = new Point(a, b);
    }
}
```

A **copy** of the arguments, the content of the variables *a* and *b*, is passed to the method—the Point constructor

The behaviour of the constructor now depends on the provided arguments

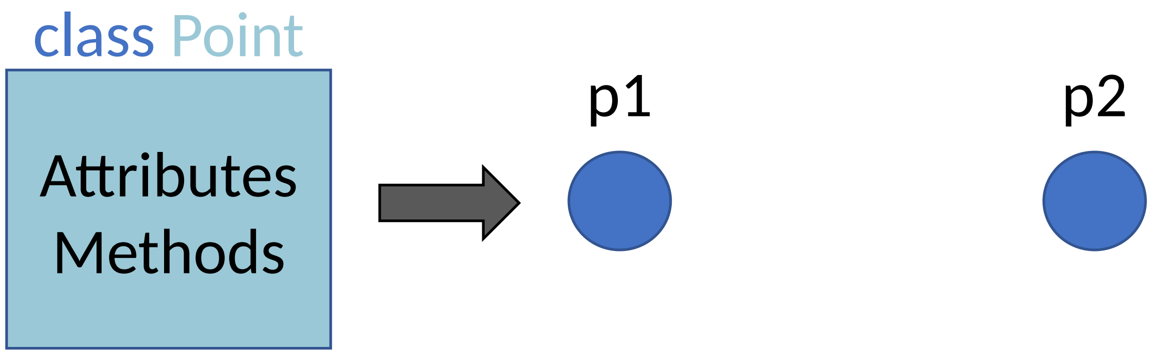
Creating Point Objects

```
static void Main(string[] args)
{
    Point p1 = new Point(6, 4);
    Point p2 = new Point(8, 2);




}
```



Creating Point Objects

```
static void Main(string[] args)
{
   Point p1 = new Point(6, 4);
   Point p2 = new Point(8, 2);

   // what is the output now?
   p1.Display();
   p2.Display();
}
```



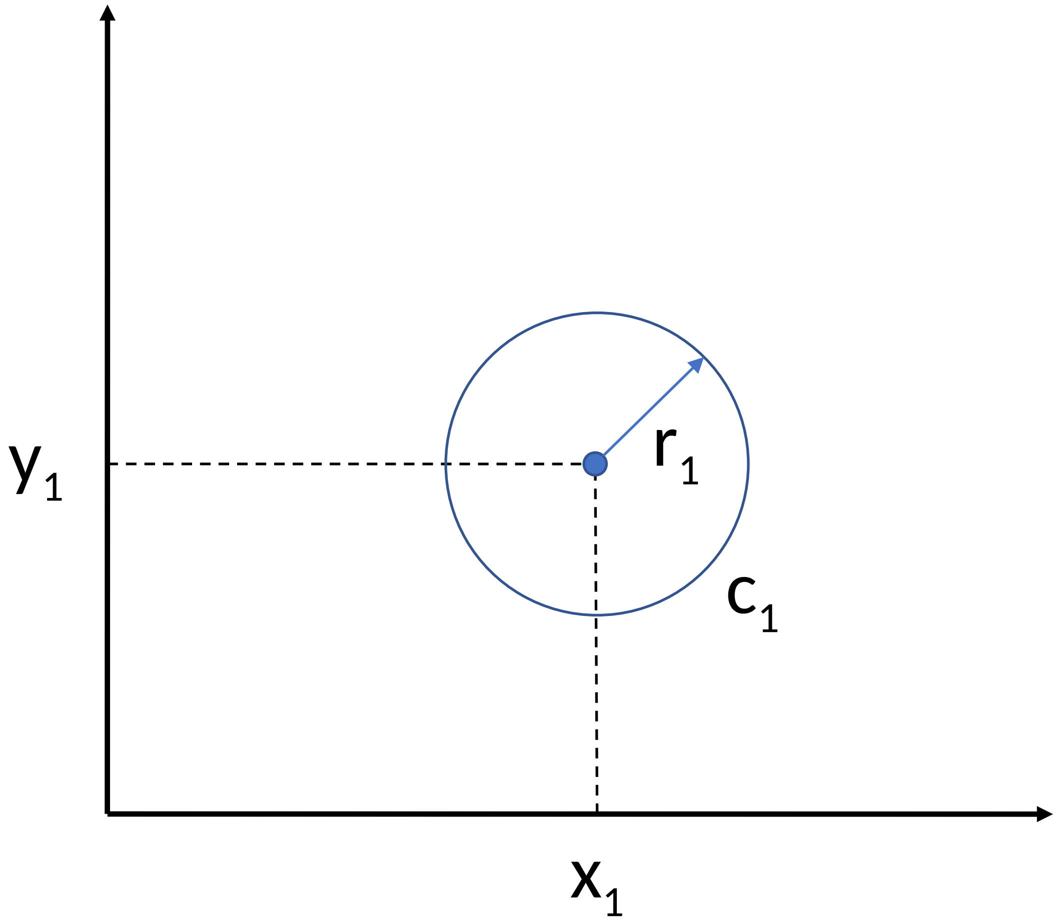
Object
Instances

Outline

- Summary of last week
- Overview of Classes and Objects
- Our first OOP program: Point Class
- Object Oriented Programs Design

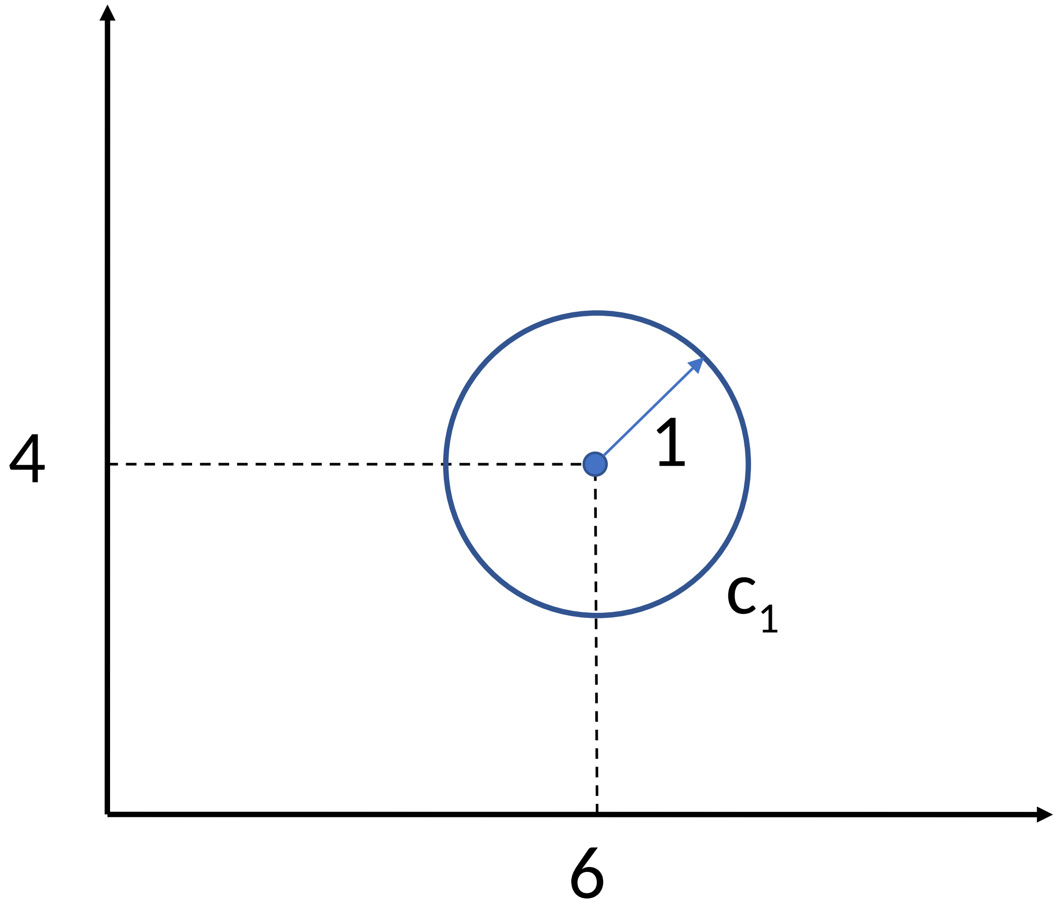
Object Oriented Programs: modules

Circle object—what are the **attributes**?



Object Oriented Programs: modules

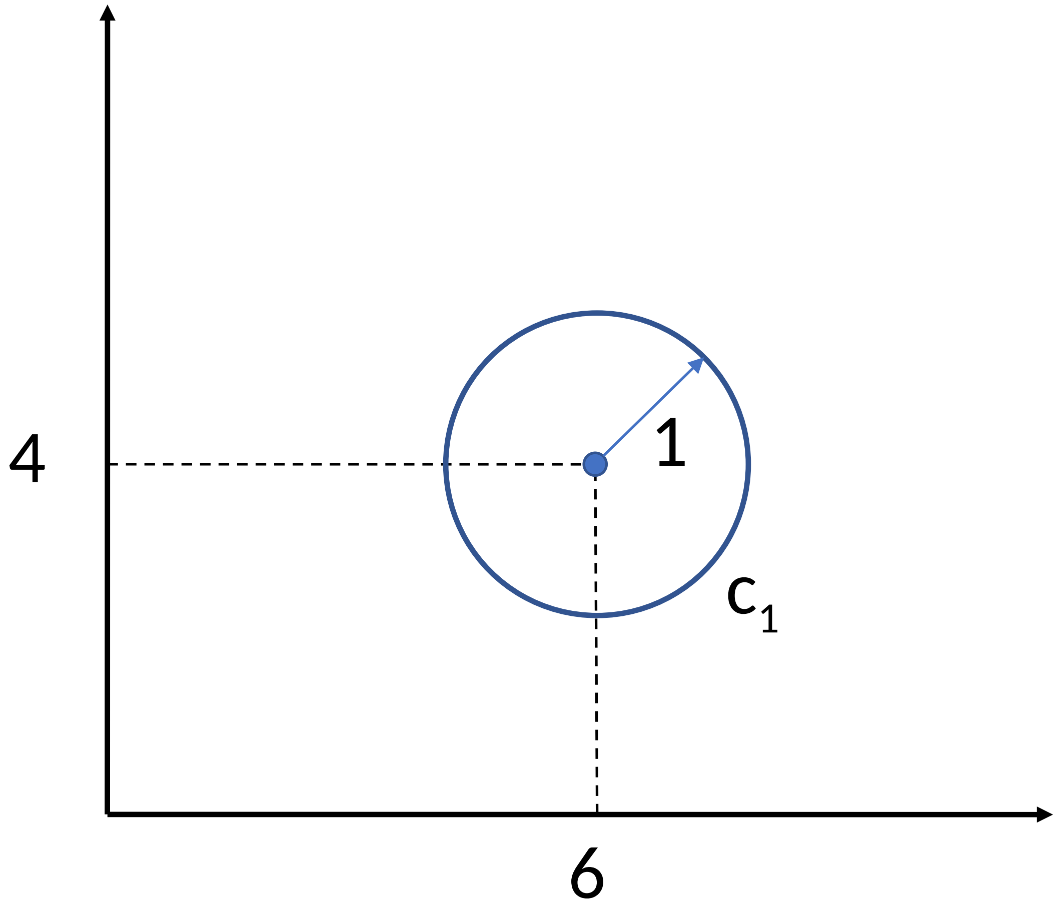
```
Point p1 = new Point(6, 4);
int r1 = 1;

Circle c1 = new Circle(p1, r1)
c1.Area();
```



Object Oriented Programs: modules

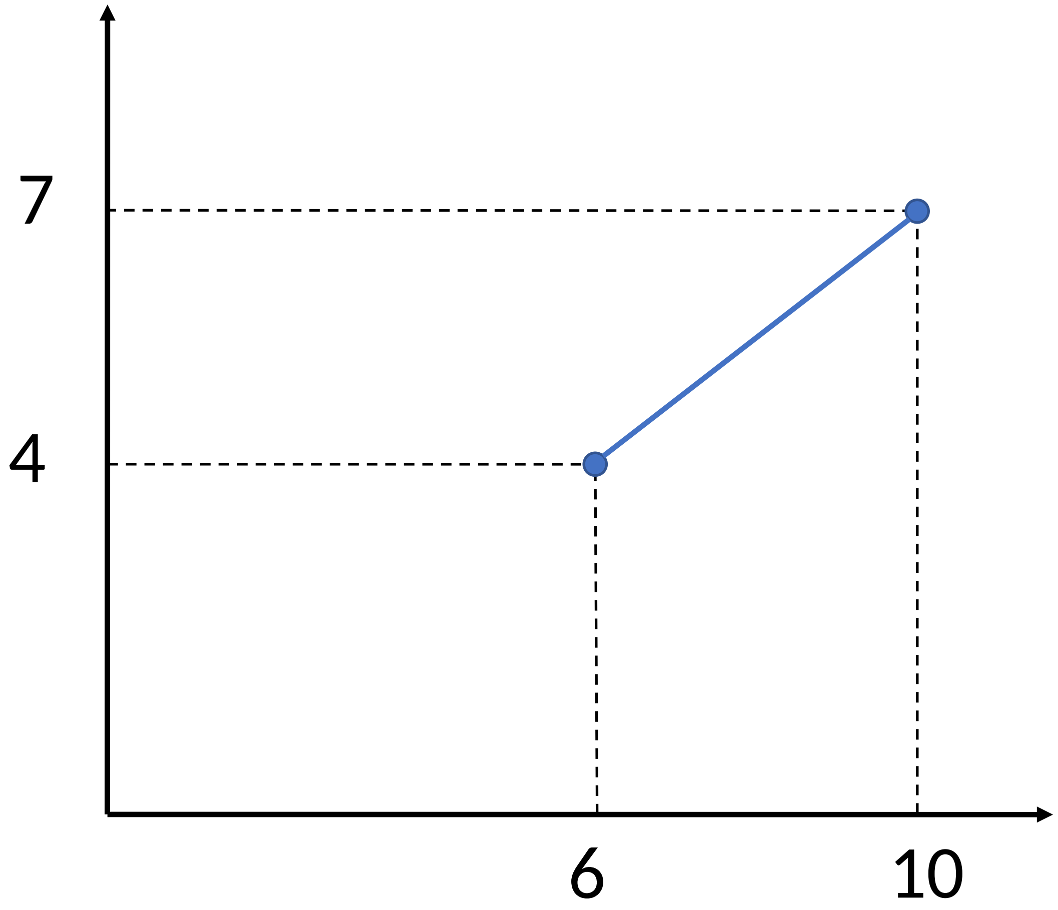
```
Point p1 = new Point(6, 4);
int r1 = 1;

Circle c1 = new Circle(p1, r1)
c1.Area();
```

By using a *Point* object as its centre, a *Circle* does not need to reimplement the functionalities already provided by a *Point*



Object Oriented Programs: modules

```
Point p1 = new Point(6, 4);
Point p2 = new Point(10, 7);

Segment s1 = new Segment(p1, p2)
s1.Length();
```



Object Oriented Programs: modules

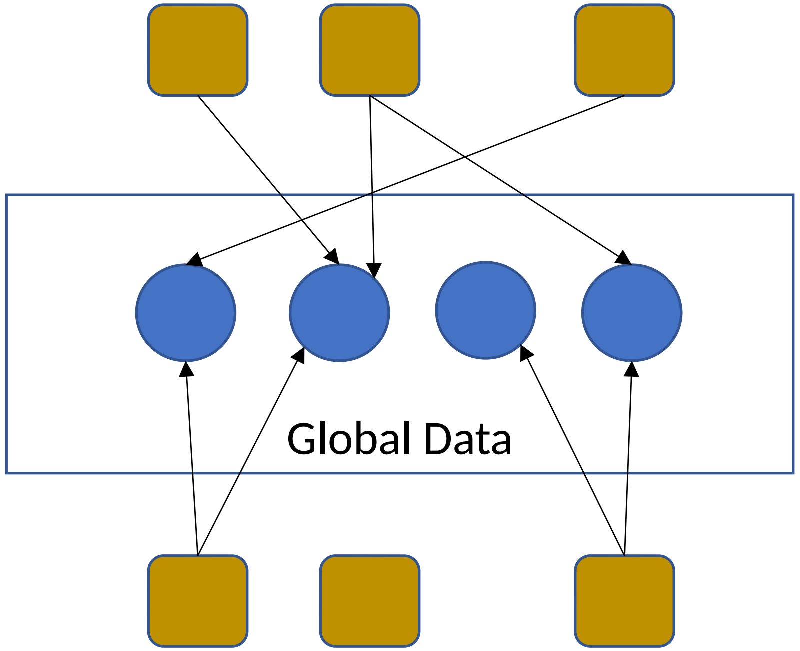
```
class Point { … }                          class Segment { … }

              class Circle { … }                            class XXX { … }
```

```
class Program
{
    static void Main(string[] args)
    {
        Point p1 = new Point(6, 4);
        Point p2 = new Point(10, 7);
        int r1 = 1;

        Circle c1 = new Circle(p1, r1)
        Segment s1 = new Segment(p1, p2)
    }
}
```

Each class is in effect a module of the program we are writing

Object Oriented Programs: modules

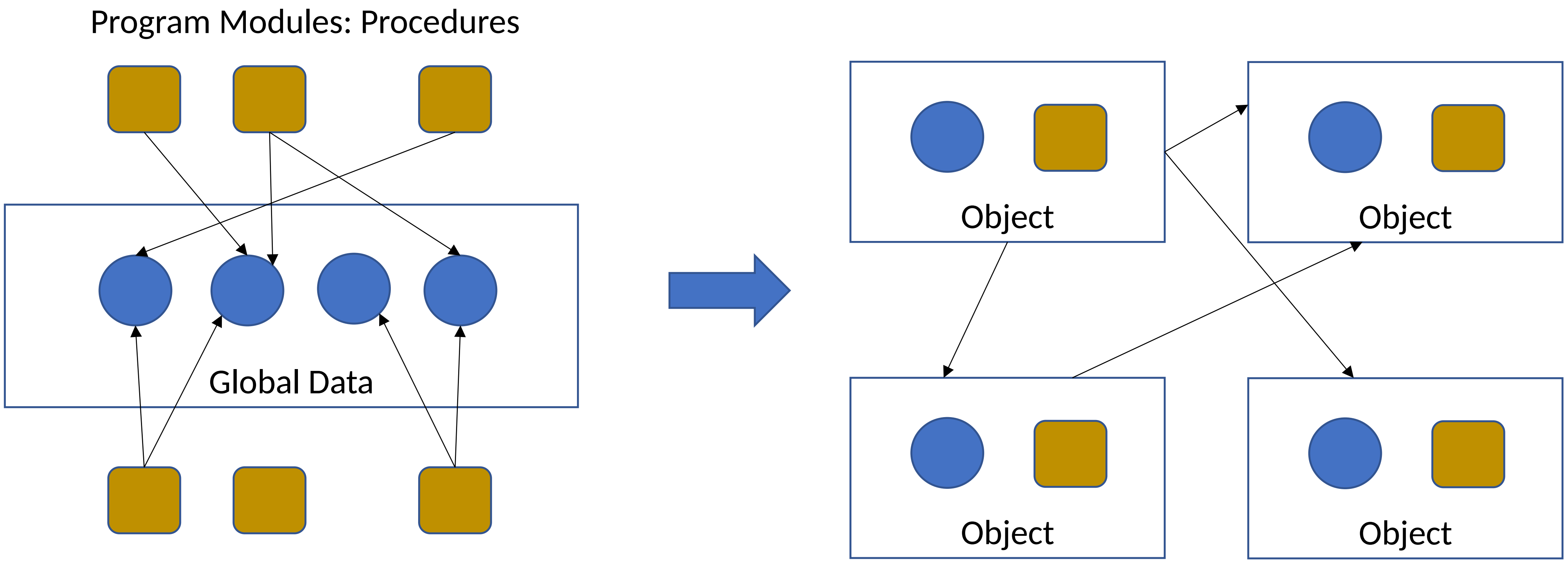
Program Modules: Procedures



Global Data

Different operations (procedures) *share*, *access* and *modify* **global data**

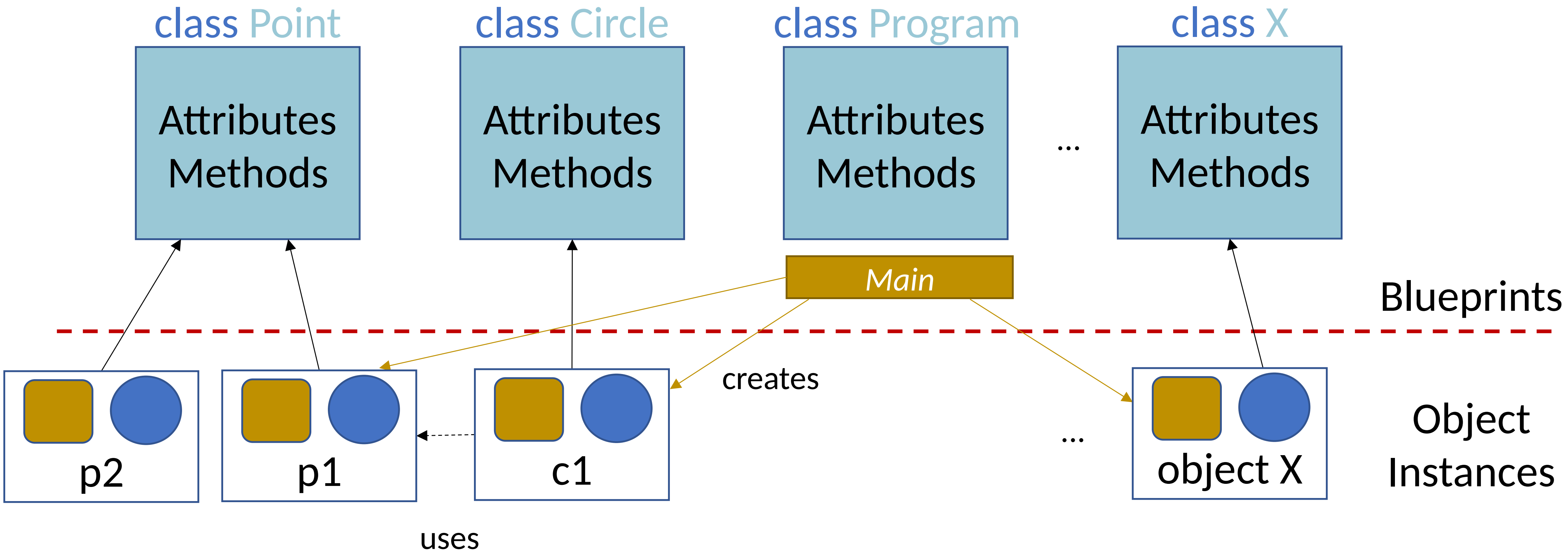
Object Oriented Programs: modules



Different operations (procedures) *share*, *access* and *modify* **global data**

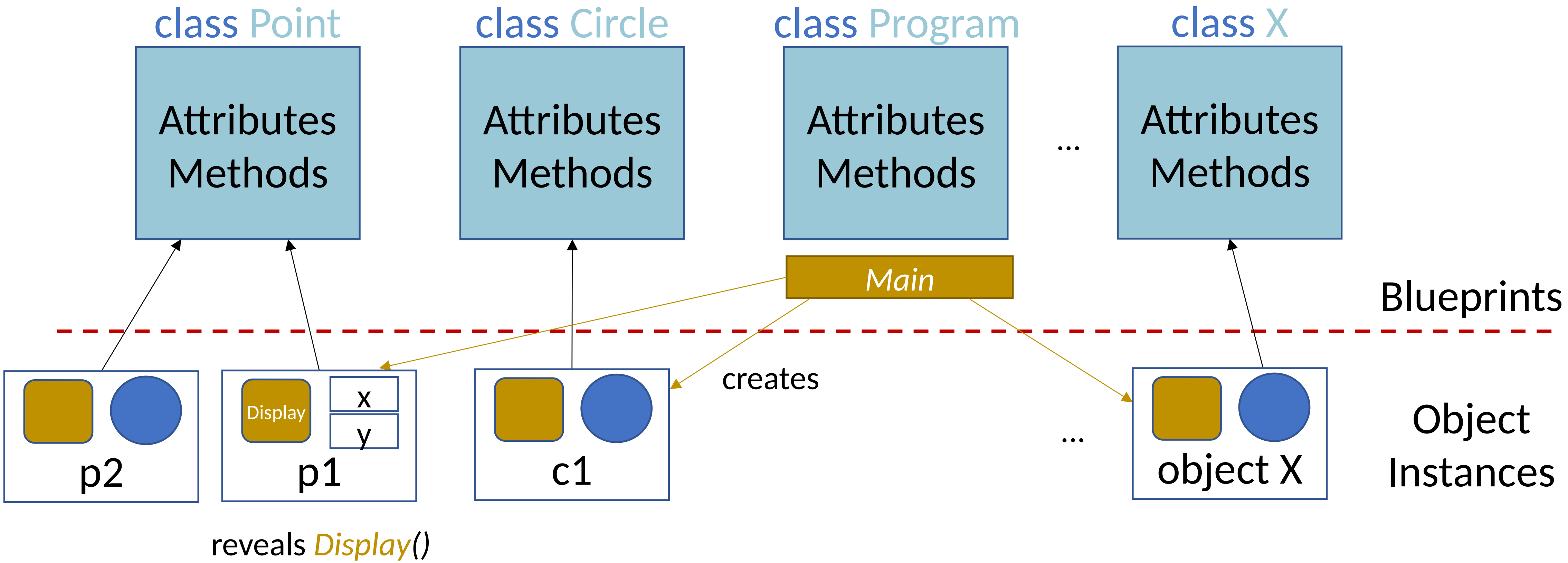
Attributes (data) and *operations* (methods) encapsulated within each object

Object Oriented Programs: modules



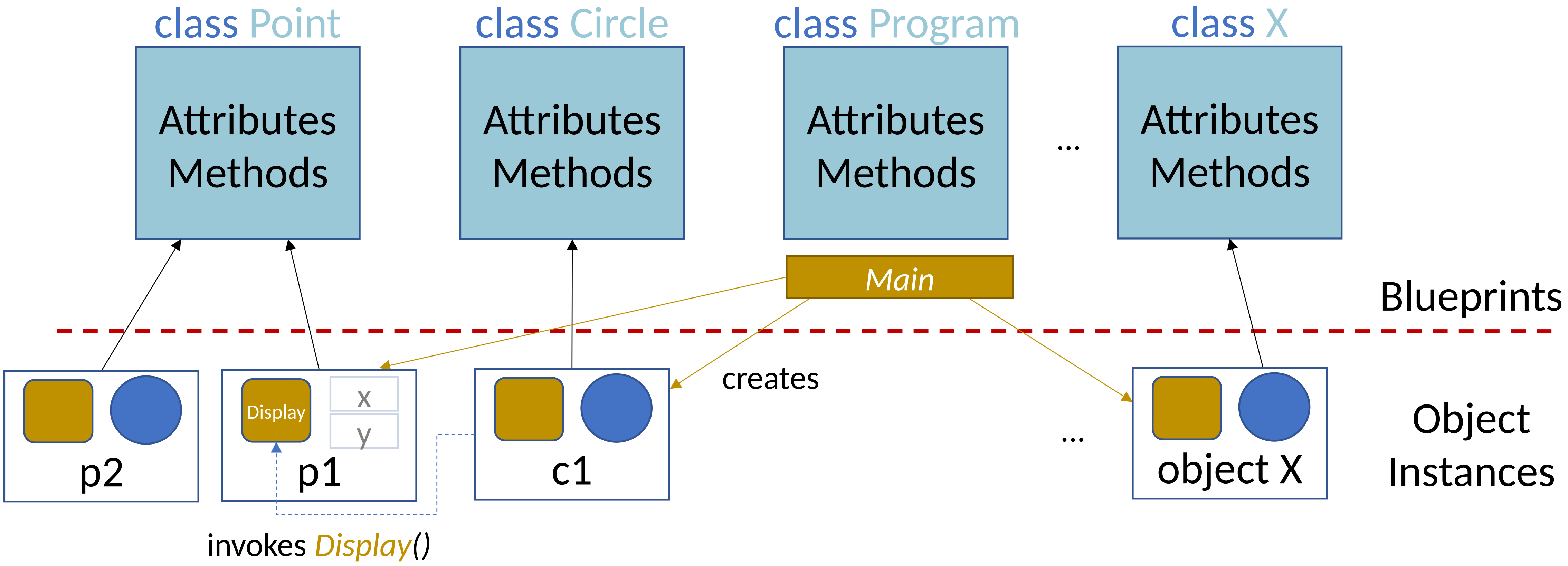
In the Main() of the Program class, the *c1* and *p1* objects are created—*c1* uses *p1* as its centre

Object Oriented Programs: modules



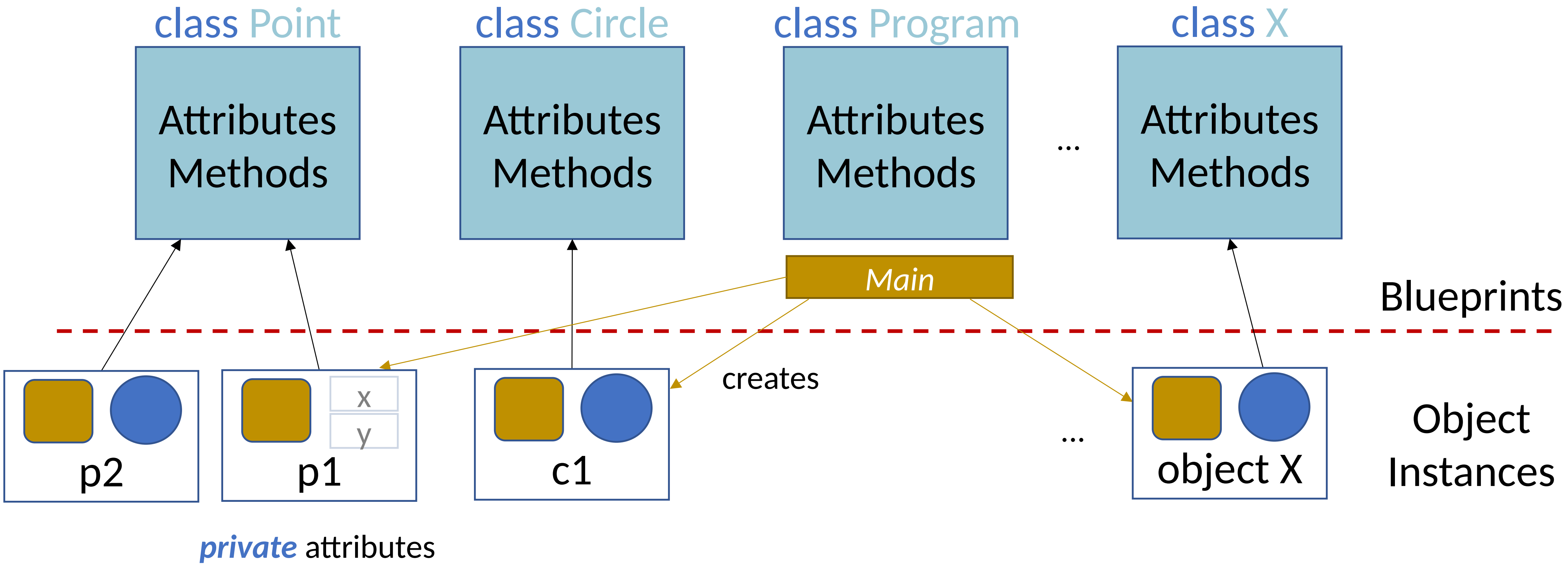
The *Point* object *p1 reveals* the public method *Display*() to other objects—they can **use it without** needing to **know how** *Display*() **works**

Object Oriented Programs: modules



The *Circle* object *c1* can invoke *Display*() to **ask** the *Point* object *p1* to print the coordinates of its centre—*c1* does **not need direct access** to the attributes *x* and *y*

Object Oriented Programs: modules



Objects have *attributes* and *methods* **encapsulated**—*p1* **does not expose** *x* and *y* to *c1* **directly**—*x* and *y* are private

Object-Oriented Programming (OOP) Principles

- **Abstraction**
- **Encapsulation**
- **Inheritance**
- **Polymorphism**

Objects **contain attributes and behaviours**— they can **control** how these are **accessed** and **hide** their implementation from objects of other classes—**data hiding**

classes we have already used

- Console class: provides basic support for applications that read / write characters from / to the console of the OS
  - WriteLine( ), Write( ), Read(), ReadLine(), …

- Random class: a pseudo-random number generator – produces a sequence of numbers that meet certain statistical requirements for randomness
  - Next(1, 7), NextDouble(), …

classes we have already used: string

- String class: sequential collection of characters – System.Char objects (UTF-16 code unit)
- string aFriend = "James"; ⟷ string aFriend = new string("James");
- Methods
  - Equals
  - CompareTo
  - ToUpper
  - Split
  - …

other classes

- All the functionalities provided by the C# library are organised as classes:
  - File
  - FileStream
  - Socket
  - …

More problems we can solve with OOP

- Classes that allow modelling Windows, Menus, Widgets of a **Graphical User Interface**
- Classes that allow interaction with **Entities of a Database**
- Classes that model and allow the **emulation/simulation** of physical **systems** (Networks, Biology, etc.)
- …

Any abstract **model** of the **real world** required to **solve** a given **problem**