Details of Writing Classes; UML Diagrams

Principles of Computer Programming I

Spring/Fall 20XX

AUGUSTA UNIVERSITY

Outline

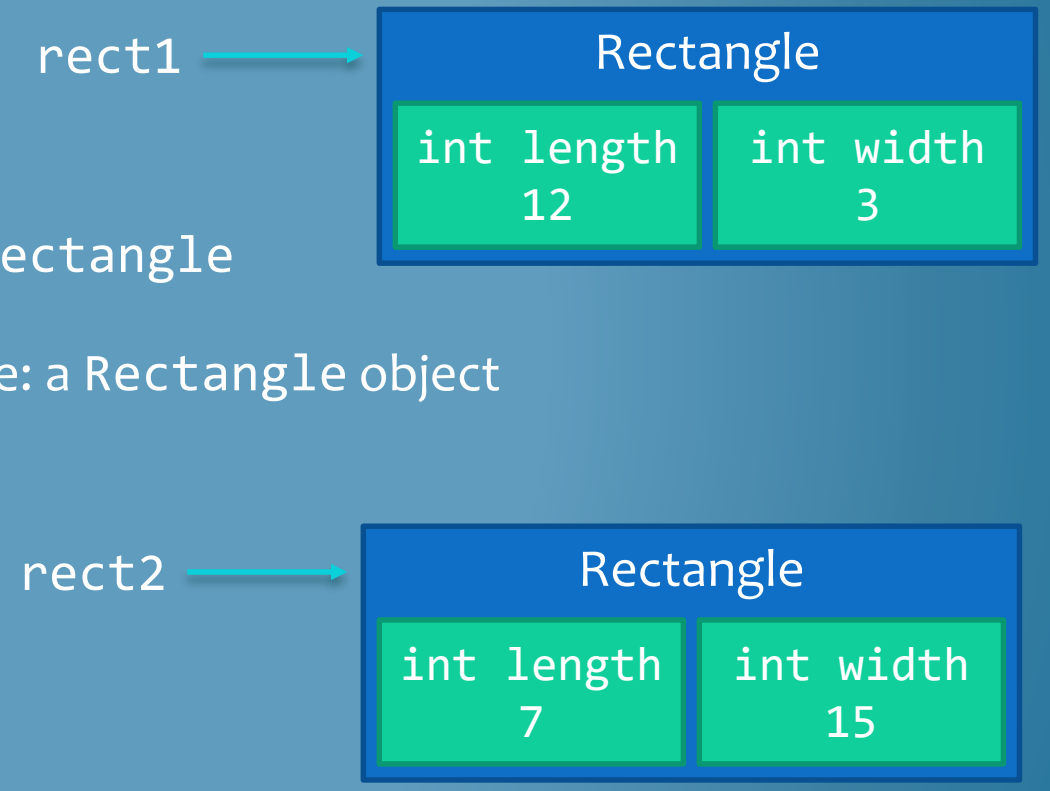
- Object and Method Details
  - Instance variable modification
  - Return types and return values
- UML Diagrams
- Variable Scope
- Constants
- Reference Types
  - Usage in variables
  - Usage in parameters

Creating and Modifying Objects

```
class Program
{
  static void Main(string[] args)
  {
    Rectangle rect1;
    rect1 = new Rectangle();
    rect1.SetLength(12);
    rect1.SetWidth(3);
    Rectangle rect2 = new Rectangle();
    rect2.SetLength(7);
    rect2.SetWidth(15);
  }
}
```

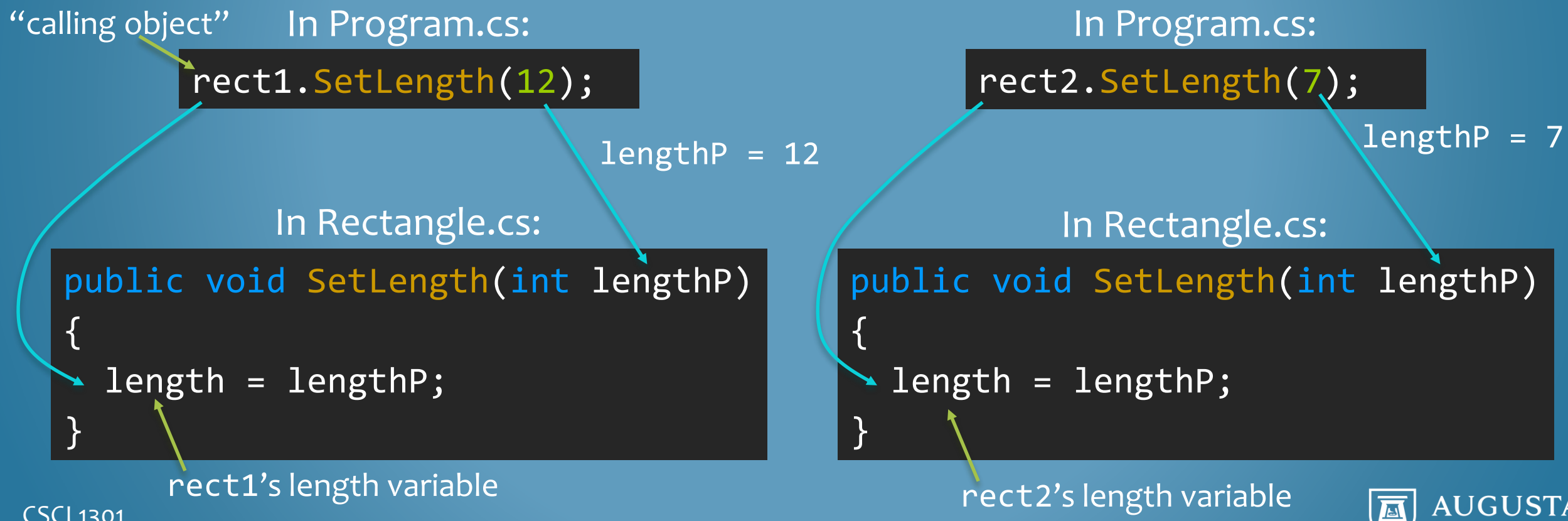
Declare a variable of type `Rectangle`

Assign it a value: a `Rectangle` object



How Does That Work?

- Calling a method transfers control to the class's code
- Which object gets modified? The one named by the variable



In More Detail: Member Access

- Dot operator = access a **member** of this object
- Usually a method, but could be an instance variable

If we wrote this…

```
class Rectangle
{
  public int length;
  public int width;
}
```

…we could do this:

```
static void Main(string[] args)
{
  Rectangle rect1 = new Rectangle();
  rect1.length = 12;
  rect1.width = 3;
}
```

This is what "violating encapsulation" looks like!

Understanding Method Calls

- Within a method call, instance variable names implicitly refer to **the calling object**'s instance variables

In Program.cs:

```
rect1.SetLength(12);
```

In Rectangle.cs:

```
public void SetLength(int lengthP)
{
    rect1.length = lengthP;
}
```

imaginary

In Program.cs:

```
rect2.SetLength(7);
```

In Rectangle.cs:

```
public void SetLength(int lengthP)
{
    rect2.length = lengthP;
}
```

imaginary

Making the Implicit Explicit

- You can make the reference explicit with keyword `this`
- `this` always names the calling (or "current") object

In Program.cs:

```
rect1.SetLength(12);
```

In Rectangle.cs:

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

this = rect1

In Program.cs:

```
rect2.SetLength(7);
```

In Rectangle.cs:

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

this = rect2

Using ≠ Modifying

- Using a variable in an expression = **reading** its value
- Variable still has the same value after reading it

```
public int ComputeArea()
{
  return length * width;
}
```

```
static void Main(string[] args)
{
  Rectangle rect1 = new Rectangle();
  rect1.SetLength(12);
  rect1.SetWidth(3);
  int area = rect1.ComputeArea();
}
```

At this point, `length` is still 12, `width` is still 3

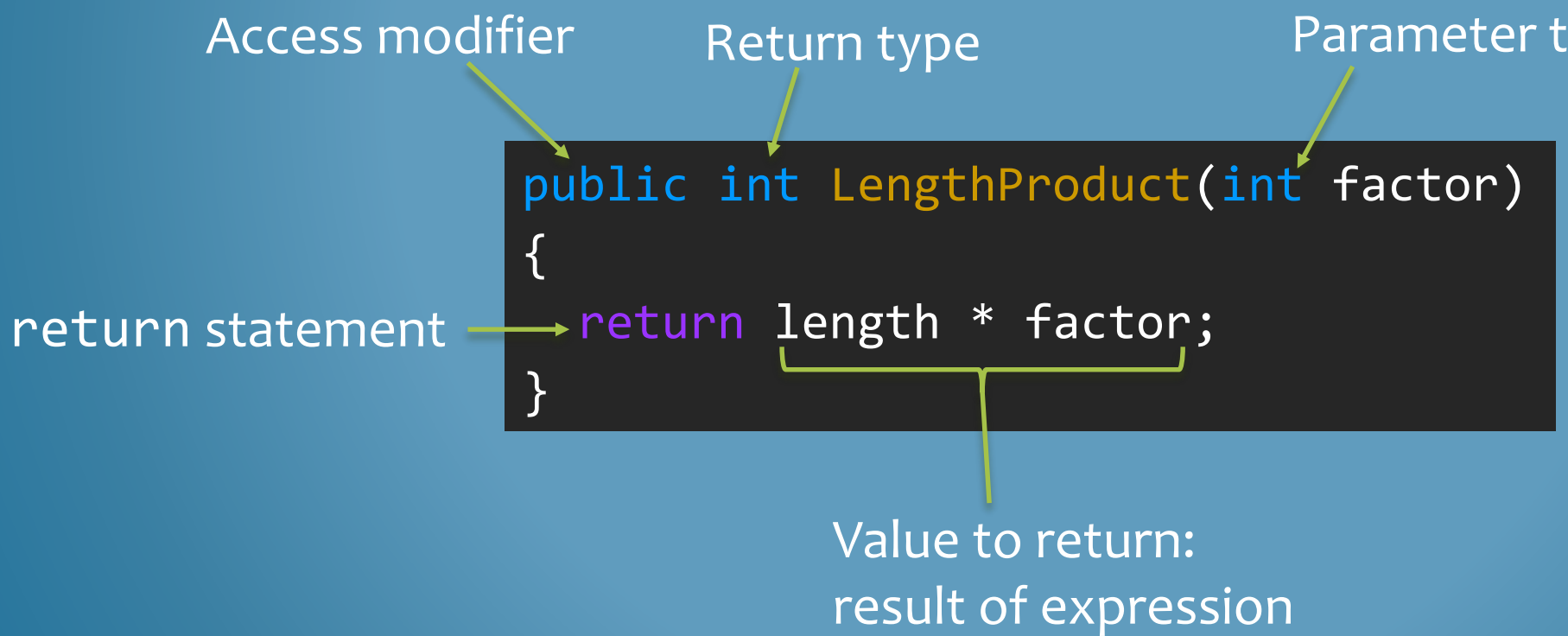
Computes `length * width` = 36

Outline

- Object and Method Details
  - Instance variable modification
  - **Return types and return values**
- UML Diagrams
- Variable Scope
- Constants
- Reference Types
  - Usage in variables
  - Usage in parameters

Methods can Return Values

- Methods are like programs: input → compute → output
- Input = parameters, output = return value



Using Return Values

- "Value" of a method call is its return value
- Can be assigned to a variable, used in math, etc.

```
static void Main(string[] args)
{
  Rectangle rect1 = new Rectangle();
  rect1.SetLength(12);
  int result = rect1.LengthProduct(2) + 1;
}
```

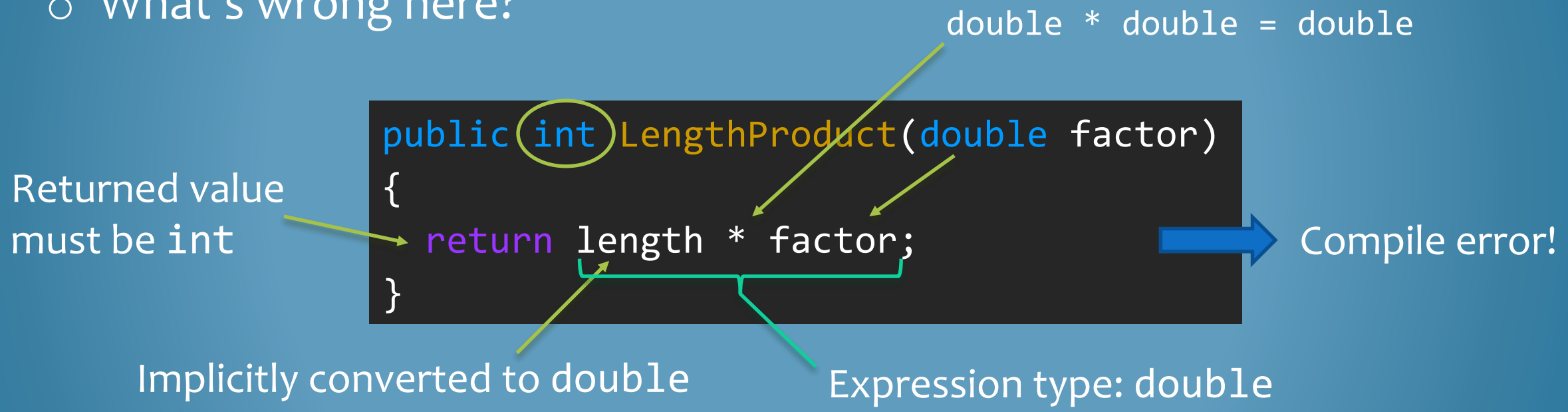
`result` is assigned value 25

value of this expression =
result of method call = 24

24 + 1 = 25

Return Requirements

- Value in `return` statement must match return type
  - What's wrong here?



Return Requirements

- Must have a return type; `void` means "nothing"
- Must return a value if the return type is not `void`

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

Error! Did not return an `int`

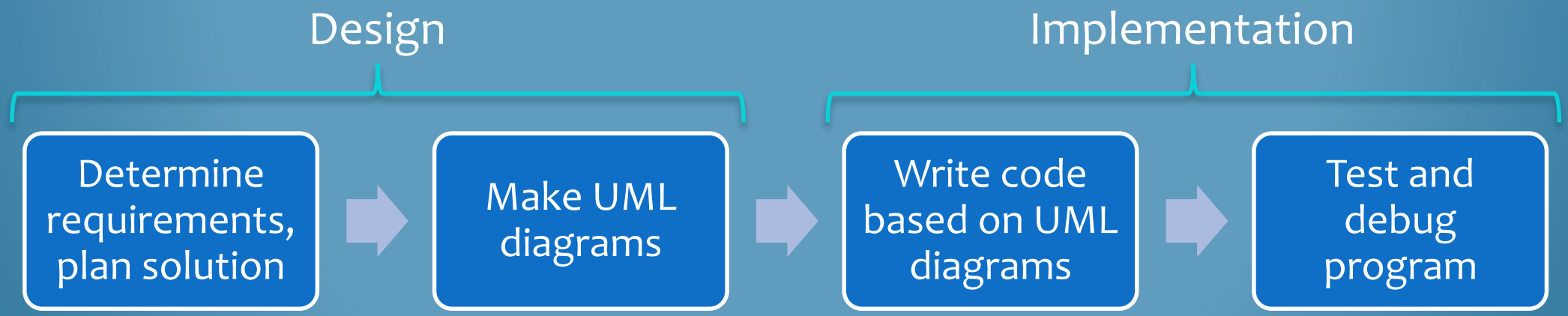
```
public void SetLength(int lengthP)
{
  length = lengthP;
}
```

Outline

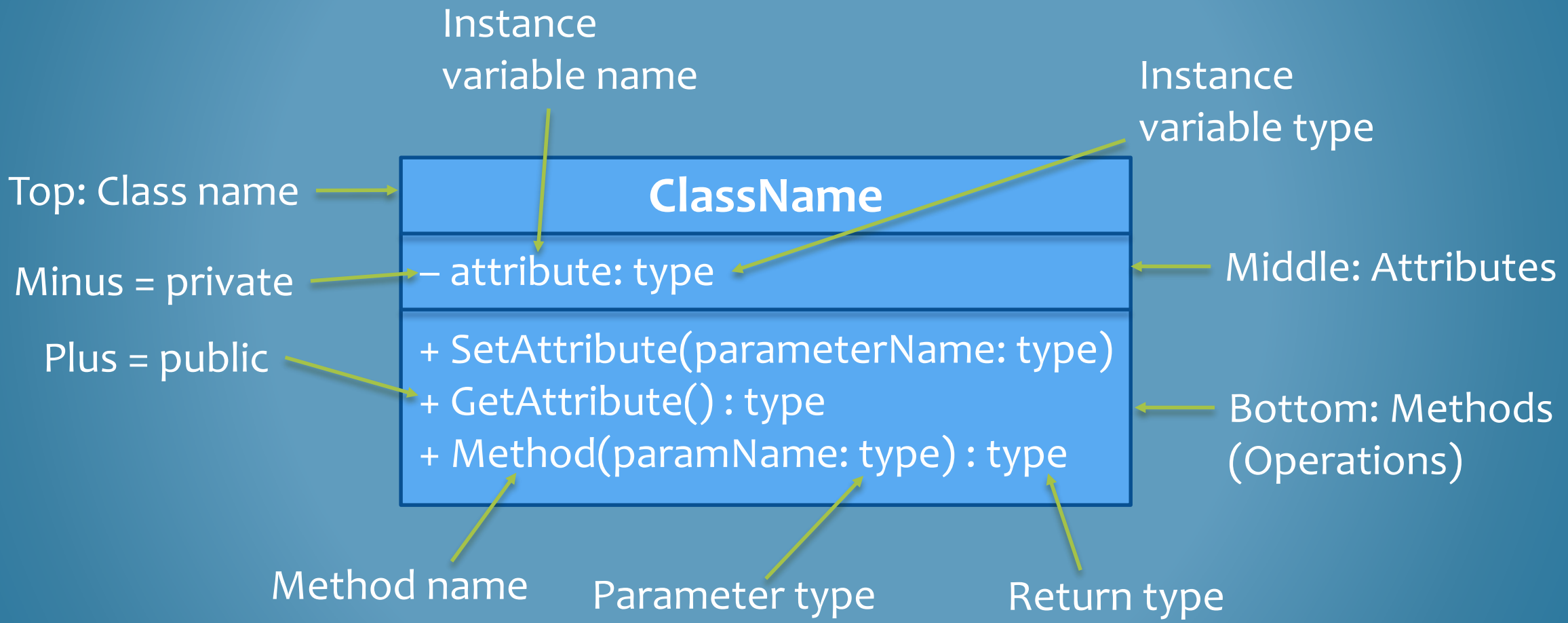
- Object and Method Details
  - Instance variable modification
  - Return types and return values
- **UML Diagrams**
- Variable Scope
- Constants
- Reference Types
  - Usage in variables
  - Usage in parameters

Planning Your Programs with UML

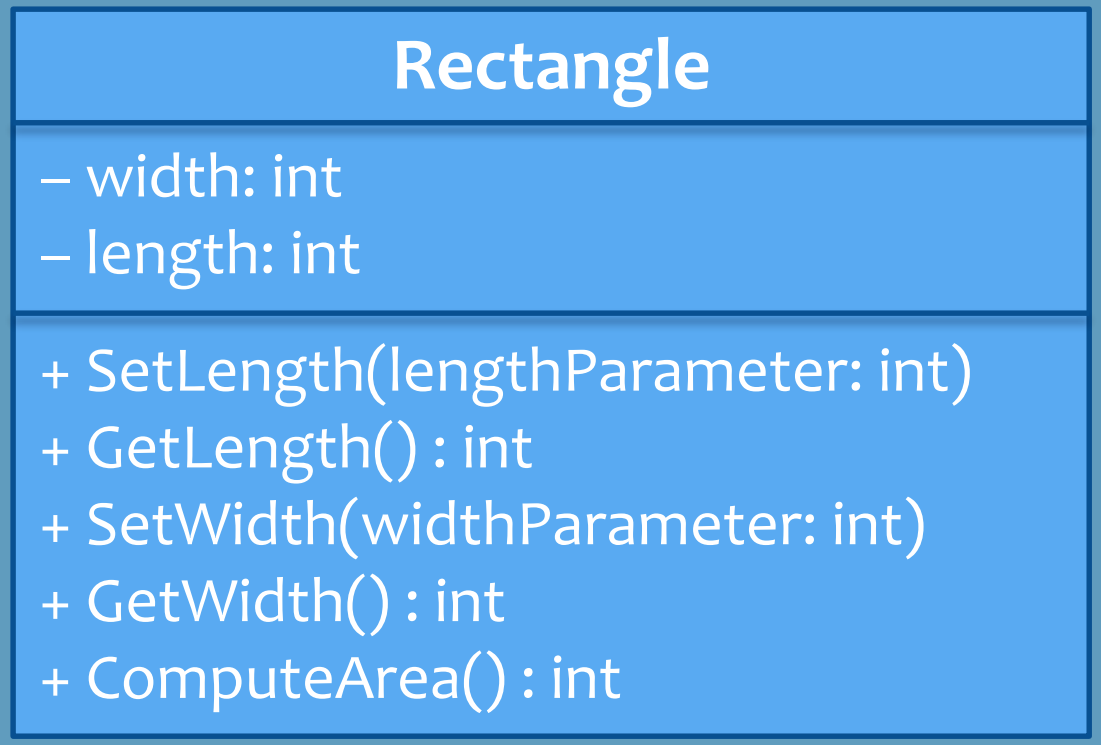
- Unified Modeling Language: specification language for software
- Describes design and structure of program with graphics
- Works for any programming language (C#, Python, Java…)
- Useful for planning, before you start writing code



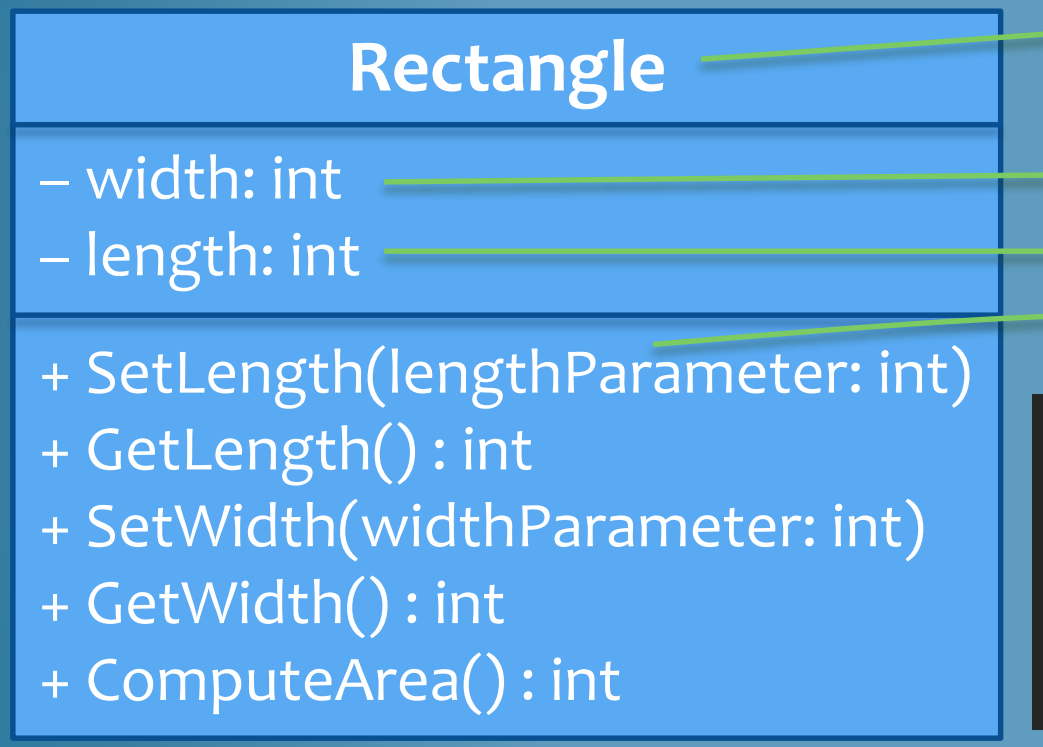
UML Class Diagram



Class Diagram for Rectangle



From Diagram to Code




```
class Rectangle
{
    private int width;
    private int length;
```

```
public void SetLength(int lengthParameter)
{
    …
}
```

Outline

- Object and Method Details
  - Instance variable modification
  - Return types and return values
- UML Diagrams
- **Variable Scope**
- Constants
- Reference Types
  - Usage in variables
  - Usage in parameters

Local vs. Instance Variables

- Instance Variables
  - Stored in object
  - Shared by all methods
  - Changes persist after method finishes executing
- Local variables
  - Visible only to one method
  - Disappear after method finishes executing

```
class Rectangle
{
  public void SwapDimensions()
  {
    int temp = length;
    length = width;
    width = temp;
  }
  public int GetLength()
  {
    return length;
  }
```

After this, `GetLength()` will return the new `length`

Cannot use `temp` here

Variable Scope

- Variables exist only in limited **time** and **space**
- Scope of a variable = region where it is accessible
- Time: *after* it is declared
- Space: within the same *code block* (defined by braces {}) where it is declared

```
class Rectangle
{
  private int length;
  private int width;
  public void SwapDimensions()
  {
    temp = 1;
    int temp;
    temp = length;
    length = width;
    width = temp;
  }
}
```

Scope = inside class Rectangle

Doesn't exist yet

Scope = inside this method

More Scope Examples

```
class Rectangle
{
  public void SwapDimensions()
  {
    int temp = length;
    length = width;
    width = temp;
  }
  public void SetWidth(int widthParam)
  {
    int temp = width;
    width = widthParam;
  }
}
```

Same name, different variables, different scopes

Parameter scope = within this method

A Scope Pitfall

- What's the problem with this code?

```
class Rectangle
{
  private int length;
  private int width;
  public void UpdateWidth(int newWidth)
  {
    int width = 5;
    width = newWidth;
  }
}
```

A Scope Pitfall

- Two variables can have the same name if they have different scopes
- The variable with the "closer" scope *shadows* or *hides* the variable with the "farther" scope
- Probably not what you wanted

```
class Rectangle
{
  private int length;
  private int width;
  public void UpdateWidth(int newWidth)
  {
    int width = 5;
    width = newWidth;
  }
}
```

Scope = all of class Rectangle

Scope = inside this method

This means the local `width`, **not** the instance variable!

Shadowing and `this`

- Keyword `this` specifies the instance variable, not the local

```
class Rectangle
{
  private int length;
  private int width;
  public void UpdateWidth(int newWidth)
  {
    int width = 5;
    this.width = newWidth;
  }
}
```

Not an instance variable

Can only mean an object member

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

Instance variable

Parameter

Outline

- Object and Method Details
  - Instance variable modification
  - Return types and return values
- UML Diagrams
- Variable Scope
- **Constants**
- Reference Types
  - Usage in variables
  - Usage in parameters

Constants

- Named values that can't change – like variables, but don't vary
- Can only be built-in types (int, double, etc.) not your own classes
- Convention: Named using ALL CAPS

```
class Calendar
{
  public const int MONTHS = 12;
  private int currentMonth;
}
```

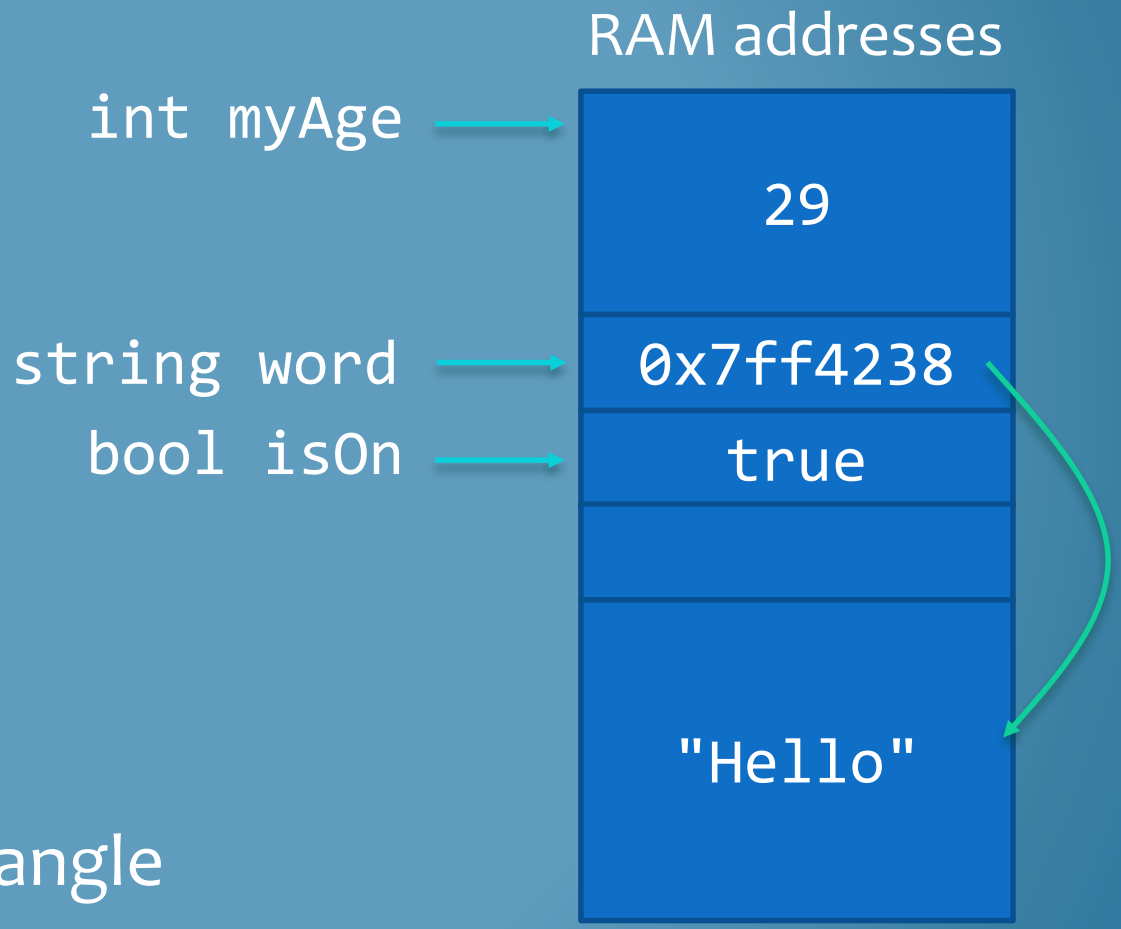
```
static void Main(string[] args)
{
  decimal yearlyPrice = 2000.0m;
  decimal monthlyPrice = yearlyPrice
    / Calendar.MONTHS;
  Calendar myCal = new Calendar();
}
```

Outline

- Object and Method Details
  - Instance variable modification
  - Return types and return values
- UML Diagrams
- Variable Scope
- Constants
- **Reference Types**
  - Usage in variables
  - Usage in parameters

Recall: Value vs. Reference Types

- **Value Type** variables: Memory location stores the value
  - `int`, `long`, `float`, `double`, `decimal`, `char`, `bool`
- **Reference Type** variables: Memory location stores a reference to the value
  - `string`, `object`
  - Any object you create, e.g. Rectangle



Assigning Reference Variables

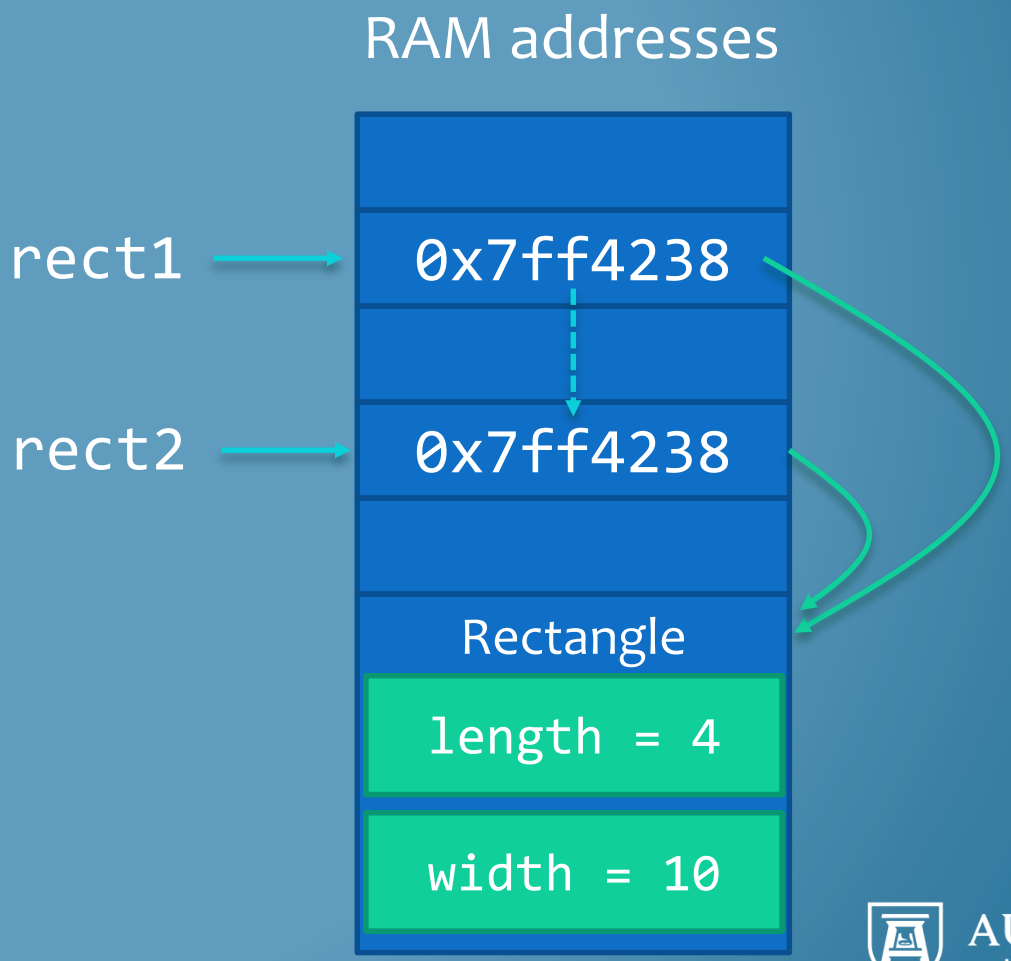
```
static void Main(string[] args)
{
  Rectangle rect1 = new Rectangle();
  rect1.SetLength(8);
  rect1.SetWidth(10);
  Rectangle rect2 = rect1;
  rect2.SetLength(4);
  Console.WriteLine($"Rectangle 1: {rect1.GetLength()} "
    + $"by {rect1.GetWidth()}");
  Console.WriteLine($"Rectangle 2: {rect2.GetLength()} "
    + $"by {rect2.GetWidth()}");
}
```

What does this print?

Assigning Reference Variables

- Assignment copies the variable (i.e. the reference), not the object it refers to

```
Rectangle rect1 = new Rectangle();
rect1.SetLength(8);
rect1.SetWidth(10);
Rectangle rect2 = rect1;
rect2.SetLength(4);
```



Application to Method Parameters

- Parameters are initialized by assignment:

```
rect1.SetLength(8);
```

```
lengthP = 8;
```

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

- If parameter is an object type, this will copy the reference

Objects Can Change Other Objects

Add this method to class Rectangle (in Rectangle.cs):

```
public void CopyToOther(Rectangle otherRect)
{
   otherRect.SetLength(length);
   otherRect.SetWidth(width);
}
```

Modifies the object referred to by otherRect

A reference to an object; the object exists outside the method

Use it like this (in Program.cs):

```
Rectangle rect1 = new Rectangle();
Rectangle rect2 = new Rectangle();
rect1.SetLength(8);
rect1.SetWidth(10);
rect1.CopyToOther(rect2);
```

rect2 starts with length 0 and width 0

Now rect2 has length 8 and width 10