OOP Terminology

Module 2

Luis Beltrán
beltran_prieto@utb.cz

Object-oriented Programming
Winter 2023
Tomas Bata University in Zlín – Faculty of Applied Informatics

Motivation for OOP

You should strive to write software that is easy for you and other programmers to understand.

Organization is important

The way your program code is organized makes a huge difference in how easy it is to debug and maintain.

Faster to fix errors

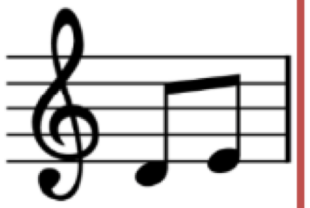
Faster to add new features

Object Oriented Programming

Object Oriented Programming (OOP) is a design philosophy invented to handle increasingly complex programs where we create objects to model things in the **problem domain** (the problem we are trying to solve).

Music app might have:

- Album
- Song
- Playlist
- …

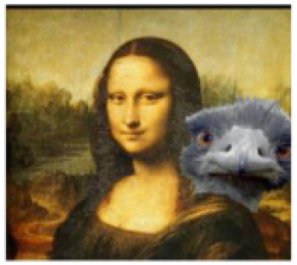


Nike app might have:

- Runner
- Workout
- Route
- …

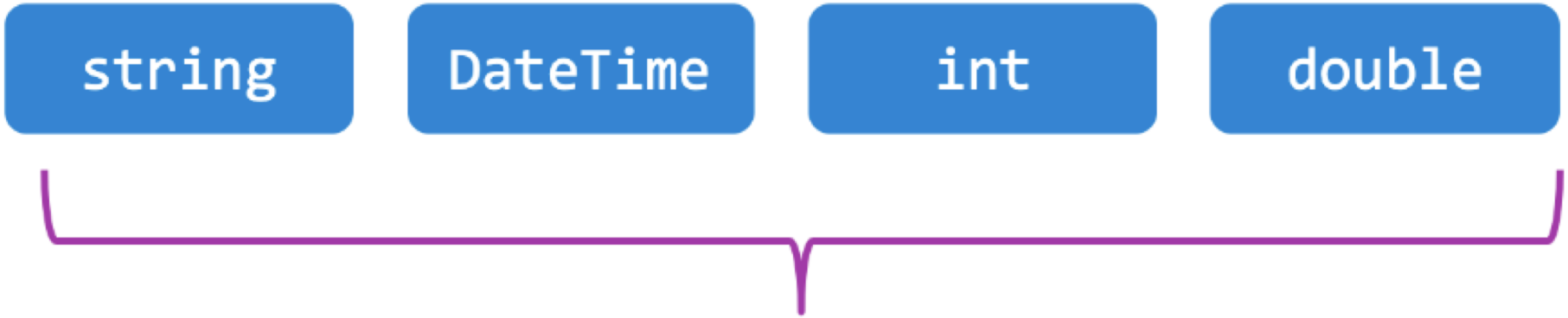
Drawing app might have:

- Circle
- Square
- Eraser
- …



Defining new types

The key idea of OOP is that the programmer can create **new types** to better model the world.



A type tells you the kind of object you are working with, these are examples of built-in types

Class

A class is a **software model** that defines a **new type** representing some concept or real-world element in your program.

Just as this model represents an airplane
and has many of the same elements

Class

A class is a blueprint from which you can create objects

A class defines the characteristics of an object

```
class House
{
    ...
}
```

Class definition with class keyword

An object is an instance of a class

myHouse

yourHouse

Creating Classes and Members

- A class has a name and a body, the body is delimited with **{** and **}**.

- Use the **class** keyword, then a name (by convention, capitalize each word)

```
public class BankAccount
{
    // Methods, fields, properties, and events will go here
}
```

- Specify an access modifier:
  - public
  - internal
  - private

- Add methods, fields, properties, and events

Classes and Files

It is recommended to put each class in its own file (but it is not mandatory)

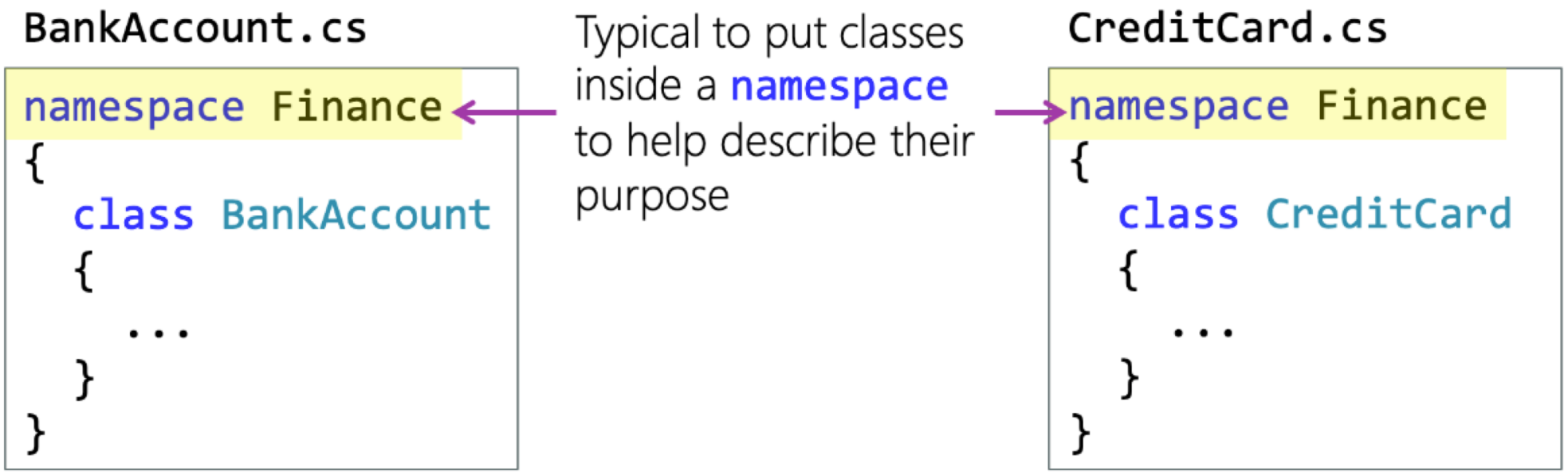
File name should be the same as the class name with .cs extension

BankAccount.cs

```
class BankAccount
{
    ...
}
```

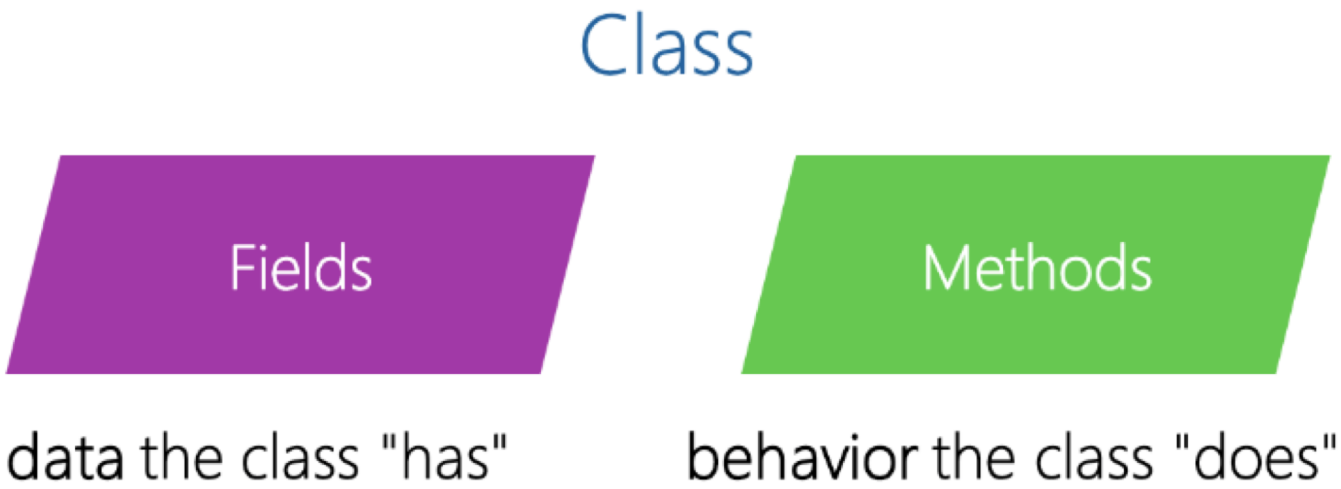
Classes and namespaces

- A namespace groups **related classes** together.
- It is useful for organization.



What is in a class?

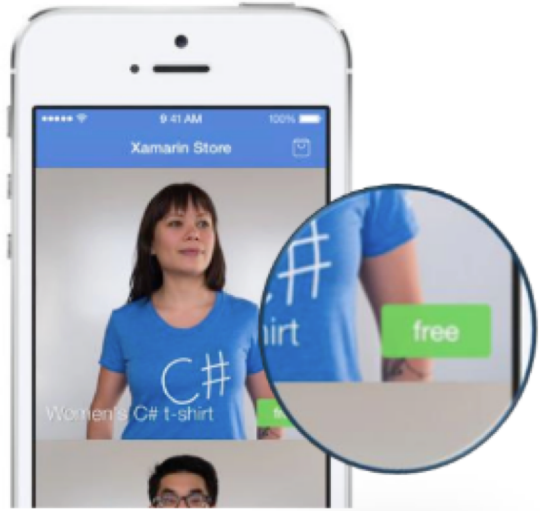
Classes contain **data** and **behavior** bundled together.



Fields

A field is a variable owned by the class that holds data.

For a dog, the fields might include *name*, *age*, *weight,* and *breed*

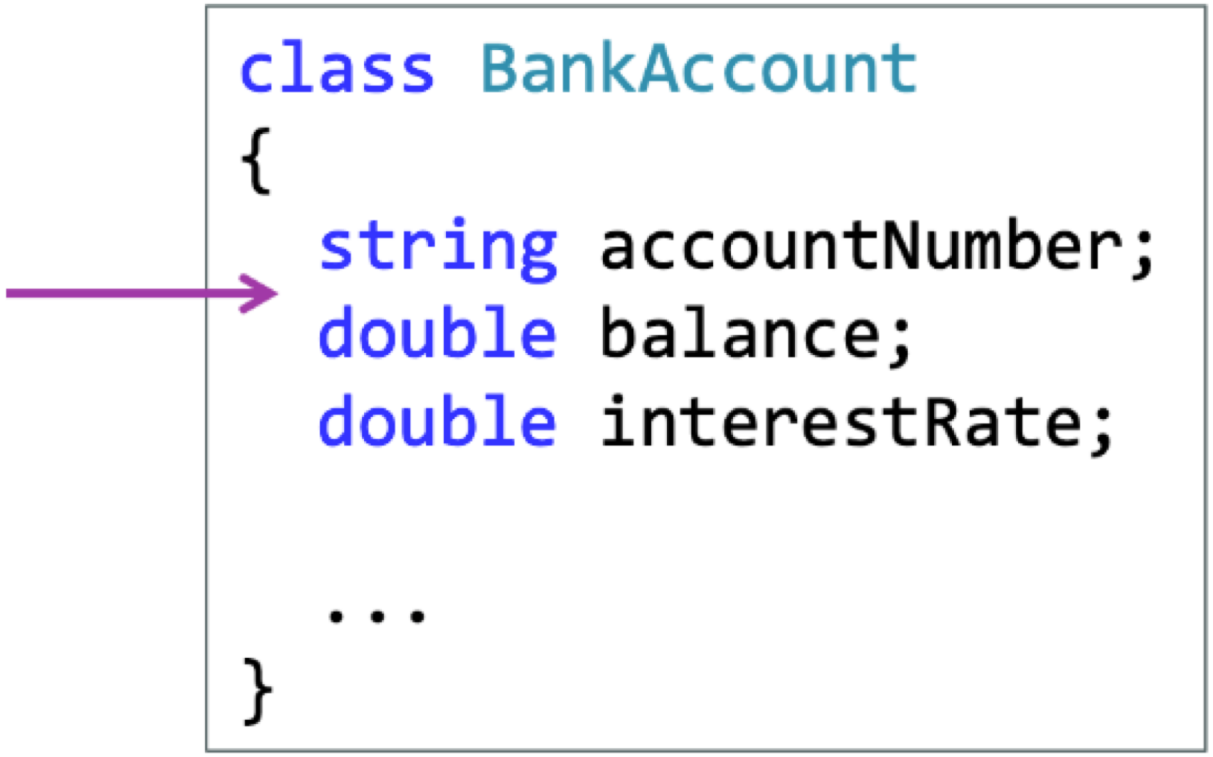


For a button, the fields might include *width*, *height*, *position*, and *text*.

Declaring fields

Variables declared inside the class define the fields.

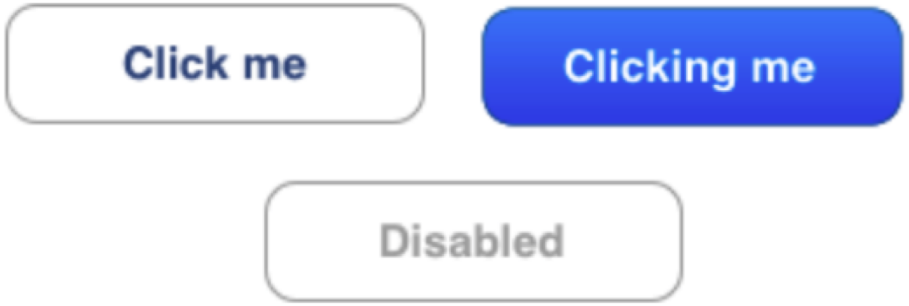
Three independent fields are included as part of the **BankAccount** class



Methods

Methods are **code blocks**, containing C# statements, that provide logic to perform work related to the class.

For a dog, the methods might include *bark*, *walk*, *eat,* and *sniff.*



For a button, the methods might include *show*, *hide*, *click,* and *resize*

Defining methods

A method is declared inside a class with a **name** that indicates what behavior or operation the method performs.

method body is contained within open { and close } curly braces

```
public class BankAccount
{
  public double Balance;
  public double InterestRate;

  public void AddInterest()
  {
    double interest = Balance * InterestRate;
    Balance += interest;
  }
  ...
}
```

The method `AddInterest` calculates the interest based on our data fields and adjusts the balance

void indicates
it does not return
any result value

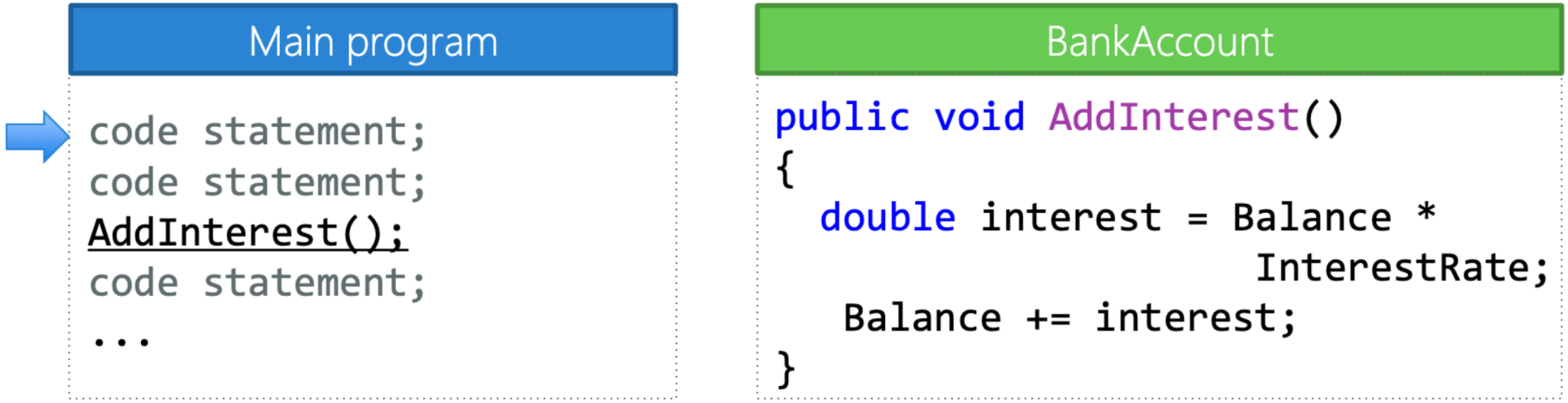
public indicates
it can be used from
outside the class

```
public class BankAccount
{
  public double Balance;
  public double InterestRate;

  public void AddInterest()
  {
    double interest = Balance * InterestRate;
    Balance += interest;
  }
  ...
}
```

open (and close)
parentheses after
the name tells the
compiler that this is
a method declared
for this class

What happens when you call a method?

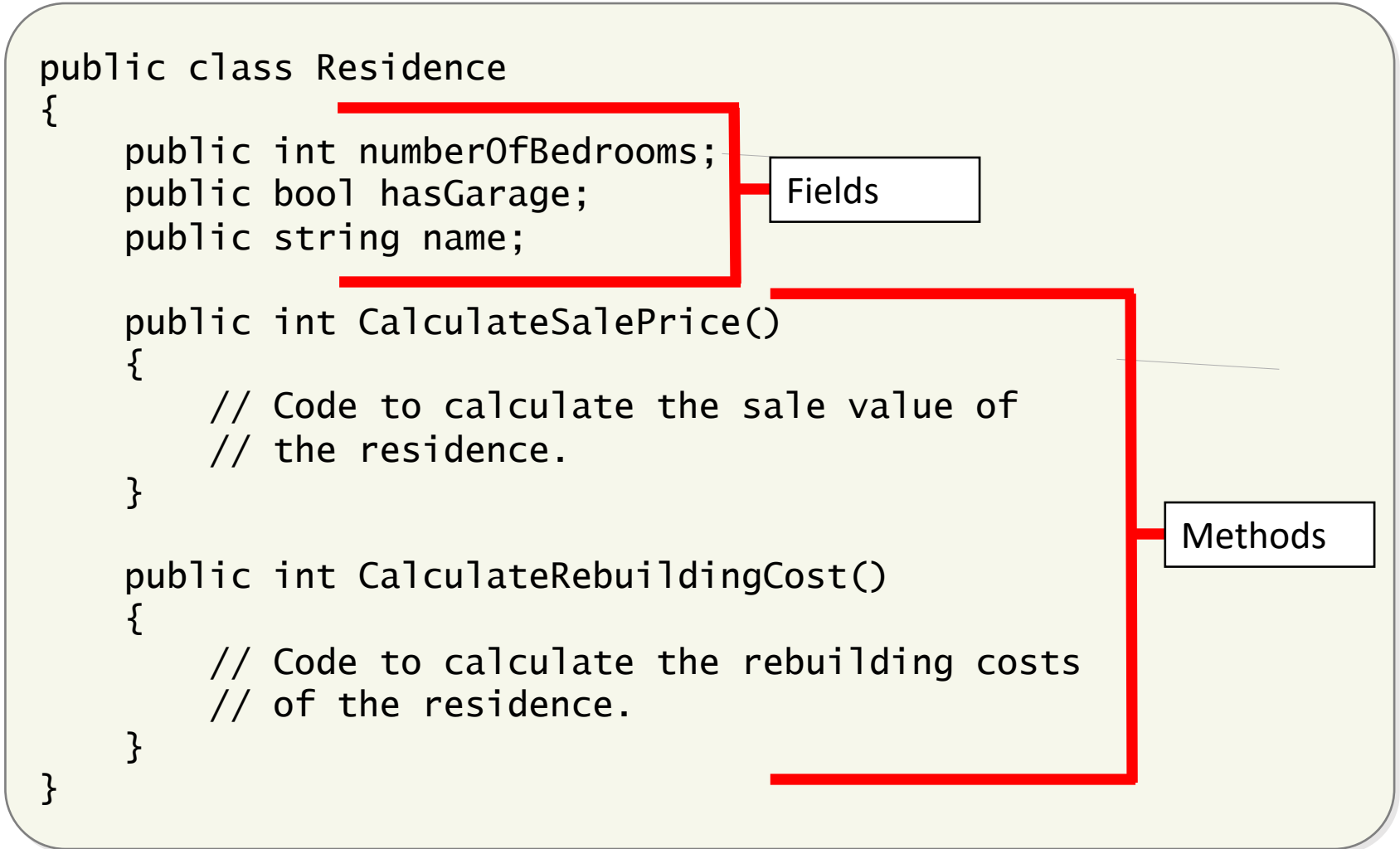
- Calling a method on an object causes your program to **execute the code contained in that method**.
- When the method finishes, your program *continues executing* the code that follows the call to the method.



Adding Members to Classes

Members define the data and behavior of the class

```
public class Residence
{
    public int numberOfBedrooms;
    public bool hasGarage;
    public string name;

    public int CalculateSalePrice()
    {
        // Code to calculate the sale value of
        // the residence.
    }

    public int CalculateRebuildingCost()
    {
        // Code to calculate the rebuilding costs
        // of the residence.
    }
}
```



Class granularity

Classes should be counted on to do one, well-understood thing.

Demo #1 – Creating a class

How to identify classes?

- Classes are *models of things in the real world*.
- We can often identify potential classes by examining what we need an application to do
- ***What are the potential classes we might need for this mapping application to support route planning? →***

- Here are some possibilities. Remember we want to accurately reflect the real world “things” we are working with.

- Map
- Current Position
- Calculated Route
- Street
- Turn

Mapping application with route planner

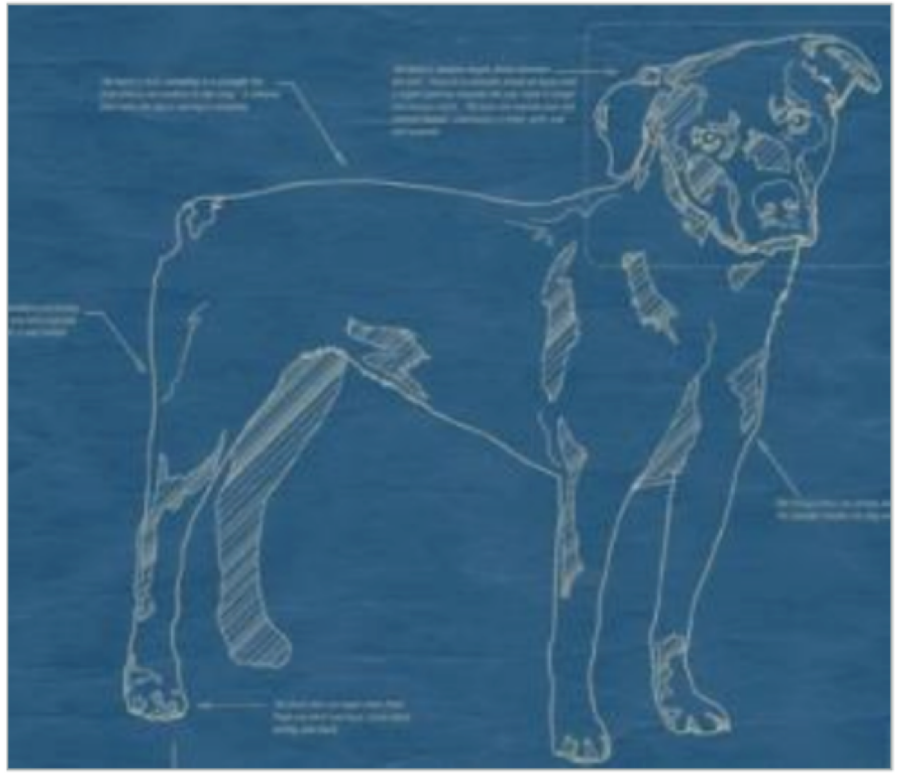
Name at least 3 potential classes that might be defined for a chat application that allows users to converse using text messages

Name at least 3 pieces of data (fields) we might want in a chat message

Name at least 3 methods (behavior) we might need in a chat message.

Objects

A **class** defines a **template,** and **objects** are **instances** of that template.



Class Dog

Instances of class Dog

Instantiating Classes (creating objects)

Objects are initially unassigned

Before you can use a class, you must create an instance of that class

You can create a new instance of a class by using the **new** operator

The **new** operator does two things:

- Causes the CLR to allocate memory for the object
- Invokes a constructor to initialize the object

Instantiating Classes (creating objects)

- To instantiate a class, use the **new** keyword

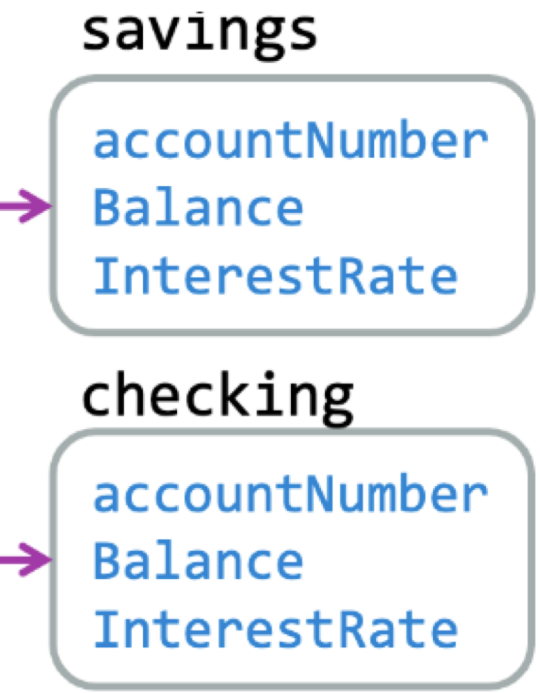
```
Residence r = new Residence();
```

- To infer the type of the new object, use the **var** keyword

```
var r = new Residence();
```

When you create instances (objects) of a class, each object gets **its own copy** of the fields.

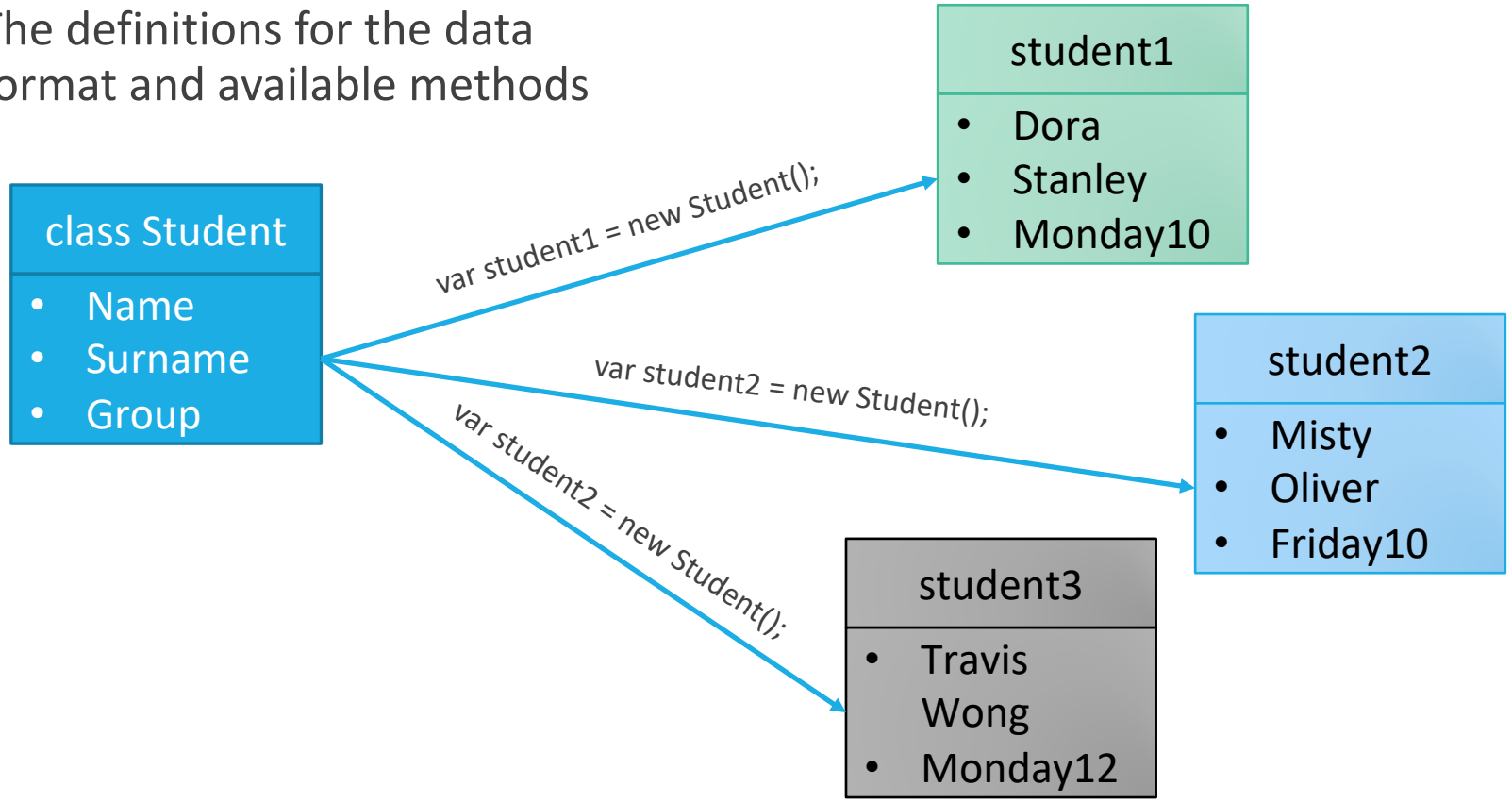
```
public class Program
{
  public static void Main()
  {
    BankAccount savings  = new BankAccount();
    BankAccount checking = new BankAccount();
    ...
  }
}
```



Classes and Objects

Instances of classes are objects in a computer memory

The definitions for the data format and available methods



Accessing Class Members

To access members on the instance, use the dot operator

```
InstanceName.MemberName
```

```
Residence r = new Residence();

r.numberOfBedrooms = 3;
r.hasGarage = true;
r.name = "U12 Halls of Residence"
r.CalculateSalePrice();
```

Accessing Class Members

```
public class Program
{
  public static void Main()
  {
    BankAccount savings  = new BankAccount();
    BankAccount checking = new BankAccount();

    savings.Balance  = 100.00;
    checking.Balance = 500.00;

    double netWorth = savings.Balance + checking.Balance;
  }
}
```

savings

accountNumber	
Balance	100.00
InterestRate	

checking

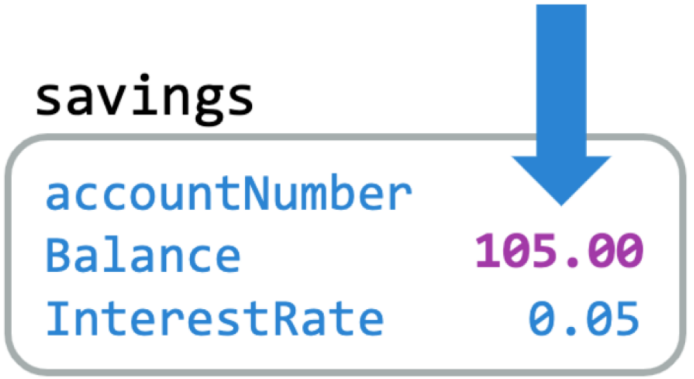
accountNumber	
Balance	500.00
InterestRate	

With the dot operator, you can access (read or change) the members of an object.

Calling methods

Use the **dot operator** and **parentheses** to invoke a **method** on an **object**.

```
public static void Main()
{
    BankAccount savings = new BankAccount();

    savings.Balance = 100.00;
    savings.InterestRate = 0.05;

    savings.AddInterest();
}
```



Calling the method **AddInterest()** will change the Balance to 105.00

Demo #2 – Classes and Objects

Method parameters

- Sometimes, methods need additional data in order to perform the logic required – this could be supplied by **setting fields** in the class.

- If the data is only used by the method, a better approach is to **pass the data inside** the method call – this is called a **parameter**.

depositing money would require some $$$ amount to add to our bank account

Passing method parameters

Method parameters are additional pieces of information passed from the caller into the method (also known as *arguments*).

```
public class Program
{
  public static void Main()
  {
    BankAccount savings = new BankAccount();

    savings.Balance = 100.00;

    savings.Deposit(50.00);
  }
}
```

```
public class BankAccount
{
  public double Balance;

  public void Deposit(double amount)
  {
    Balance += amount;
  }
  ...
}
```

Parameters act as *local variables* within the method

We use amount directly as if it were declared in our method

Method parameter validation

Method parameters must define the type of value they expect – the compiler will enforce this and not allow unexpected values to be passed in.

```
public class BankAccount
{
  public double Balance;

  public void Deposit(double amount)
  {
    Balance += amount;
  }
  ...
}
```

This method expects a double numeric value

```
BankAccount account = ...;

account.Deposit(500.0);

account.Deposit(true);

account.Deposit(500);

account.Deposit("500");
```




Passing multiple parameters

Methods can take as many parameters as they need to perform their work.

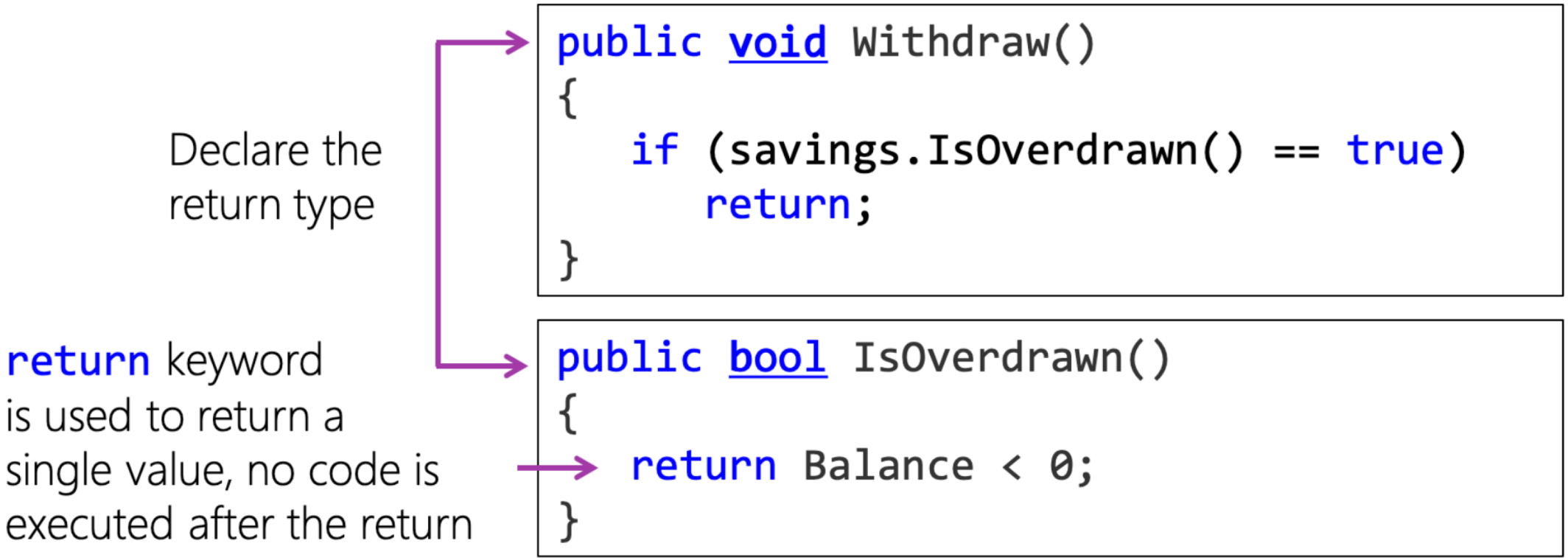
```
public class BankAccount
{
  private string accountNumber;
  public  double Balance;
  public  double InterestRate;

  public void Initialize(string account, double balance, double rate)
  {
    accountNumber = account;
    Balance       = balance;
    InterestRate  = rate;
  }
  ...
}
```

parameter #1 parameter #2 parameter #3

Each parameter can have a different type

How methods return values

- Methods can compute and return a **single value** to the caller.
- Each method must declare the **type** it returns (or **void** to indicate no value)



Method overloading

- Sometimes two or more methods perform the same logic but require different parameters.

- C# allows you to create more than one method with the same name but *different parameters*; this is called **method overloading:**
  - **Different parameter types**
  - **Different number of parameters**

```
double Add(double x, double y);

int Add(int x, int y);

double Add(double x);
```

three variations of an **Add** method on a calculator, each taking different parameters

Return values

Return values and visibility are not considered in method overloading

```
public class Calculator
{
    public double Add(double x, double y)
    {
       ...
    }

    private int Add(double x, double y)
    {
        ...
    }
}
```

```
error CS111: A member
'Calculator.Add(double,double)' is already defined.
Rename this member of use different parameter types
```

When we create an object (class instance), we need to make sure it’s ready to be accessed

Demo #3 – Methods and parameters

Class Constructor

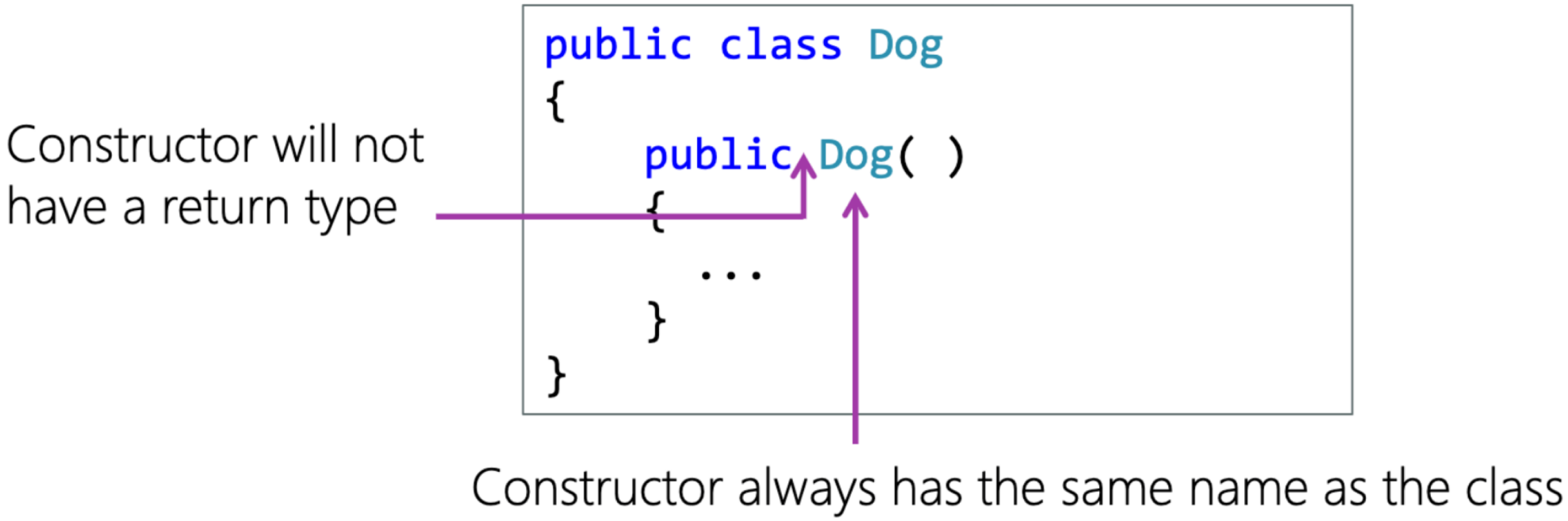
- A **class constructor** is a special piece of code that is called automatically by C# when the object is created.
- It is responsible for **initializing** the object.

```
Dog lassie = new Dog();

// Use lassie here
```

C# does two things when we call **new** on a class – it **allocates** memory to hold the object, and then **builds** the object by calling the constructor method

Defining a constructor

Constructors look like methods, but they have two unique characteristics in how they are defined



Why would you use a constructor?

Constructors let you assign default values and ensure the object is correctly set up before it is used.

We can assign reasonable values to any fields and properties, so clients do not get unexpected results when they use the object

```
public class Dog
{
    public string Breed { get; set; }

    public Dog()
    {
        Breed = "Unknown";
    }
}
```

Default constructors

A default constructor is any constructor that takes no parameters

```
public class Dog
{
    public int Age { get; set; }
    public string Breed { get; set; }
    public bool Pure { get; set; }

    public Dog()
    {
        Age = 0;
        Breed = "Unknown";
        Pure = false;
    }
}
```

Default constructors

If you do not declare any constructors, the compiler will give you an invisible, default constructor that does nothing

fields and properties are initialized to default values

```
public class Dog
{
    public int Age { get; set; }
    public string Breed { get; set; }
    public bool Pure { get; set; }
    // Age = 0;
    // Breed = null;
    // Pure = false;
}
```


`null` is a special value assigned to **`string`** and other non-numeric types that indicates it has *no value*, this is different from an empty string (**`""`**) which is a string with no data

Assigning values to objects

Of course, you can assign properties after the object is constructed

```
public class Dog
{
    public string Breed { get; set; }

    public Dog() // Default constructor
    {
        Breed = "Unknown";
    }
}
```

```
Dog lassie = new Dog ();
lassie.Breed = "Collie";
...
```

To use the dog, we assign the **Breed** property *after* we create the object.

Passing parameters to constructors

We can pass parameters to a constructor so we can set fields and properties in the constructor itself.

```
public class Dog
{
    public string Breed { get; set; }

    public Dog(string breed)
    {
        Breed = breed;
    }
}
```

Define parameters to be passed into the constructor

```
Dog dog = new Dog();
dog.Breed = "Collie";


Dog dog = new Dog("Collie");
```

If you declare a constructor with parameters, you must supply the parameters when you create the object.

```
public class Dog
{
    public Dog(string breed) { ... }
    ...
}
```

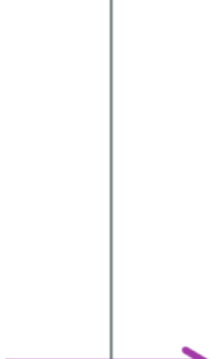
```
Dog dog = new Dog();  // not valid
```

This can be very beneficial if the class *must* have the supplied information to make the object valid, but what if it's not required?

Multiple constructors – overloading

You can overload the constructor (create multiple constructors) to support different requirements.

Declare a default constructor to allow for simpler creation scenarios



```
public class Dog
{
    public string Breed
        { get; set; }

    public Dog(string breed)
    {
        Breed = breed;
    }

    public Dog()
    {
        Breed = "Unknown";
    }
}
```

Constructors

A constructor is a special method that the runtime invokes implicitly

```
public class Residence
{
    public Residence(int numberOfBedrooms)
    {
        this.numberOfBedrooms = numberOfBedrooms.
    }

    public Residence(int numberOfBedrooms, string name)
    {
    }

    public Residence(int numberOfBedrooms, string name, bool hasGarage)
    {
    }
}
```

The **this** keyword refers to the current instance of the class.

Using the **this** keyword we can differentiate between fields and local variables or method parameters.

Three constructors

If you do not initialize a field in a class, it is assigned its default value

Instantiating Classes (creating objects) - again

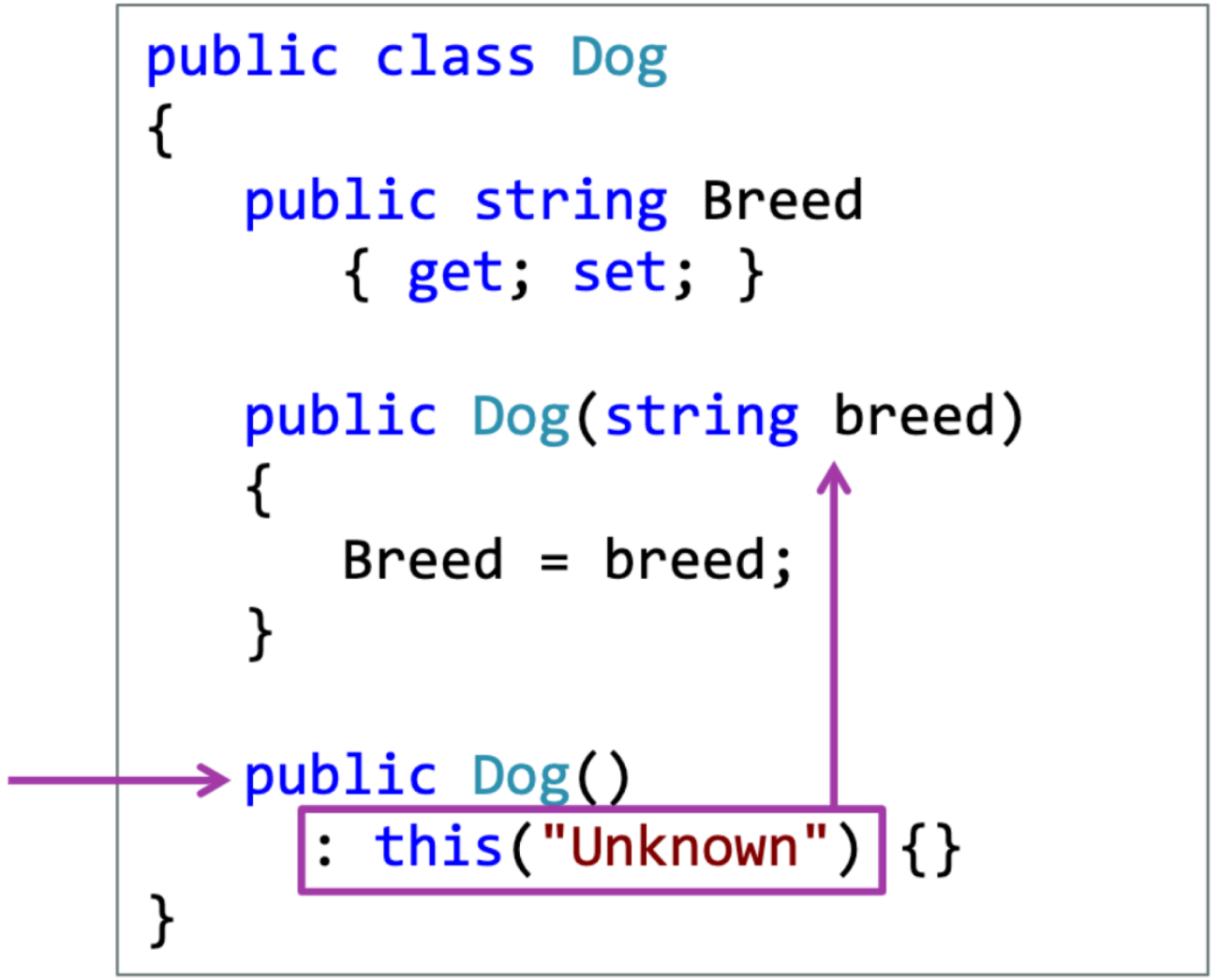
```
// Create a residence with 2 bedrooms.
Residence myFlat = new Residence(2);

// Create a residence with 3 bedrooms and a name
Residence myHouse = new Residence(3, “Bata Residence”);

// Create a residence with 2 bedrooms, a name, and a garage
Residence myBungalow =
   new Residence(2, “Rest House”, true);
```

Demo #4 – Constructors

Chaining constructors

One constructor can call another one to share the initialization code.

Here, the default constructor calls the first constructor with a parameter



Constructors – Summary

- Constructors are a type of method:
  - Share the name of the class
  - Called when you instantiate a class
- A default constructor accepts no arguments

```
public class DrinksMachine
{
   public void DrinksMachine()
   {
      // This is a default constructor.
   }
}
```

- Classes can include multiple constructors
- Use constructors to initialize member variables
- Constructors don’t return values

Visibility

- Some things in a class are **public** – can and should be seen by other classes.

- Other things in a class are **private** and should only be visible inside the class.

Encapsulation

Definition

- An essential object-oriented principle
- Hides internal data and algorithms
- Provides a well-defined public operation

Benefits

- Makes external code simpler and more consistent
- Enables you to change implementation details later

Access modifiers

Access modifiers limit what fields and methods contained in the object can be used from the outside.

Anyone can use this field or method

Only the class can use this field or method
This is the default!

Private and Public Members

private Least permissive access level

```
class Sales
{
    private double monthlyProfit;
    private void SetMonthlyProfit(double monthlyProfit)
    {
        this.monthlyProfit = monthlyProfit;
    }
}
```

Only accessible from within the Sales class

public Most permissive access level

```
class Sales
{
    private double monthlyProfit;
    public void SetMonthlyProfit(double monthlyProfit)
    {
        this.monthlyProfit = monthlyProfit;
    }
}
```

Accessible to any other type

Well-designed classes (public vs private)

You should carefully decide what visibility give for each field and method:

- Do other classes need access to this field, or is this field internal information that only needs to be used inside the class itself to make decisions?

- Is this method useful by other classes, or is it a method that helps other methods inside the class do their job?

Remember that the **default visibility** for classes, fields and methods is always **private**.

Recommendation for fields declaration

private fields are only accessible from inside the class

public fields are values that other classes can access from outside the class, e.g. from **Main**

```
public class BankAccount
{
    private string accountNumber;

    public double Balance;
    public double InterestRate;

    ...
}
```

It is typical to use different capitalization for **public** vs **private**.

Demo #5 – Access Modifiers

Other access modifiers

- `protected`
  - Can be accessed only by code in the same class or struct, or in a class that is derived from that class.
  - Most important access modifier in **inheritance**.

- `internal`
  - can be accessed by any code in the same assembly, but not from another assembly.

The problem with fields

When you make a field **public**, code outside your class can read and alter the value of the field

```
public class BankAccount
{
  public double Balance;
  ...
}
```

```
BankAccount account = ...;

account.Balance = 100.0;
...
account.Balance -= 200.0;
```

Here we are dropping our balance below zero, should that be allowed? How can my class stop this from happening?

The solution… methods?

We can make fields private and then use methods to read and change the values – this allows our class to ensure the field is always valid

```
public class BankAccount
{
  private double balance;

  public double GetBalance() { return balance; }
  public void SetBalance(double value) {
    if (value >= 0)
       balance = value;
  }
  ...
}
```

```
BankAccount account = ...;

account.SetBalance(100.0);
...
account.SetBalance(
   account.GetBalance()
         - 200.0);
```

This solves our problem, but is more complex .. and ugly

What we really want is…

Ideally, we could create something that looks *like a field*, but provides *methods* to **get and set** the stored value so we *can control access* to the data.

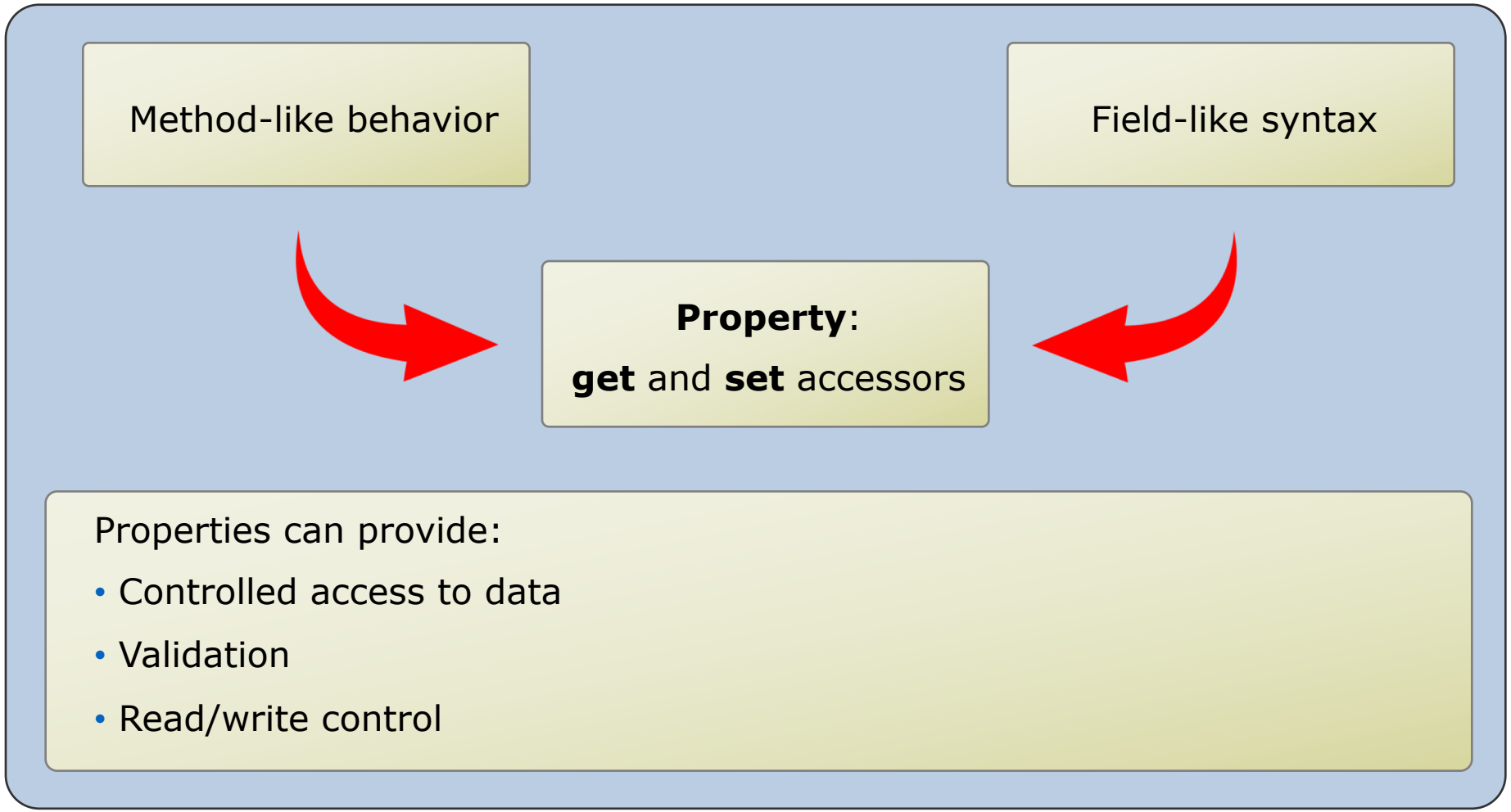
```
BankAccount account = ...;

account.Balance = 100.0;
...
account.Balance -= 200.0;
```

This syntax is very elegant and natural

```
BankAccount account = ...;

account.SetBalance(100.0);
...
account.SetBalance(
  account.GetBalance() - 200.0);
```

… but this provides the *behavior* we want

Properties


Method-like behavior
Field-like syntax
Property:
get and **set** accessors
Properties can provide:
• Controlled access to data
• Validation
• Read/write control

Property

A C# **property** consists of a pair of keywords which provide access to a data value exposed by the class.

```
public class BankAccount
{
   private double balance;
   public double Balance
   {
      get { return balance; }
      set {
         if (value >= 0)
            balance = value;
      }
   }
   ...
}
```

typically has a private field to store the value

Property body is enclosed in curly braces

```
public class BankAccount
{
   private double balance;
   public double Balance
   {
      get { return balance; }
      set {
         if (value >= 0)
            balance = value;
      }
   }
   ...
}
```

Property is **public** and Pascal-cased

get method used to retrieve the value

```
public class BankAccount
{
   private double balance;
   public double Balance
   {
      get { return balance; }
      set {
          if (value >= 0)
              balance = value;
      }
   }
   ...
}
```

```
public class BankAccount
{
   private double balance;
   public double Balance
   {
      get { return balance; }
      set {
         if (value >= 0)
            balance = value;
      }
   }
   ...
}
```

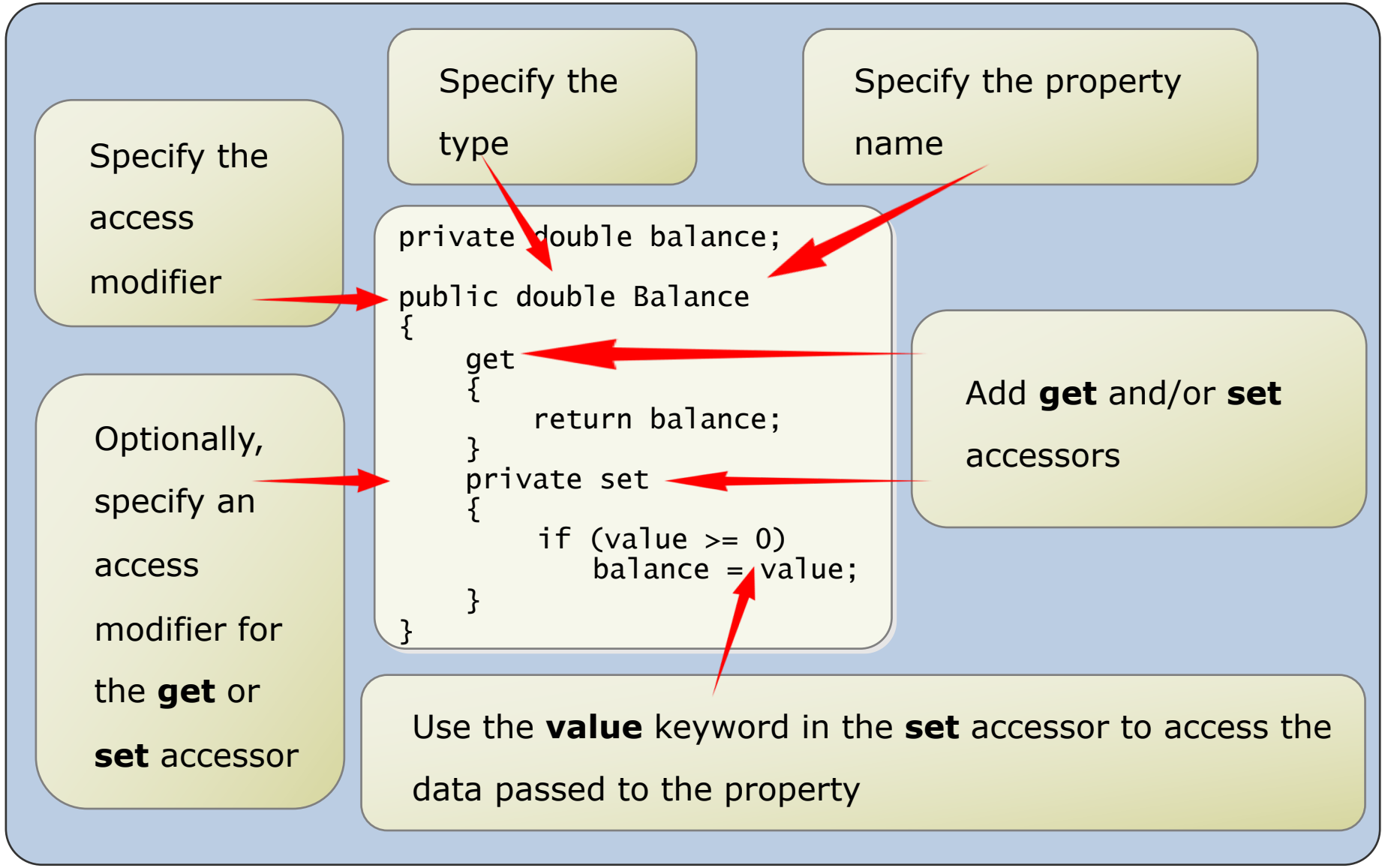
set method is used to change the value and provide any necessary validation

```
public class BankAccount
{
   private double balance;
   public double Balance
   {
      get { return balance; }
      set {
         if (value >= 0)
            balance = value;
      }
   }
   ...
}
```

```
account.Balance = 200;
```

200 is passed in as **value**

Defining a Property



Using a property

- Code outside the class will use the property to access the data.
- It looks like a field, but acts like a method

```
BankAccount account = ...;

account.Balance = 100.0;
Console.WriteLine(account.Balance);
...
account.Balance -= 200.0;
Console.WriteLine(account.Balance);
```

Here we attempt to change our balance to a negative value, but the property logic stops that and the value remains unchanged

```
100.0
100.0
```

Properties are really methods that are being used:

- The **getter** is called whenever the code **reads the value.**
- The **setter** is called whenever code attempts to **alter the value**.

```
BankAccount account = ...;
account.Balance = 100.0;
Console.WriteLine(account.Balance);
```

We write this nice, natural code which is easy to understand

```
BankAccount account = ...;
account.set_Balance(100.0);
Console.WriteLine(account.get_Balance());
```

... and C# turns that into calls to the defined property methods

Demo #6 - Properties

Simplifying properties

Properties are often just simple wrappers around a private field - no additional logic is provided beyond getting and setting the field's value.

```
public class BankAccount
{
    private string id;
    public string Id
    {
        get { return id; }
        set { id = value; }
    }
}
```

Auto Properties

When you don’t need any additional logic for the property, you can use an **auto property**.

```
public class BankAccount
{
    public string Id
    {
        get;
        set;
    }
}
```

You cannot supply method bodies for the getter or setter with auto properties

Automatic Properties

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

The automatic property shown above is converted by the compiler to code similar to:

```
private string _name;

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

- Useful when you do not need to add custom logic to the property accessors
- Must specify both **get** and **set** accessors
- Important for forward compatibility
- No difference between automatic properties and normal properties to consuming applications

Instantiating an Object by using Properties

```
Employee john = new Employee { Name = "John" };

Employee louisa = new Employee() { Department = "Technical" };

Employee mike = new Employee
                    {
                        Name = "Mike",
                        Department = "Technical"
                    };
```

- An object initializer avoids problems with defining several constructors
- A default constructor should instantiate properties to default values
- A constructor is called the object initializer
- A constructor is always run first, and properties are set after, so properties take precedence
- An object initializer can use no brackets to call the default constructor, brackets with no parameters to explicitly call the default constructor, or brackets with parameters to call a nondefault constructor

Read-only properties

Normally, we define both a getter and a setter on a property, but you can define a **read-only property** by leaving off the setter.

Here, we only define a getter for the **`Balance`** property – outside code can only *read* the value

```
public class BankAccount
{
  private double balance;
  public double Balance
  {
    get { return balance; }
  }
  ...
}
```

Calculated values

Properties are often used to provide access to **calculated values** which do not have an associated field, but are calculated when the getter is called.

```
public class BankAccount
{
  public double ExpectedInterest
  {
     get { return balance * interestRate; }
  }
  ...
}
```

We calculate the expected interest each time something reads the property

Demo #7 – Advanced Properties

Object Oriented Programming Features

- Object is an instance of class and consists of:
  - Data strucures, also known as attributes, data, **fields (C#, Java)**, member variables (C++) and
  - functions, which are logically related to a particular data structure, also know as procedures, **methods (C#, Java)**, member variables (C++).

- Encapsulation
  - Binds together data and methods that manipulates data
  - Hides private (internal) data and methods and provides public methods intended for use by code outside the object.

- Abstraction
  - We can hide internal implementation and outside the object we use only public interface, it is easy to work with object and w can replace internal implementations later.

- Composition
  - One class contains instances of other classes.

Optional Parameters

C# supports **optional parameters** where the method **declares a default value** which is used if the parameter is not passed.

```
double CalculateTax( double amount, double rate = 15.0 )
{
    return amount * rate / 100;
}
```

optional parameter(s) must be at the end of the parameter list

```
double tax = CalculateTax(2000);          // Rate = 15.0
```

```
double tax = CalculateTax(2000, 20);      // Rate = 20.0
```

What if more than one value should be returned in a method?

Methods can return **zero** or **one** value… whaf if more are needed?

```
public ??? CalculateMortgage(double loanAmount)
{
    double monthlyPayment = ...;
    double totalInterest = ...;

    return ???
}
```

How can we return *both* the monthly payment and total interest?

Solution A

One solution would be to create a new class to hold our return data.

```
public class MortgageInfo
{
    public double Payment { get; set; }
    public double TotalInterest { get; set; }
    ...
}
```

Add properties to hold the data we want to return

```
public MortgageInfo  CalculateMortgage(double loanAmount)
{
    ...
    MortgageInfo payment = new MortgageInfo {
        Payment = ...,
        TotalInterest = ...
    };
    return payment;
}
```

Return single object with all the data

Solution B

Methods can return more than one value by using **out** parameters.

out is a keyword indicating that the called method assigns a value to the parameter which updates the variable that the caller passed in

```
public double CalculateMortgage(
                        double loanAmount, out double totalInterest)
{
    double monthlyPayment = ...;
    totalInterest = 10240;

    return monthlyPayment;
}
```

Out parameters *must* have a value assigned in the method. The caller's variables will have the assigned values.

Getting multiple values out of a method

Must add the **out** keyword to the parameter when calling the method as well – this ensures you know the **value will be changed**.

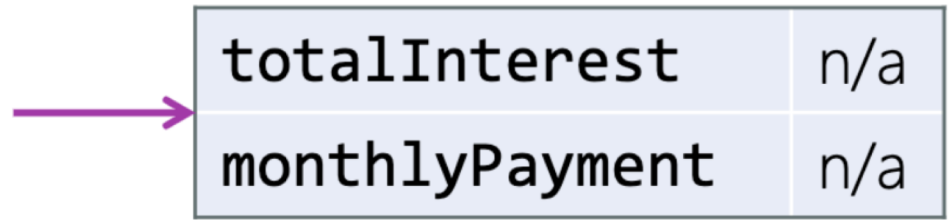
```
public double CalculateMortgage(
                double loanAmount, out double totalInterest) ...
```

Variable does not need to have an initial value before passing to the method

```
double totalInterest;
double monthlyPayment = bank.CalculateMortgage(10000,out totalInterest)
```

```
double totalInterest;
double monthlyPayment = bank.CalculateMortgage(10000,out totalInterest)
```

fields start out with no value (uninitialized)




```
public double CalculateMortgage(
                double loanAmount, out double totalInterest)
{
   totalInterest = loanAmount * 0.20;
   double monthlyPayment = (loanAmount + totalInterest) / 36;
   return monthlyPayment;
}
```

```
double totalInterest;
double monthlyPayment = bank.CalculateMortgage(10000,out totalInterest)
```

totalInterest	2000
monthlyPayment	333.333333

```
public double CalculateMortgage(
                double loanAmount, out double totalInterest)
{
   totalInterest = loanAmount * 0.20;
   double monthlyPayment = (loanAmount + totalInterest) / 36;
   return monthlyPayment;
}
```

Demo 8 – Parameters: optional and out

OOP Terminology

Module 2

Thank you for your attention!

Luis Beltrán
beltran_prieto@utb.cz

Object-oriented Programming
Winter 2023
Tomas Bata University in Zlín – Faculty of Applied Informatics