Method Signatures, Overloading, and Properties

Principles of Computer Programming I

Spring/Fall 20XX

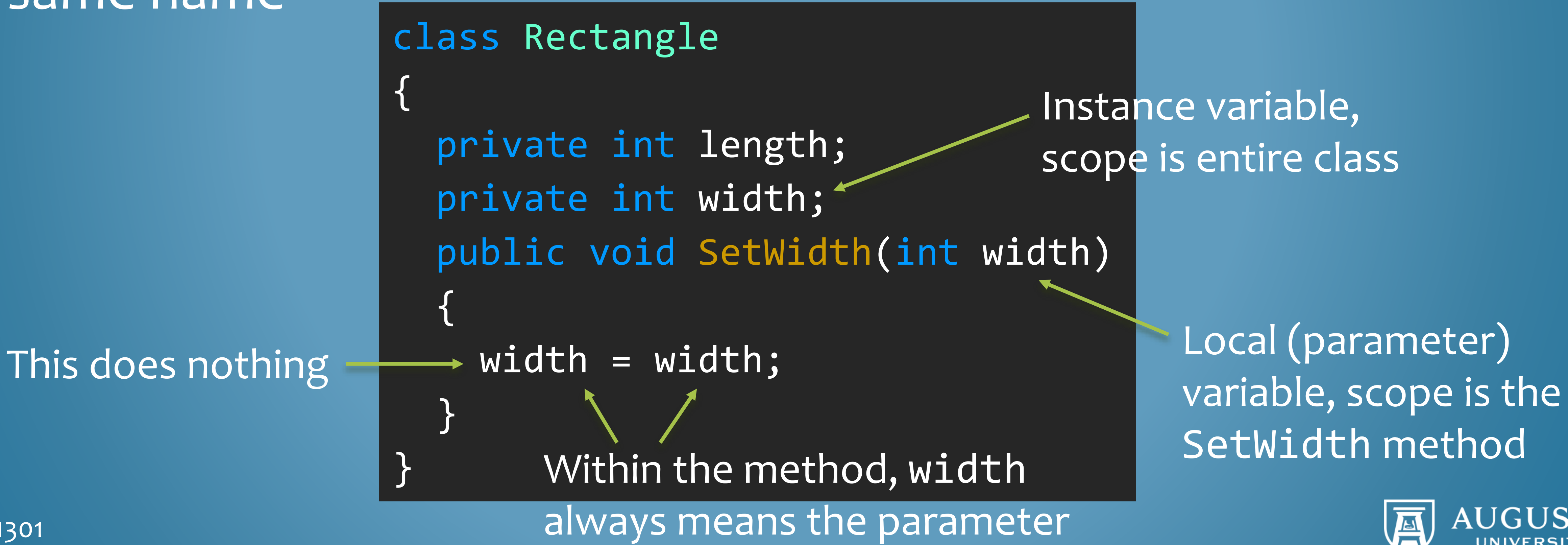
AUGUSTA UNIVERSITY

Outline

- Name Uniqueness
- Signatures and Overloading
  - Type Conversions in Arguments
- Properties

Exceptions to Unique Name Rule

- Scope: Variables in different scopes can have the same name
- Shadowing: Local variable will "hide" instance variable with same name



```
class Rectangle
{
  private int length;
  private int width;
  public void SetWidth(int width)
  {
    width = width;
  }
}
```

Instance variable, scope is entire class

Local (parameter) variable, scope is the `SetWidth` method

This does nothing

Within the method, `width` always means the parameter

Exceptions to Unique Name Rule

- Namespaces: Classes can have the same name if they are in different namespaces

```
namespace MyProject
{
  class Rectangle
  {
    …
  }
}
```

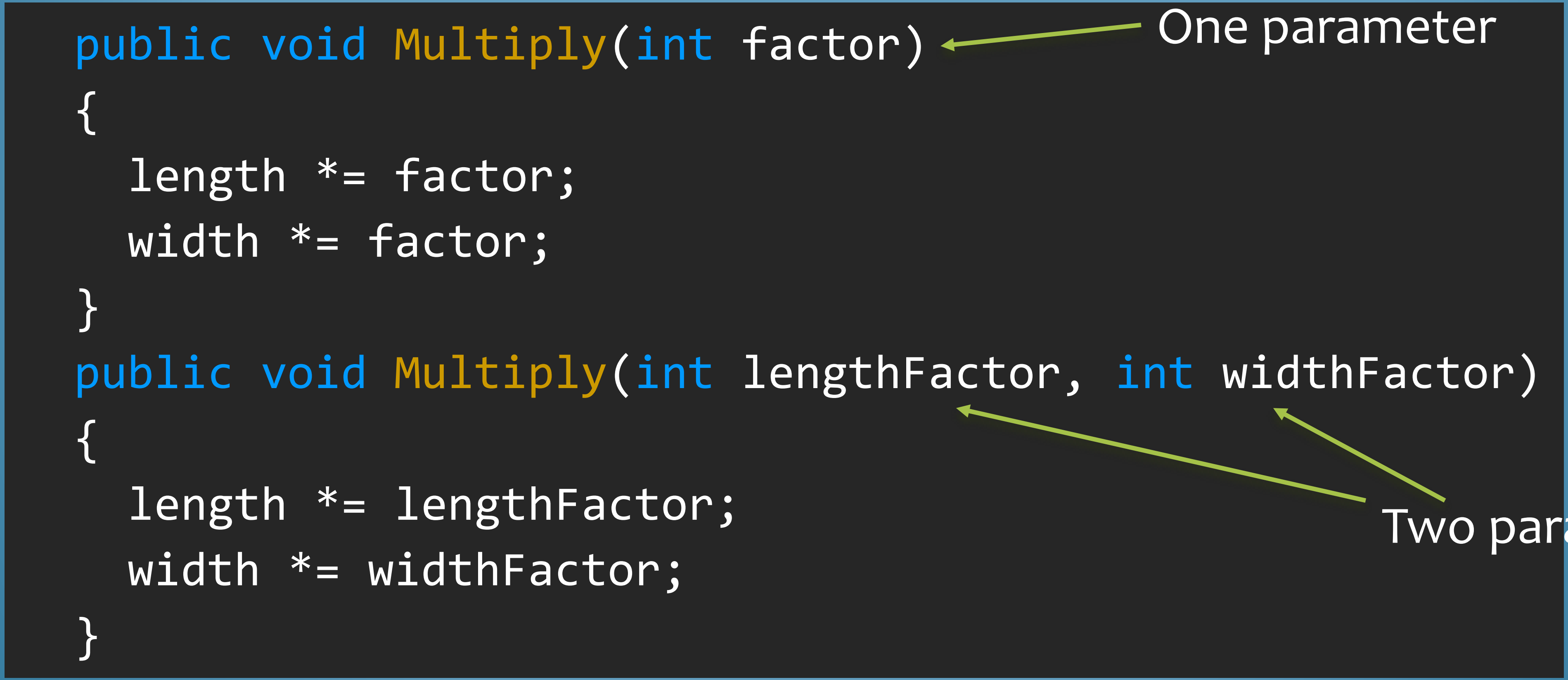
```
namespace ShapesLibrary
{
  class Rectangle
  {
    …
  }
}
```

Can be used like this:

```
MyProject.Rectangle rect1;
ShapesLibrary.Rectangle rect2;
```

Exceptions to Unique Name Rule

- Overloading: Methods can have the same name if they have different parameters



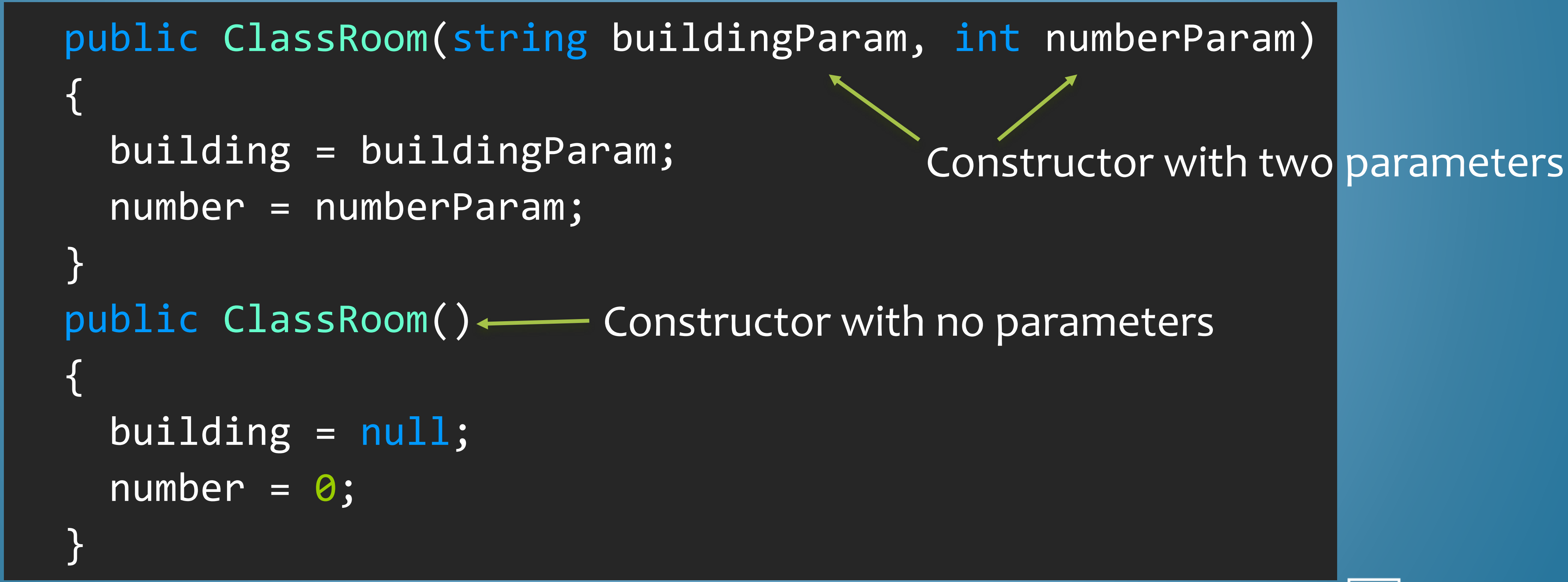
```
public void Multiply(int factor)
{
  length *= factor;
  width *= factor;
}
public void Multiply(int lengthFactor, int widthFactor)
{
  length *= lengthFactor;
  width *= widthFactor;
}
```

One parameter

Two parameters

Exceptions to Unique Name Rule

- Overloading we have already used: multiple constructors with different parameters



```
public ClassRoom(string buildingParam, int numberParam)
{
  building = buildingParam;
  number = numberParam;
}
public ClassRoom()
{
  building = null;
  number = 0;
}
```

Constructor with two parameters

Constructor with no parameters

Outline

- Name Uniqueness
- **Signatures and Overloading**
  - Type Conversions in Arguments
- Properties

Signatures and Overloading

- Method Signature = name of method + parameter types

```
public void Multiply(int lengthFactor, int widthFactor)
```

Signature: `Multiply(int, int)`

```
public void Multiply(int factor)
```

Signature: `Multiply(int)`

```
public void Multiply(double factor)
```

Signature: `Multiply(double)`

- Methods are unique as long as their *signatures* are unique

Signature Details

- Parameter names are **not** in the signature

```
public void SetWidth(int widthInMeters)
```

```
public void SetWidth(int widthInFeet)
```

❌ Same signature: SetWidth(int)

- Why? Method signature must be evaluated by calling code

```
Rectangle myRectangle = new Rectangle();
myRectangle.SetWidth(16);
```

Nothing here to indicate name of parameter. Which SetWidth method should be called?

Method name: SetWidth

Argument: an int value

Signature Details: Return Type

- Return type is **not** in the signature

```
public void Multiply(int factor)
```

```
public int Multiply(int factor)
```

❌ Same signature: `Multiply(int)`

- Calling code doesn't always know return type – you can ignore it

```
myRectangle.ComputePerimeter();
```

← `ComputePerimeter` returns `int`, but this does nothing with it

```
int result = myRectangle.Multiply(6);
```

← Definitely a call to `int Multiply`

```
myRectangle.Multiply(19);
```

← Could be a call to `void Multiply`, or a call to `int Multiply` that ignores the return value

Signature Details: Order

- Parameter order matters – *if* types are different

In class ClassRoom:

```
public void Update(int number, string name)
```

Signature: Update(int, string)

```
public void Update(string name, int number)
```

Signature: Update(string, int)

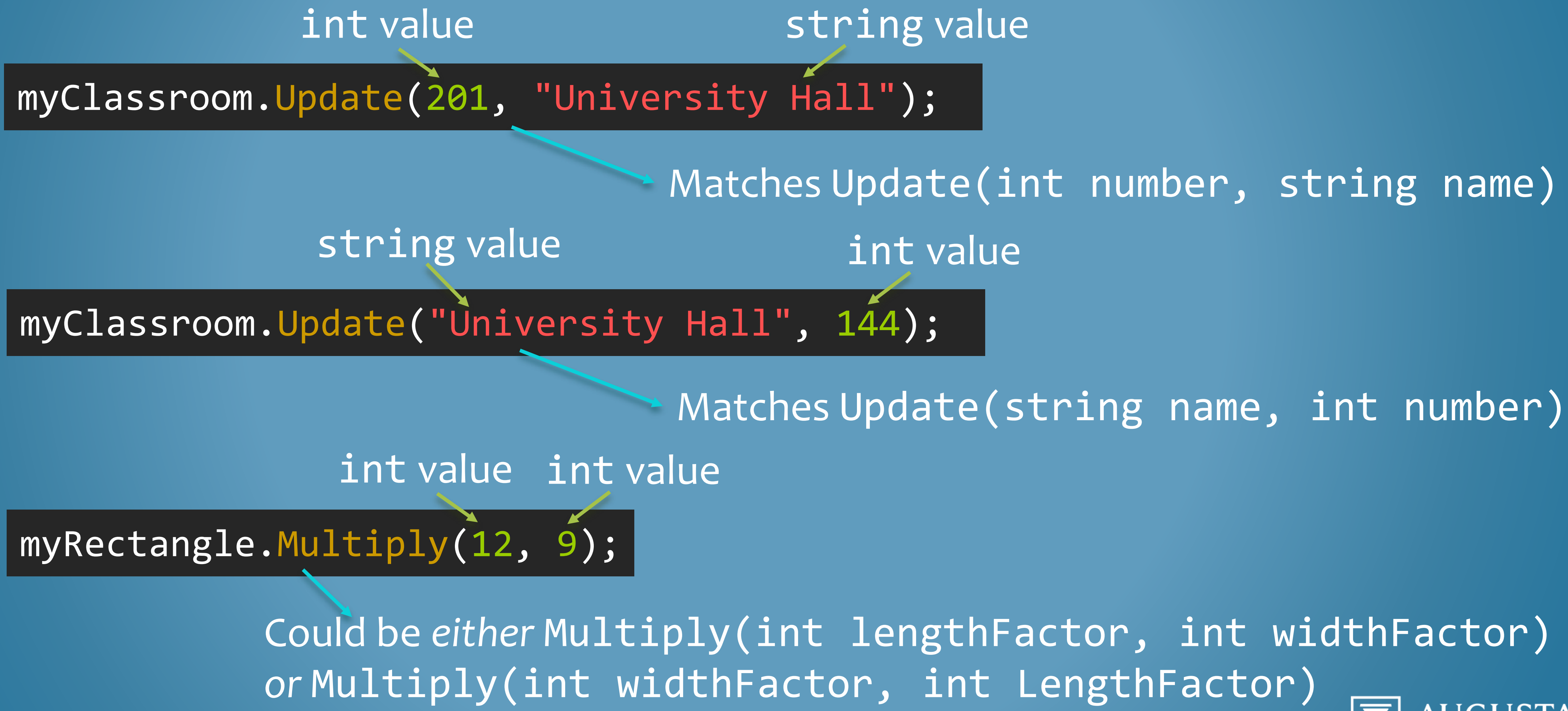
- If types are the same, no way to distinguish different orders

```
public void Multiply(int lengthFactor, int widthFactor)
```

```
public void Multiply(int widthFactor, int lengthFactor)
```

Both have same signature: Multiply(int, int)

Signature Details: Order



int value string value

```
myClassroom.Update(201, "University Hall");
```

Matches `Update(int number, string name)`

string value int value

```
myClassroom.Update("University Hall", 144);
```

Matches `Update(string name, int number)`

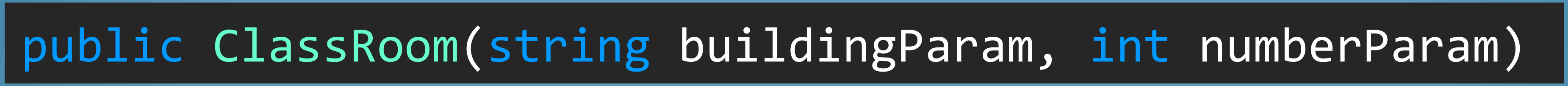
int value int value

```
myRectangle.Multiply(12, 9);
```

Could be *either* `Multiply(int lengthFactor, int widthFactor)`
or `Multiply(int widthFactor, int LengthFactor)`

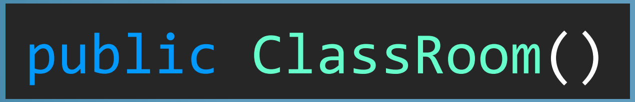
Constructors are Methods Too

- Constructors are unique if their signatures are unique



```
public ClassRoom(string buildingParam, int numberParam)
```

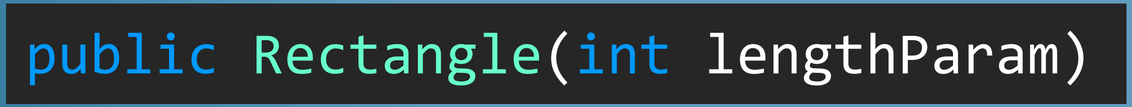
Signature: `ClassRoom(string, int)`



```
public ClassRoom()
```

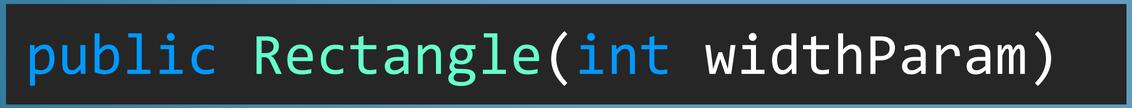
Signature: `ClassRoom()`

- Cannot have 2 constructors with the same signature



```
public Rectangle(int lengthParam)
```

Signature: `Rectangle(int)`



```
public Rectangle(int widthParam)
```

Signature is also `Rectangle(int)`

Constructors are Methods Too

- Constructor overloading was key in the lab
  - First constructor:

```
public Room(int lengthM, int widthM, string name)
```

Signature: Room(int, int, string)

Behavior: Initialize length and width assuming parameters are meters

  - Second constructor:

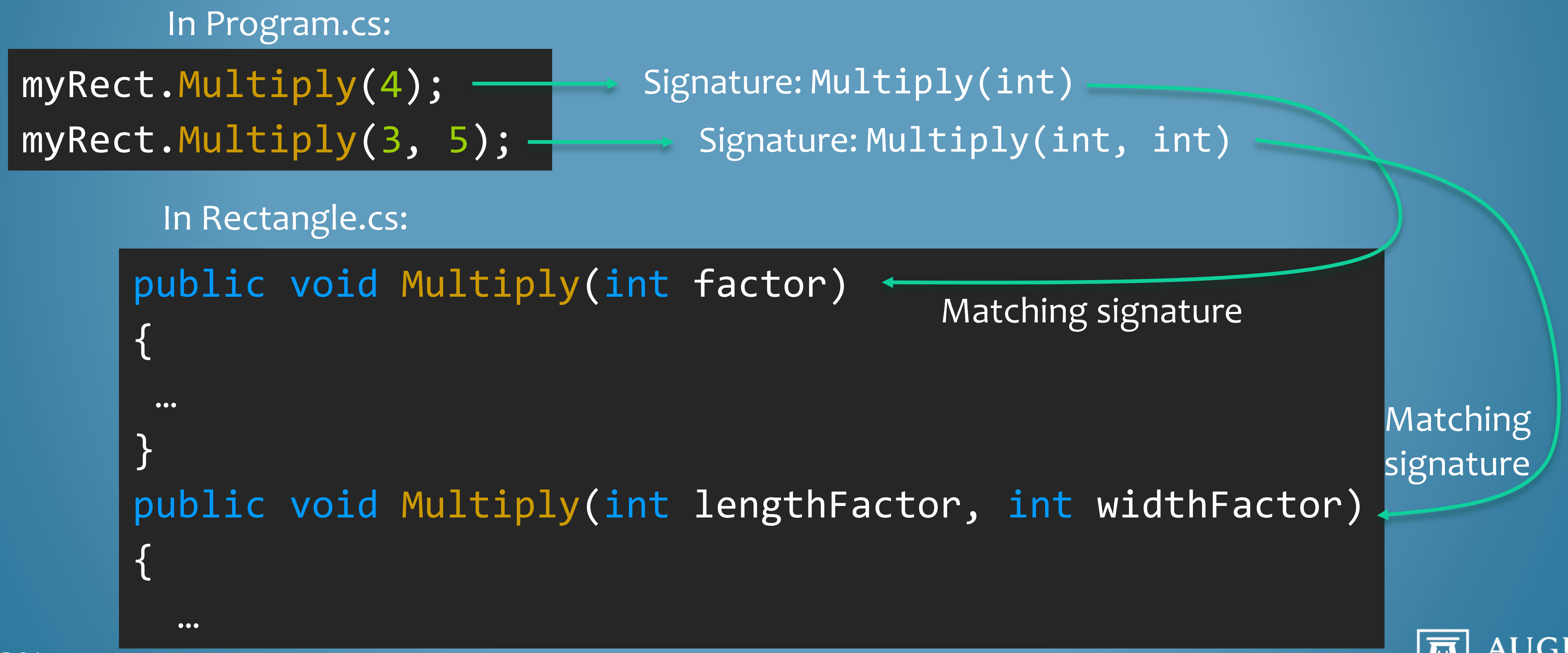
```
public Room(int lengthFt, int widthFt)
```

Signature: Room(int, int)

Behavior: Initialize length and width assuming parameters are feet

Calling Overloaded Methods

- Compare signature of call to signatures of method definitions



In Program.cs:

```
myRect.Multiply(4);
myRect.Multiply(3, 5);
```

Signature: `Multiply(int)`

Signature: `Multiply(int, int)`

In Rectangle.cs:

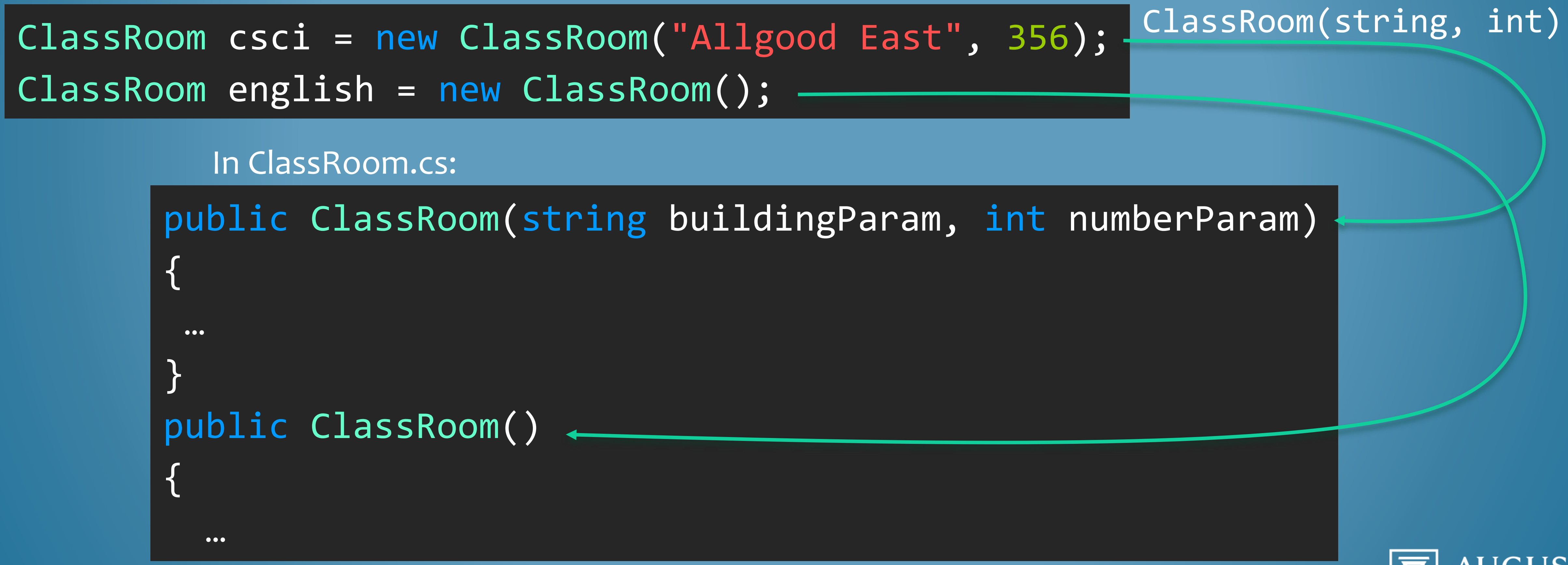
```
public void Multiply(int factor)
{
  …
}
public void Multiply(int lengthFactor, int widthFactor)
{
   …
```

Matching signature

Matching signature

Calling Overloaded Constructors

- Compare signature of call to signatures of constructors



In Program.cs:

```
ClassRoom csci = new ClassRoom("Allgood East", 356);
ClassRoom english = new ClassRoom();
```

ClassRoom(string, int)

In ClassRoom.cs:

```
public ClassRoom(string buildingParam, int numberParam)
{
 …
}
public ClassRoom()
{
   …
```

Outline

- Name Uniqueness
- Signatures and Overloading
  - **Type Conversions in Arguments**
- Properties

Argument Types and Methods

- Recall: Argument value must match type of parameter

```
Rectangle rect1 = new Rectangle();
int myIntVar = 10;
rect1.SetLength(myIntVar);
rect1.SetLength(myIntVar / 2.0);
```

Success

Error! Argument type is double, parameter type is int

```
public void SetLength(int lengthP)
{
  length = lengthP;
}
```

- If not, same conversion rules as with variable assignment

```
PreciseRectangle rect2 = new PreciseRectangle();
int myIntVar = 10;
rect2.SetLength(myIntVar / 2.0);
rect2.SetLength(myIntVar);
```

Success, argument and parameter are both double

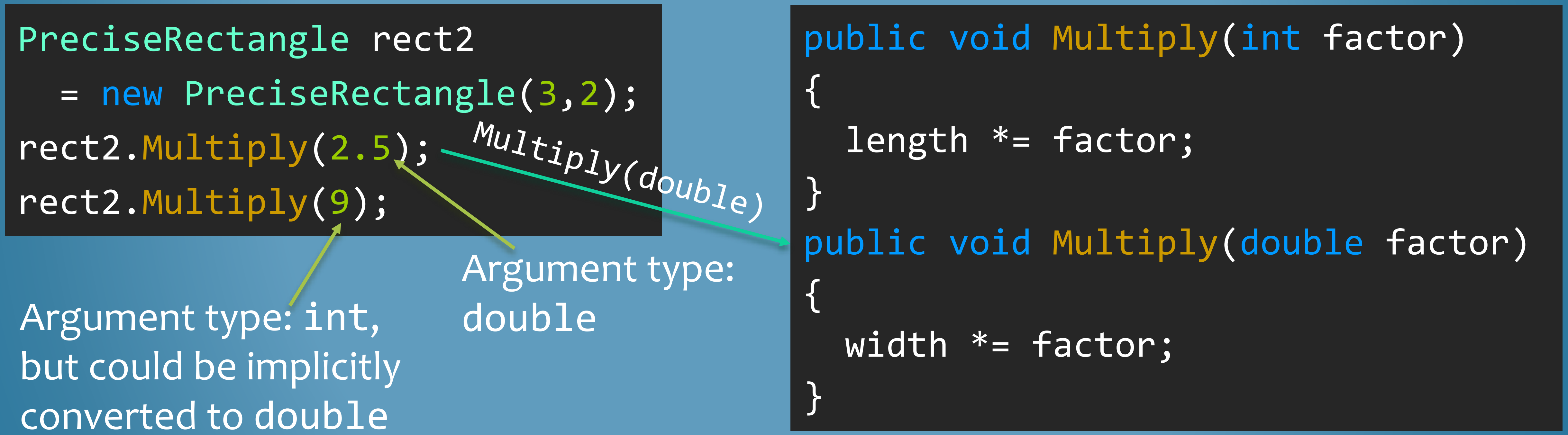
Success, argument implicitly converted to double

Argument Types and Methods

- What about overloaded methods?

In Main method:

In PreciseRectangle.cs:



```
PreciseRectangle rect2
   = new PreciseRectangle(3,2);
rect2.Multiply(2.5);
rect2.Multiply(9);
```

```
public void Multiply(int factor)
{
   length *= factor;
}
public void Multiply(double factor)
{
   width *= factor;
}
```

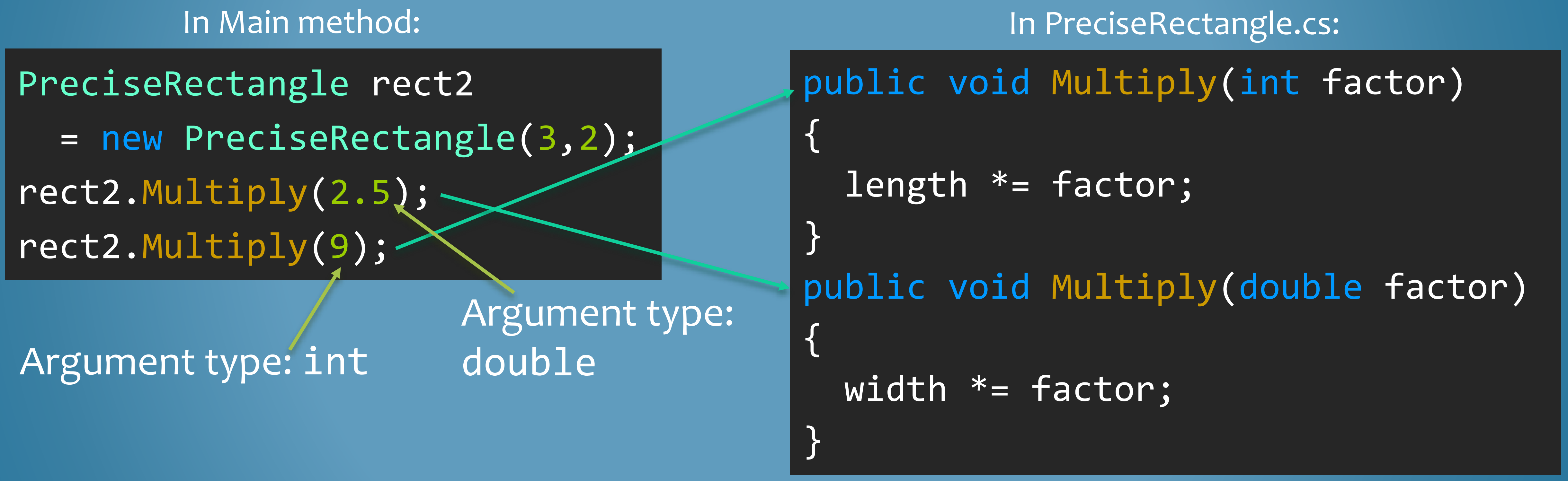
Argument type: int, but could be implicitly converted to double

Argument type: double

(these are silly methods, but just for example)

Argument Types and Methods

- Method with **closest match** to arguments gets called



In Main method:

```
PreciseRectangle rect2
   = new PreciseRectangle(3,2);
rect2.Multiply(2.5);
rect2.Multiply(9);
```

Argument type: int

Argument type: double

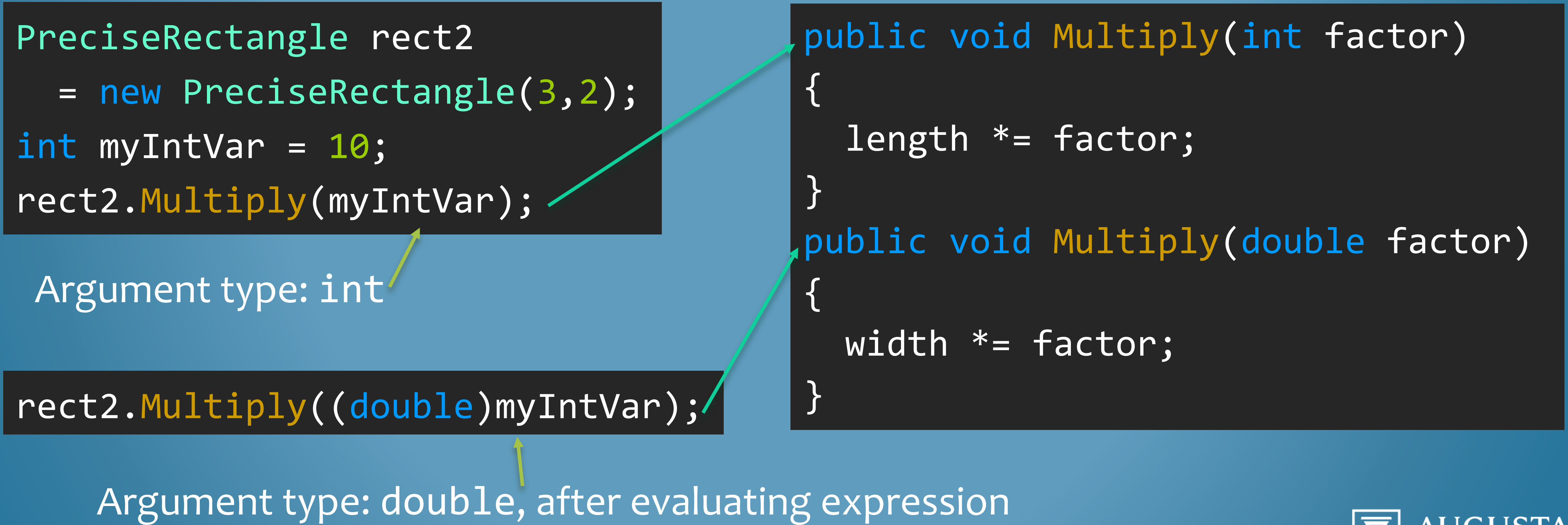
In PreciseRectangle.cs:

```
public void Multiply(int factor)
{
  length *= factor;
}
public void Multiply(double factor)
{
  width *= factor;
}
```

- Implicit conversion **only** if there is no exact match

Type Conversion and Signatures

- What if you want to choose a different overload?
  - e.g. you have an `int` variable but want to call `Multiply(double)`



```
PreciseRectangle rect2
  = new PreciseRectangle(3,2);
int myIntVar = 10;
rect2.Multiply(myIntVar);
```

Argument type: `int`

```
rect2.Multiply((double)myIntVar);
```

Argument type: `double`, after evaluating expression

```
public void Multiply(int factor)
{
  length *= factor;
}
public void Multiply(double factor)
{
  width *= factor;
}
```

Outline

- Name Uniqueness
- Signatures and Overloading
  - Type Conversions in Arguments
- **Properties**

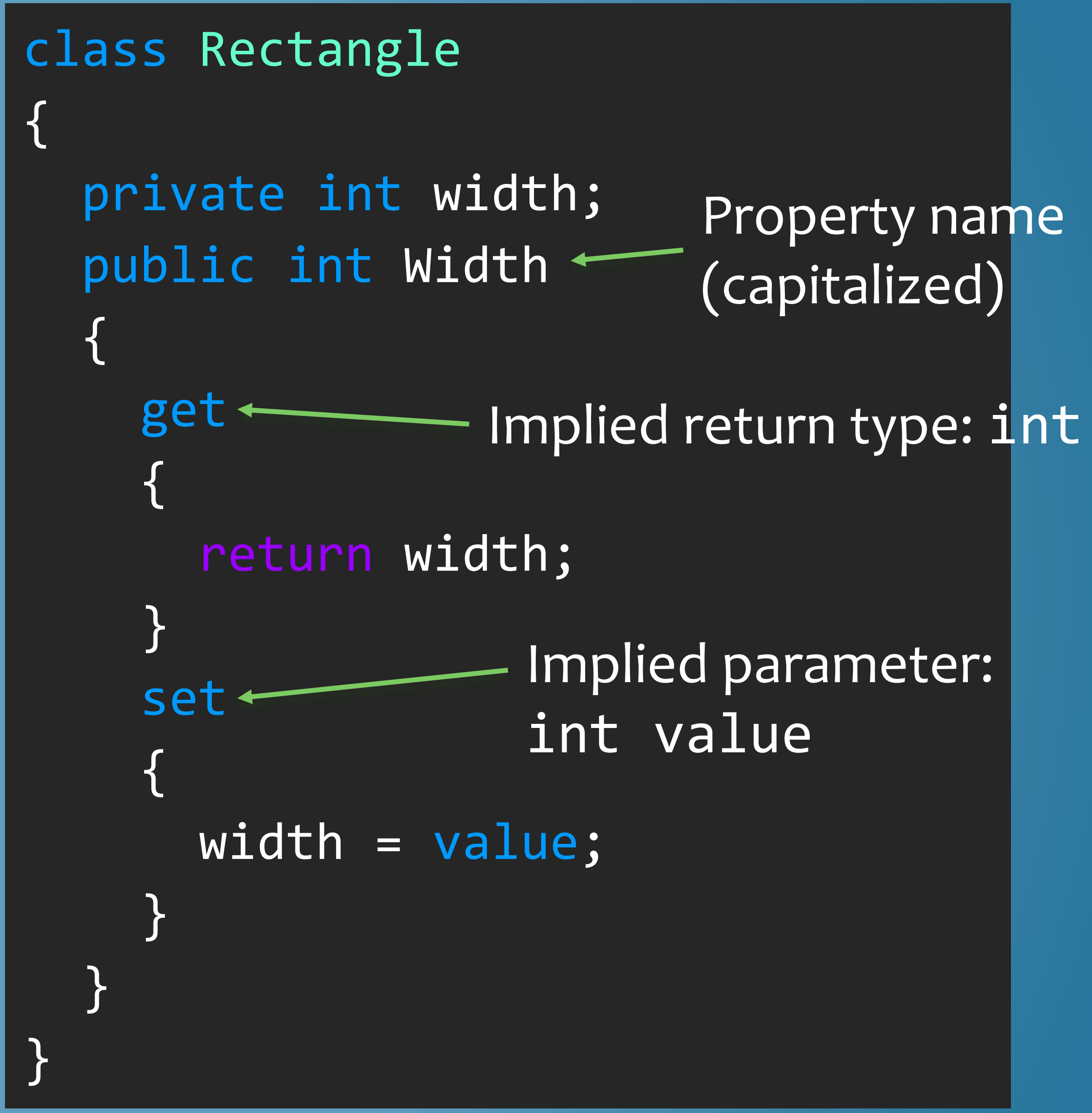
Implementing Attributes

- To give an object an attribute:
  - Declare an instance variable
  - Write a “get” accessor method
  - Write a “set” accessor method
- Properties: A shortcut for writing this code

```
class Rectangle
{
  private int width;
  public void SetWidth(int value)
  {
    width = value;
  }
  public int GetWidth()
  {
    return width;
  }
}
```

Properties

- Declaration: Type, name, get accessor, set accessor
- Keyword get: declares a method that should return the property’s value
- Keyword set: declares a method to set the property
    - Automatic parameter always named value



```
class Rectangle
{
  private int width;
  public int Width
  {
    get
    {
      return width;
    }
    set
    {
      width = value;
    }
  }
}
```

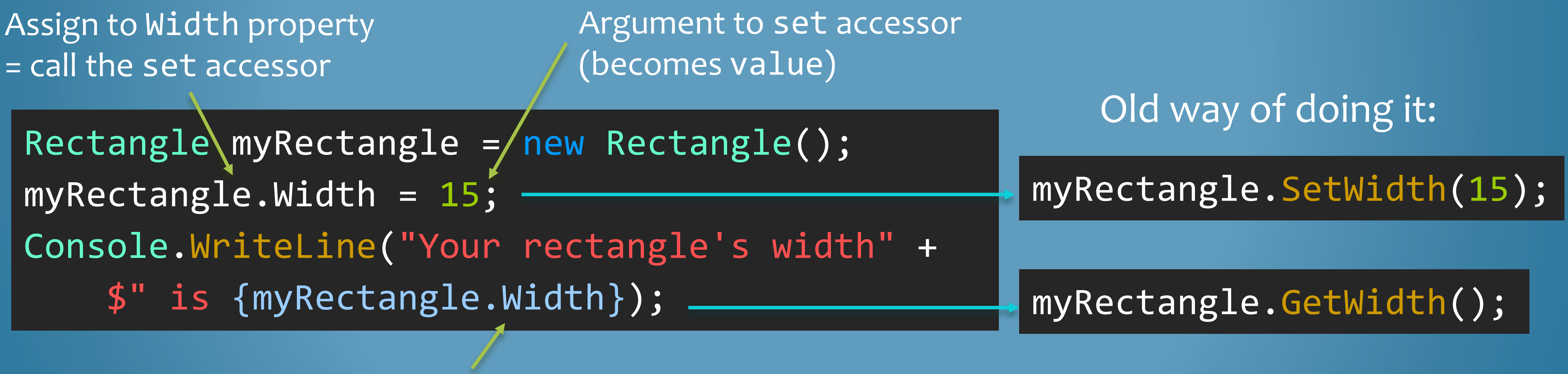
Property name (capitalized)

Implied return type: int

Implied parameter: int value

Using Properties

- Reading from a property calls the get accessor
- Writing to a property (assigning) calls the set accessor



Assign to Width property = call the set accessor

Argument to set accessor (becomes value)

```
Rectangle myRectangle = new Rectangle();
myRectangle.Width = 15;
Console.WriteLine("Your rectangle's width" +
    $" is {myRectangle.Width});
```

Old way of doing it:

```
myRectangle.SetWidth(15);
```

```
myRectangle.GetWidth();
```

Use Width as a value = call the get accessor

- Remember, C# is case-sensitive! myRectangle.width will not work

Properties Within the Class

- Can access a property within the same class
- Equivalent to calling getter and setter functions

In Rectangle.cs:

```
public int ComputeArea()
{
  return Width * Length;
}
```

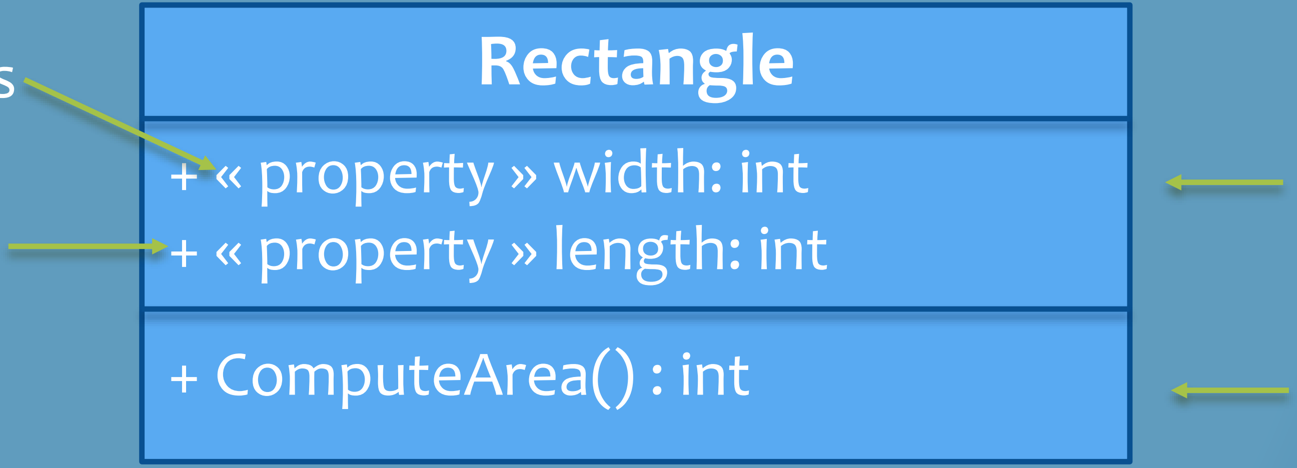
Old way of doing it:

```
public int ComputeArea()
{
  return GetWidth() * GetLength();
}
```

- Capitalized is the property, lowercase is the instance variable

Properties in UML

- Since properties automatically have get and set accessors, no need to write them in “methods” section



Rectangle

+ « property » width: int
+ « property » length: int

+ ComputeArea() : int

Guillemets

Public: the *property* is public, though the instance variable is private

Attributes

Operations