# Details of Object Mechanics; UML Diagrams

Principles of Computer Programming I

Spring/Fall 20XX

AUGUSTA UNIVERSITY

# Outline

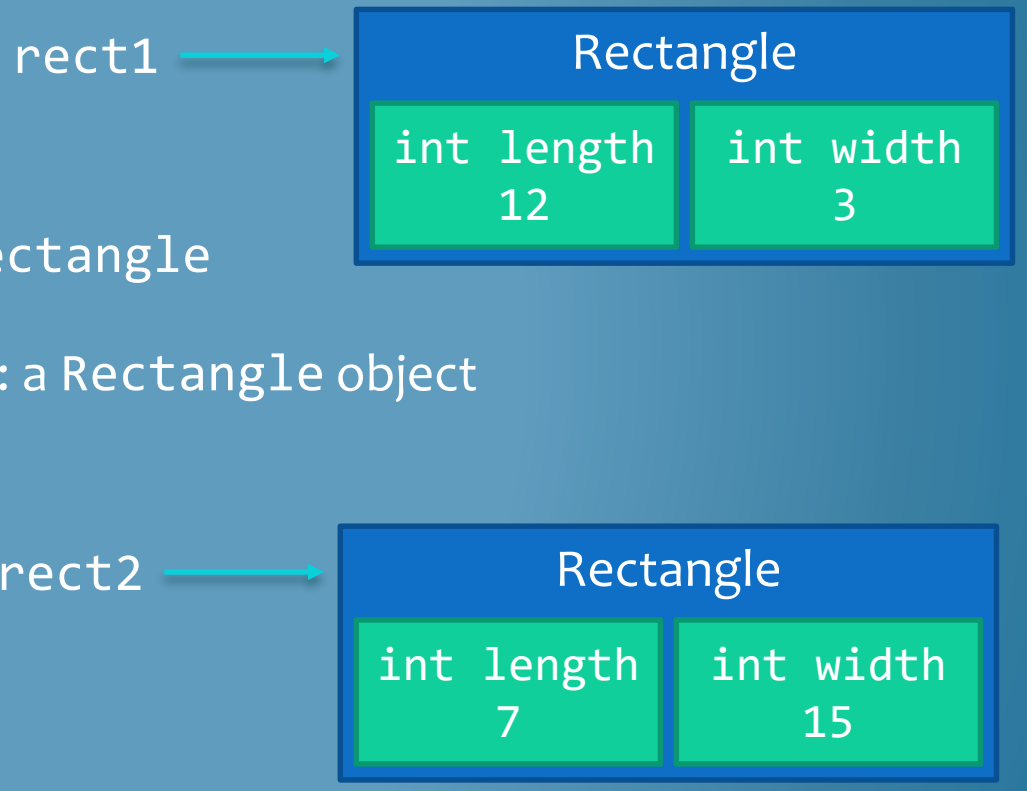
- Object and Method Details
  - Instance variable modification
  - Return types and return values
  - Parameters and arguments
- UML Diagrams

# 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

"calling object"

In Program.cs:

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

lengthP = 12

In Rectangle.cs:

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

rect1's length variable

In Program.cs:

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

lengthP = 7

In Rectangle.cs:

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

rect2's length 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;
}
```

Reads the instance variables, does not assign them

```
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**
  - Parameters and arguments
- UML Diagrams

# Method Input and Output

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

Access modifier

Return type

Parameter type

```
public int LengthProduct(int factor)
{
    return length * factor;
}
```

return statement

Value to return:
result of expression

# 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

# Using Return Values

- Implicit/explicit conversion rules apply to return values

```
double result = rect1.LengthProduct(2) * 1.5;
```

value of this expression = 24

a double value

Implicit conversion

36.0 ← 24.0 * 1.5

```
int intResult = (int)(rect1.LengthProduct(2) * 1.5);
```

Cast expression to int

value of this expression = 36.0

36

# Using Return Values

- If the return type is `void`, there is no return value to use

```
public void SwapDimensions()
{ …
```

```
int result = rect1.SwapDimensions() + 5;
```

Compile error: Cannot add `void` to `int`

- What if you don't use the return value?

```
Rectangle rect1 = new Rectangle();
rect1.SetLength(10);
rect1.LengthProduct(5);
```

Expression returns value 50

```
public int LengthProduct(int factor)
{
  return length * factor;
}
```

- `rect1`'s length is still `10`
- Value `50` not stored anywhere

# Return Requirements

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

`double * double = double`

```
public int LengthProduct(double factor)
{
  return length * factor;
}
```

Returned value must be `int`

Compile error!

Implicitly converted to `double`

Expression type: `double`

# 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
  - **Parameters and arguments**
- UML Diagrams

# Parameters: Input to Method

- When a method needs input *from the caller*, use a parameter
- Value passed to parameter in a single call = **argument**

In the Main method:

```
Time myTime = new Time();
myTime.SetHours(2);
myTime.SetMinutes(30);
myTime.SetSeconds(33);
myTime.AddMinutes(30);
myTime.AddMinutes(15);
```

Call method twice, with different arguments

Parameter type

Parameter name

```
public void AddMinutes(int numMinutes)
{
  int totalNewMinutes = minutes
    + numMinutes;
  minutes = totalNewMinutes % 60;
  hours += totalNewMinutes / 60;
}
```

numMinutes is 30 15

minutes is 00

result is 05

result is 0

# Argument Passing Details

- Parameter variable is *assigned* the value of the argument

```
myTime.AddMinutes(30);
```

```
numMinutes = 30;
```

```
myTime.AddMinutes(15);
```

```
numMinutes = 15;
```

```
public void AddMinutes(int numMinutes)
{
  int totalNewMinutes = minutes
    + numMinutes;
  minutes = totalNewMinutes % 60;
  hours += totalNewMinutes / 60;
}
```

- Same rules as any other assignment

# Expressions in Arguments

- An expression must be evaluated before its result can be used
- Result type must match parameter type

```
Rectangle rect1 = new Rectangle();
rect1.SetLength(2 + 3);
int myIntVar = 16;
rect1.SetLength(myIntVar / 2);
rect1.SetLength(myIntVar * 3.5);
```

Result is 5

```
lengthP = 2 + 3;
```

Result is 8

```
lengthP = myIntVar / 2;
```

```
lengthP = myIntVar * 3.5;
```

Result is 56.0

Error! Can't assign double value to int variable

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

# Variables in Arguments

- Assignment stores a copy or "snapshot" of the value
- Variables used in arguments do not get "stored" in the object

```
int firstVar = 11;
int secondVar = firstVar;
firstVar *= 2;
Console.WriteLine(secondVar);
```

secondVar gets value 11

firstVar is now 22

secondVar is still 11, this displays "11"

```
Rectangle rect1 = new Rectangle();
int myIntVar = 16;
rect1.SetLength(myIntVar);
myIntVar *= 2;
Console.WriteLine(rect1.GetLength());
```

rect1's length set to 16

myIntVar is now 32

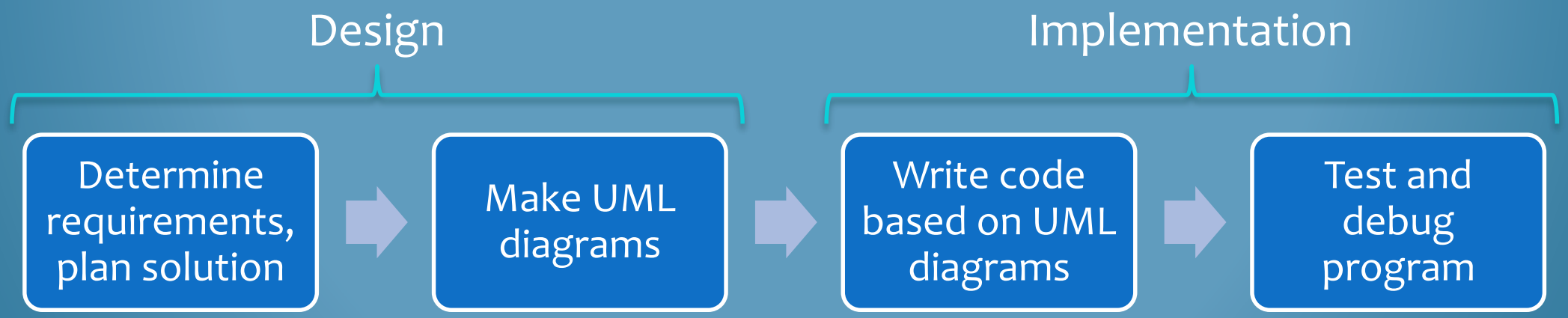
rect1's length is still 16, this displays "16"

# Outline

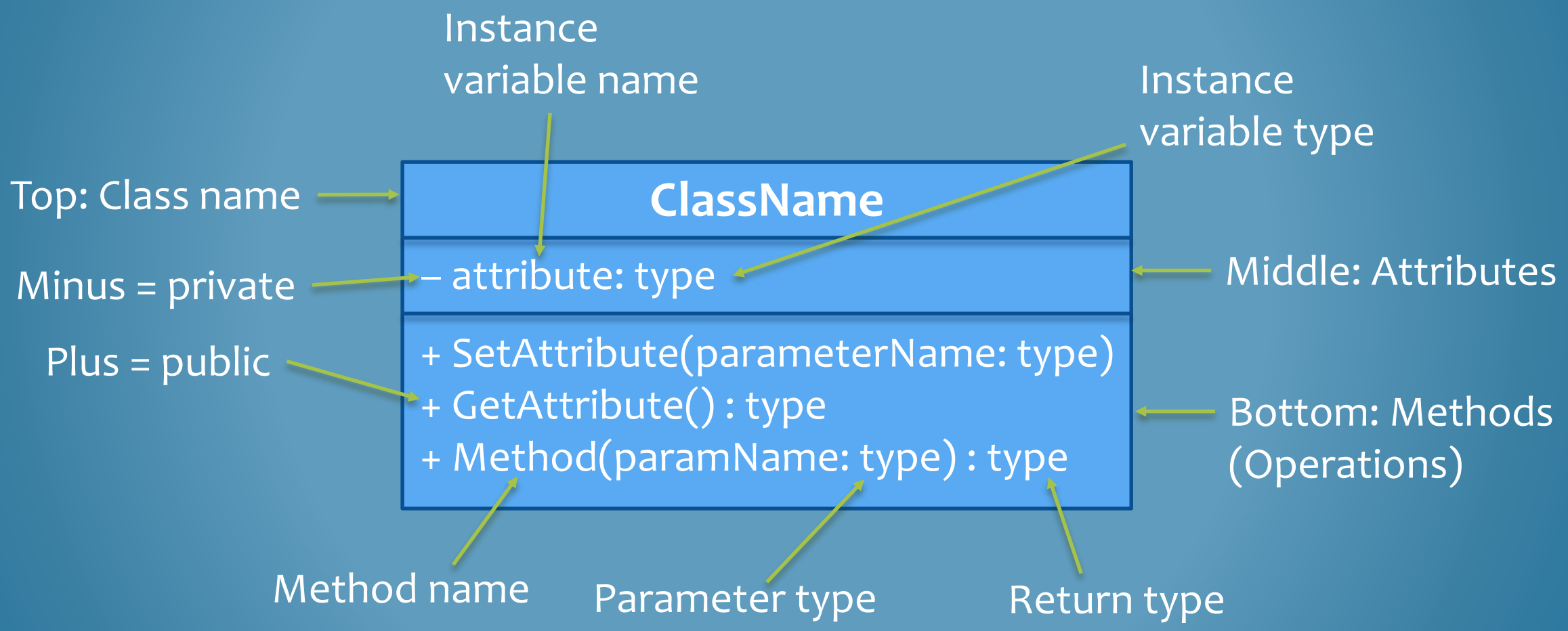
- Object and Method Details
  - Instance variable modification
  - Return types and return values
  - Parameter types and expressions
- **UML Diagrams**

# 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

**Rectangle**

– width: int
– length: int

+ SetLength(lengthParameter: int)
+ GetLength() : int
+ SetWidth(widthParameter: int)
+ GetWidth() : int
+ ComputeArea() : int

# From Diagram to Code

**Rectangle**

– width: int
– length: int

+ SetLength(lengthParameter: int)
+ GetLength() : int
+ SetWidth(widthParameter: int)
+ GetWidth() : int
+ ComputeArea() : int

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

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