Method Signatures, Overloading, and Properties

Principles of Computer Programming I

Spring/Fall 20XX

AUGUSTA UNIVERSITY

Outline

- Name Uniqueness
- Signatures and Overloading
- Constructors in UML
- 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**
- Constructors in UML
- 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)`

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

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

Same signature: `Multiply(int)`

Signature Details

- Parameter order matters – if types are different

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

Signature: `Update(int, string)`

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

Signature: `Update(string, int)`

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

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

Both have same signature: `Multiply(int, int)`

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()`

- This was key in the lab:

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

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

Different signatures

Different code, interprets parameters as 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:

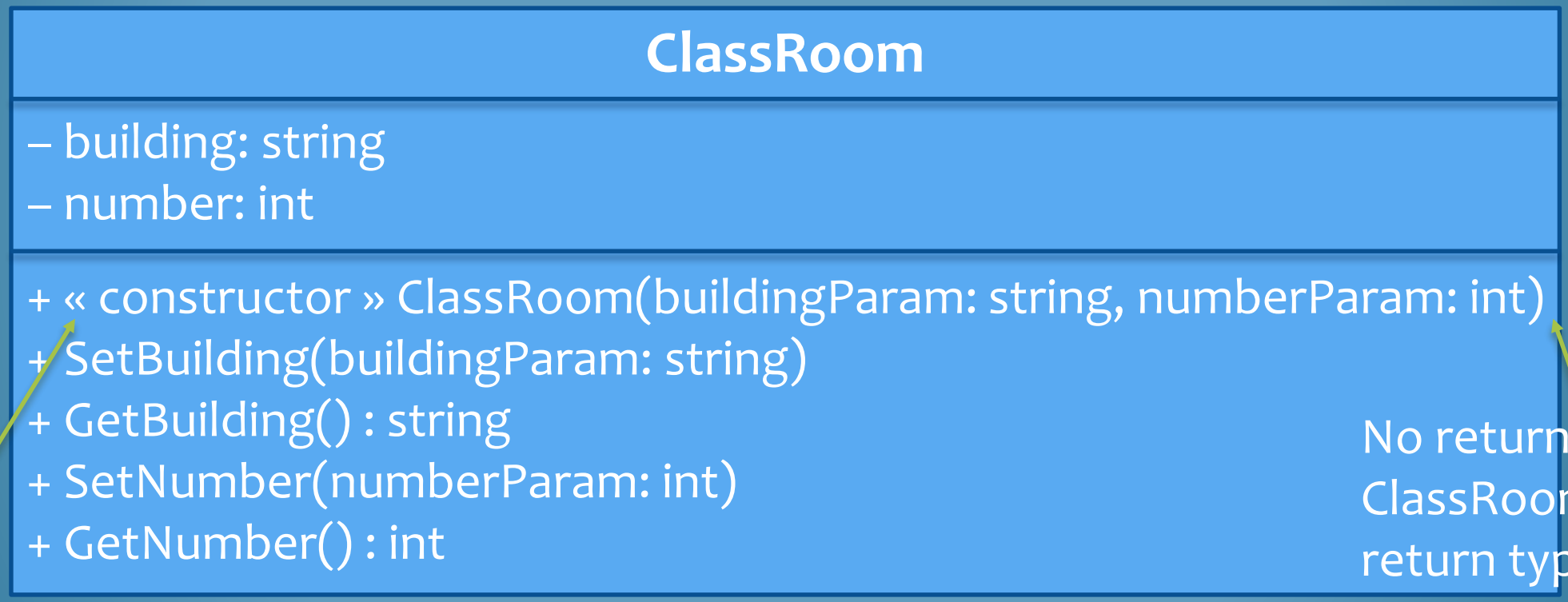
```
public ClassRoom(string buildingParam, int numberParam)
{
 …
}
public ClassRoom()
{
  …
```

Outline

- Name Uniqueness
- Signatures and Overloading
- **Constructors in UML**
- Properties

Constructors: Part of the Interface

- Non-default constructors should be planned in UML



Outline

- Name Uniqueness
- Signatures and Overloading
- Constructors in UML
- **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

```
static void Main(string[] args)
{
   Rectangle myRectangle = new Rectangle();
   myRectangle.SetLength(6);
   myRectangle.Width = 15;
   Console.WriteLine("Your rectangle's length is" +
        $"{myRectangle.GetLength()}, and " +
        $"its width is {myRectangle.Width}");
}
```

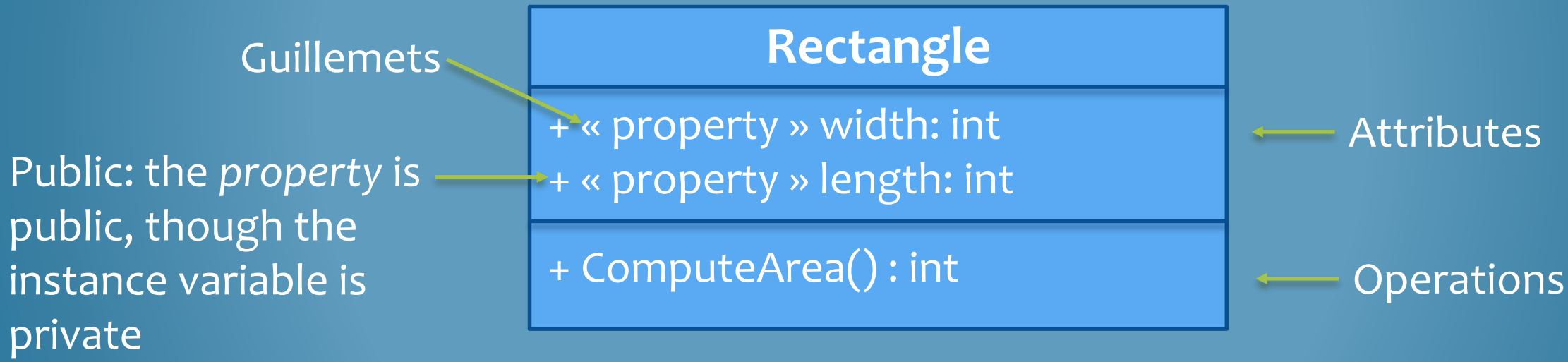
Argument to set accessor (becomes value)

Assign to Width = call the set accessor

Use Width as a value = call the get accessor

Properties in UML

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



Summary

- Name Uniqueness
- Signatures and Overloading
- Constructors in UML
- Properties