COMP1618
Lecture 4
Classes and UML

Lecture Objectives

We will discuss the following main topics on:

- Object-oriented programming
- Class vs. Object
- Constructors, Data Field Encapsulation
- Design a Class with UML
- Driver class

OO Programming Concepts

- Object-oriented programming (OOP) involves programming using objects.
- An *object* represents an entity in the real world that can be distinctly identified.
  - For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects.

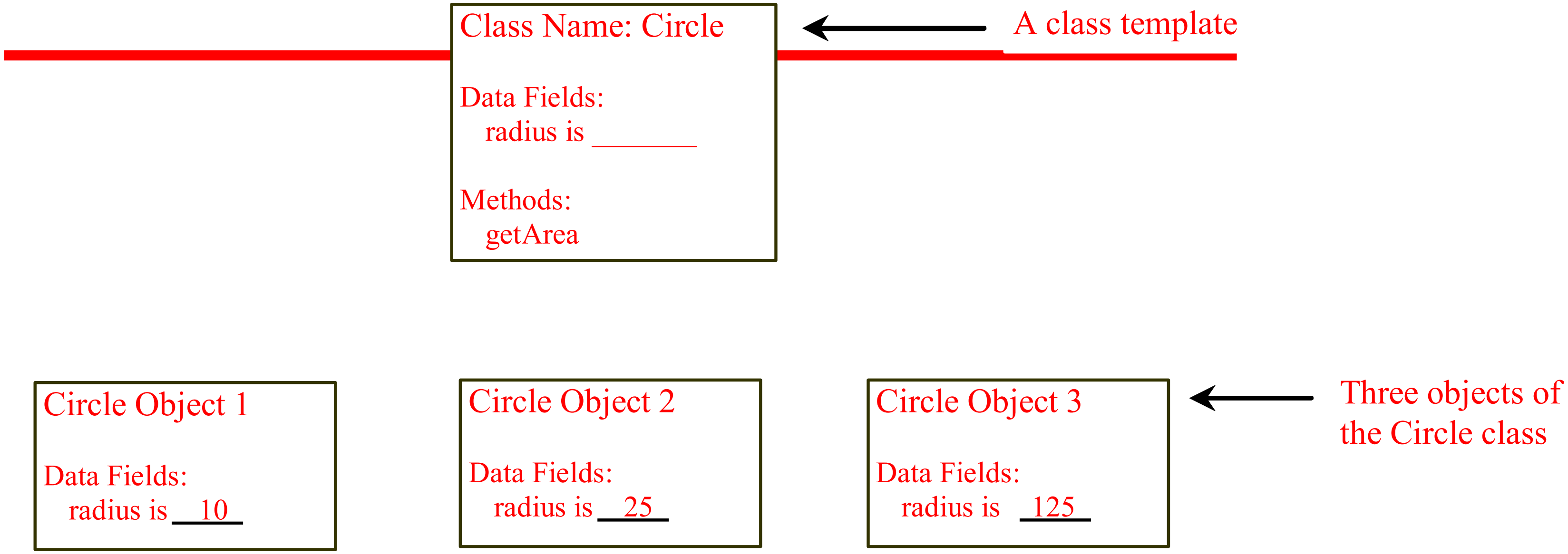
OO Programming Concepts

- Objects of the same type are defined using a common class.

- A *class* is a template, blueprint, or *contract* that defines what an object’s data fields and methods will be.

- An object is an instance of a class. You can create many instances of a class. Creating an instance is referred to as *instantiation*.

Objects



Class Name: Circle

Data Fields:
radius is _______

Methods:
getArea

A class template

Circle Object 1

Data Fields:
radius is __10_

Circle Object 2

Data Fields:
radius is __25_

Circle Object 3

Data Fields:
radius is _125_

Three objects of
the Circle class

An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.

Classes

A Java class uses variables to define data fields and methods to define behaviors.

Additionally, a class provides a special type of methods, known as constructors, which are invoked to construct objects from the class.

Designing a Class

- When designing a class, decisions about the following must be made.
  - what data must be accounted for,
  - what actions need to be performed,
  - what data can be modified,
  - what data needs to be accessible, and
  - …..
- Class design typically is done with the aid of a Unified Modeling Language (UML) diagram.

Attributes

- The data elements of a class defines the object to be instantiated from the class.

- Example: A *circle* class has an attribute:
  - Radius

- The attributes are then accessed by methods within the class.

Constructors

- Constructors are used to perform operations at the time an object is created.

- Constructors typically initialize instance fields and perform other object initialization tasks.

Constructors

- Constructors have a few special properties that set them apart from normal methods.
  - Constructors have the same name as the class.
  - Constructors have no return type (not even void).
  - Constructors may not return any values.
  - Constructors are typically public.
- Example: ConstructorDemo.java
- Example: RoomConstructor.java

The Default Constructor

- If a constructor is not defined, Java provides a default constructor.
- The default constructor is a constructor with no parameters.
- Default constructors are used to initialize an object in a default configuration.

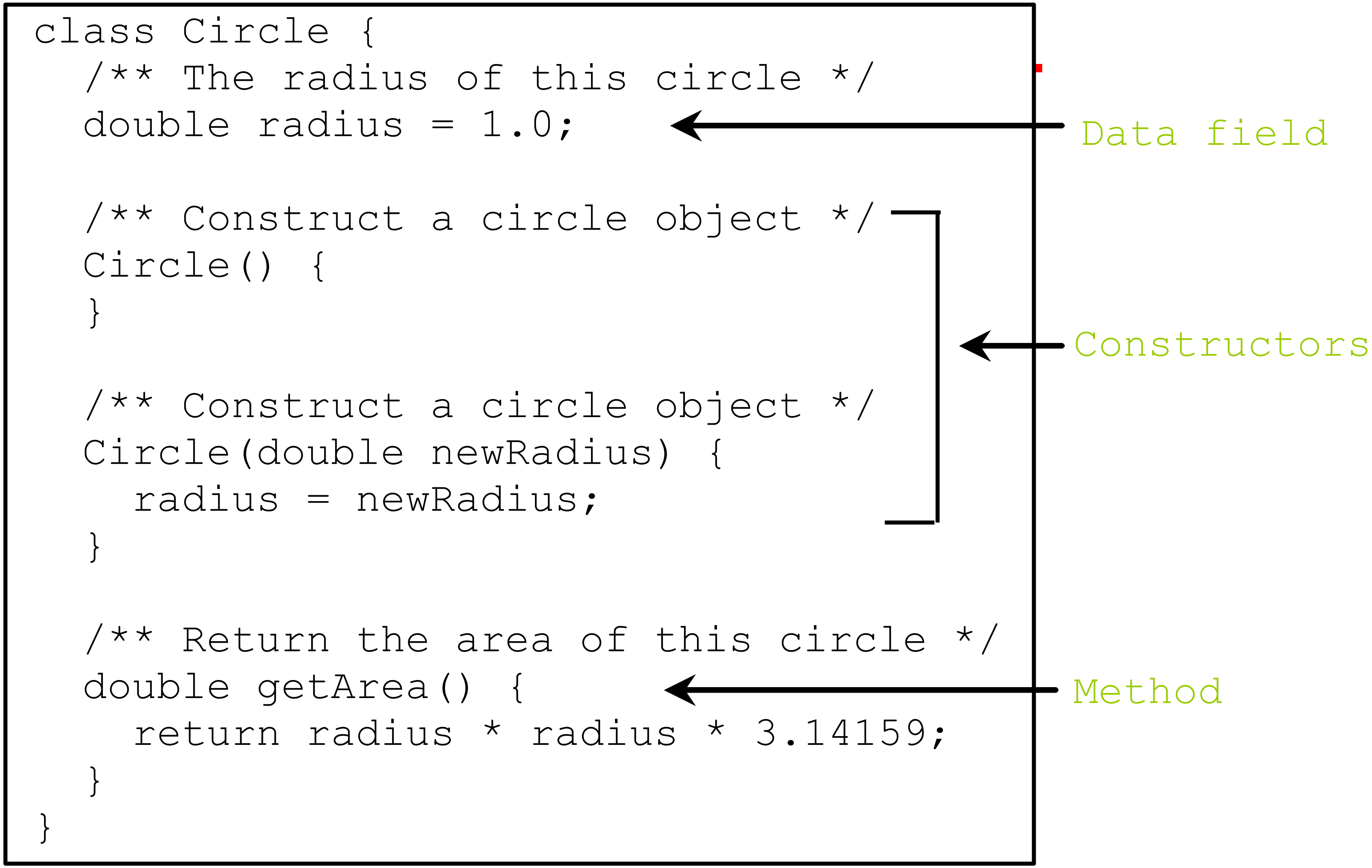
Methods

- The class’ methods define what actions an instance of the class can perform
- Methods headers have a format:

```
AccessModifier ReturnType MethodName(Parameters)
{
   //Method body.
}
```

- Methods that need to be used by other classes should be made public.

Classes



```
class Circle {
  /** The radius of this circle */
  double radius = 1.0;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * 3.14159;
  }
}
```

Data Field Encapsulation

Making data fields private helps to make code easy to maintain since the client programs cannot modify them.

To prevent direct modifications of data fields, declaring the data fields private is known as *data field encapsulation*.

To make a private data field accessible, provide a *getter/ accessor* method to return its value.

To enable a private data field to be updated, provide a *setter/ mutator* method to set a new value.

Accessors and Mutators

- For the `Rectangle` class, the mutators and accessors are:
  - `setLength` : Sets the value of the `length` field.

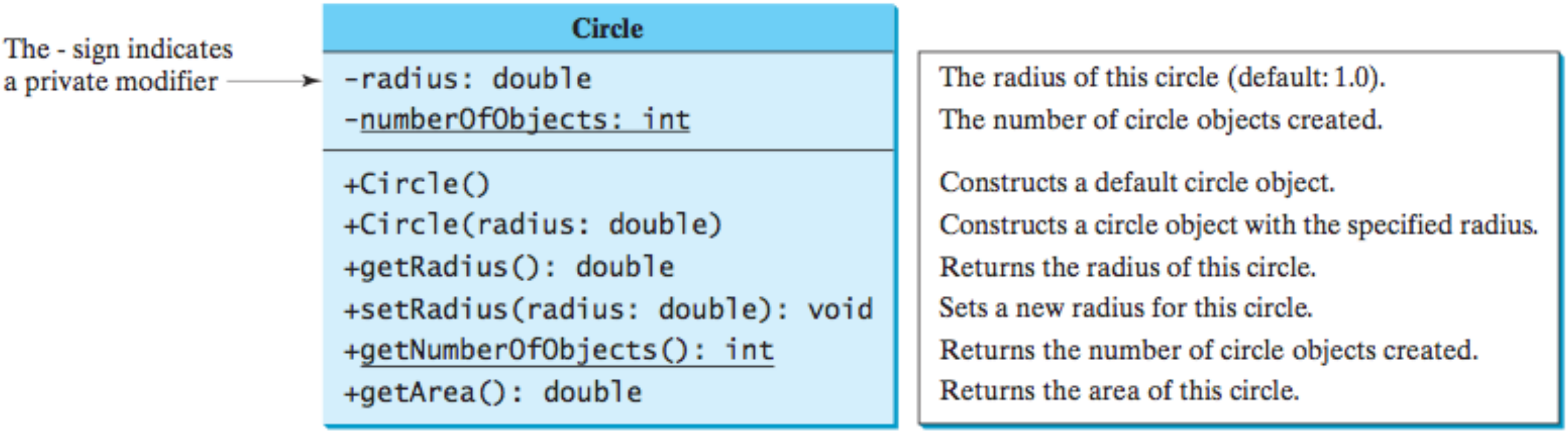
    `public void setLength(double len) …`
  - `setWidth` : Sets the value of the `width` field.

    `public void setLength(double w) …`
  - `getLength` : Returns the value of the `length` field.

    `public double getLength() …`
  - `getWidth` : Returns the value of the `width` field.

    `public double getWidth() …`

Example of Data Field Encapsulation



The - sign indicates a private modifier

Circle

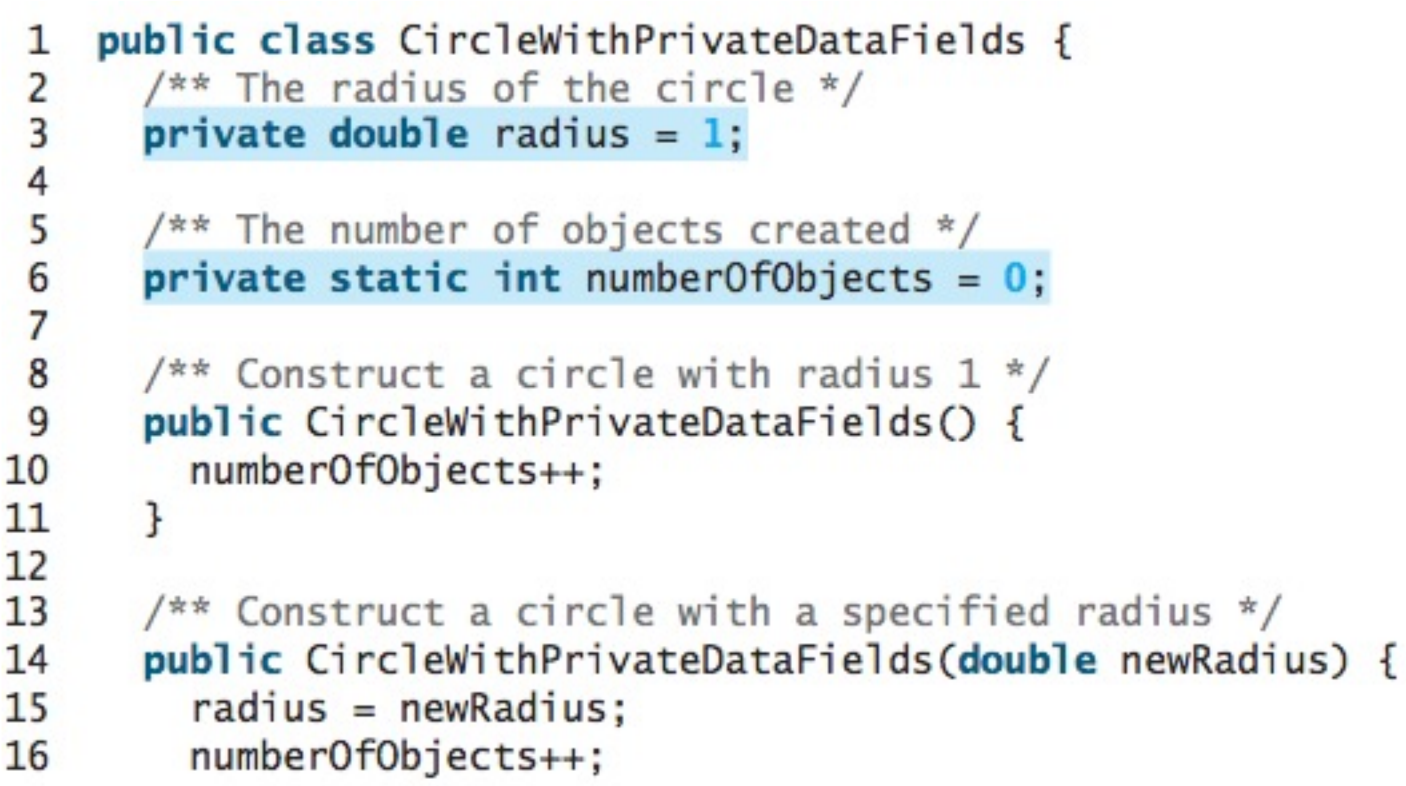
-radius: double
-numberOfObjects: int

+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getNumberOfObjects(): int
+getArea(): double

The radius of this circle (default: 1.0).
The number of circle objects created.

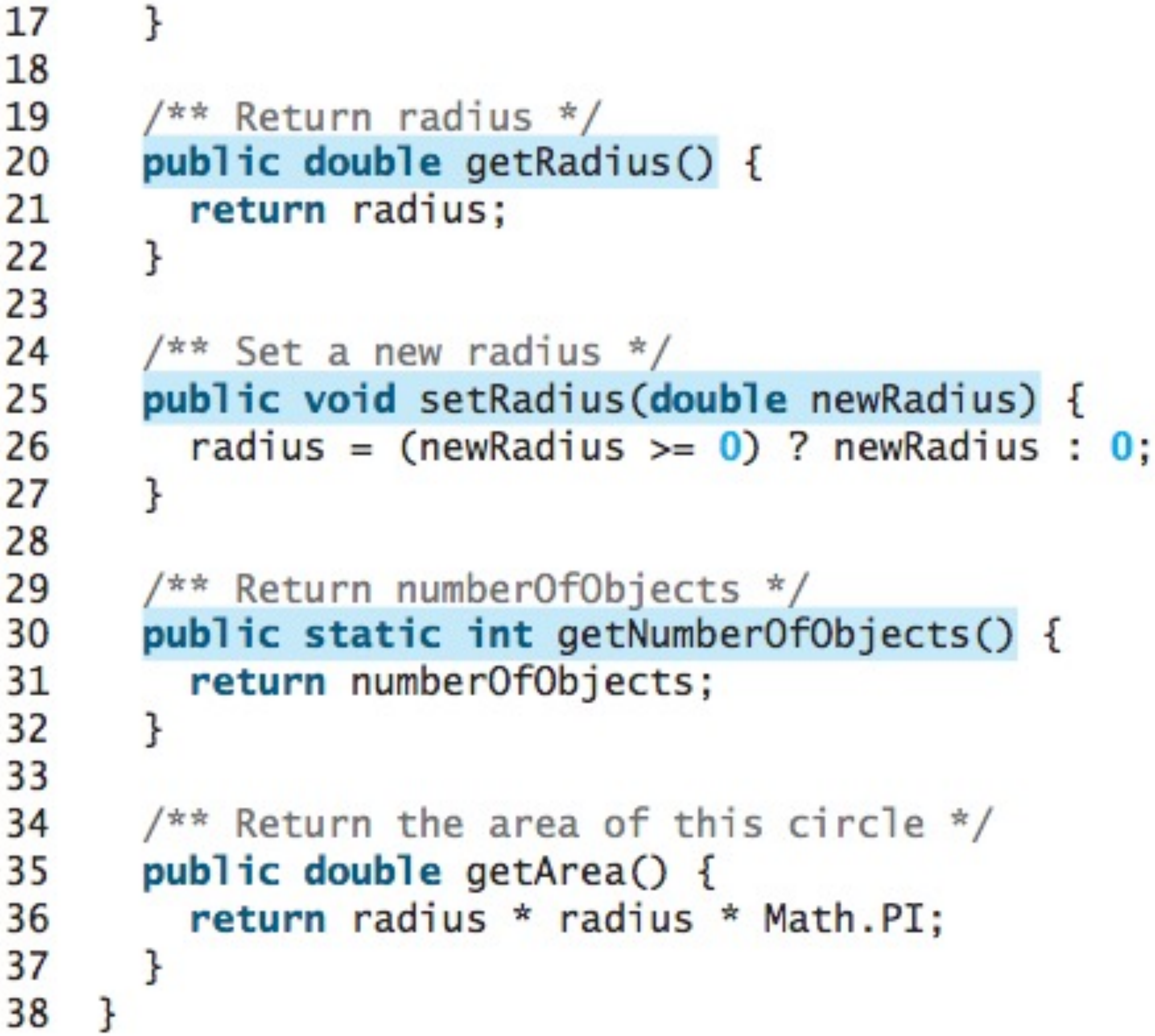
Constructs a default circle object.
Constructs a circle object with the specified radius.
Returns the radius of this circle.
Sets a new radius for this circle.
Returns the number of circle objects created.
Returns the area of this circle.

Example of Data Field Encapsulation



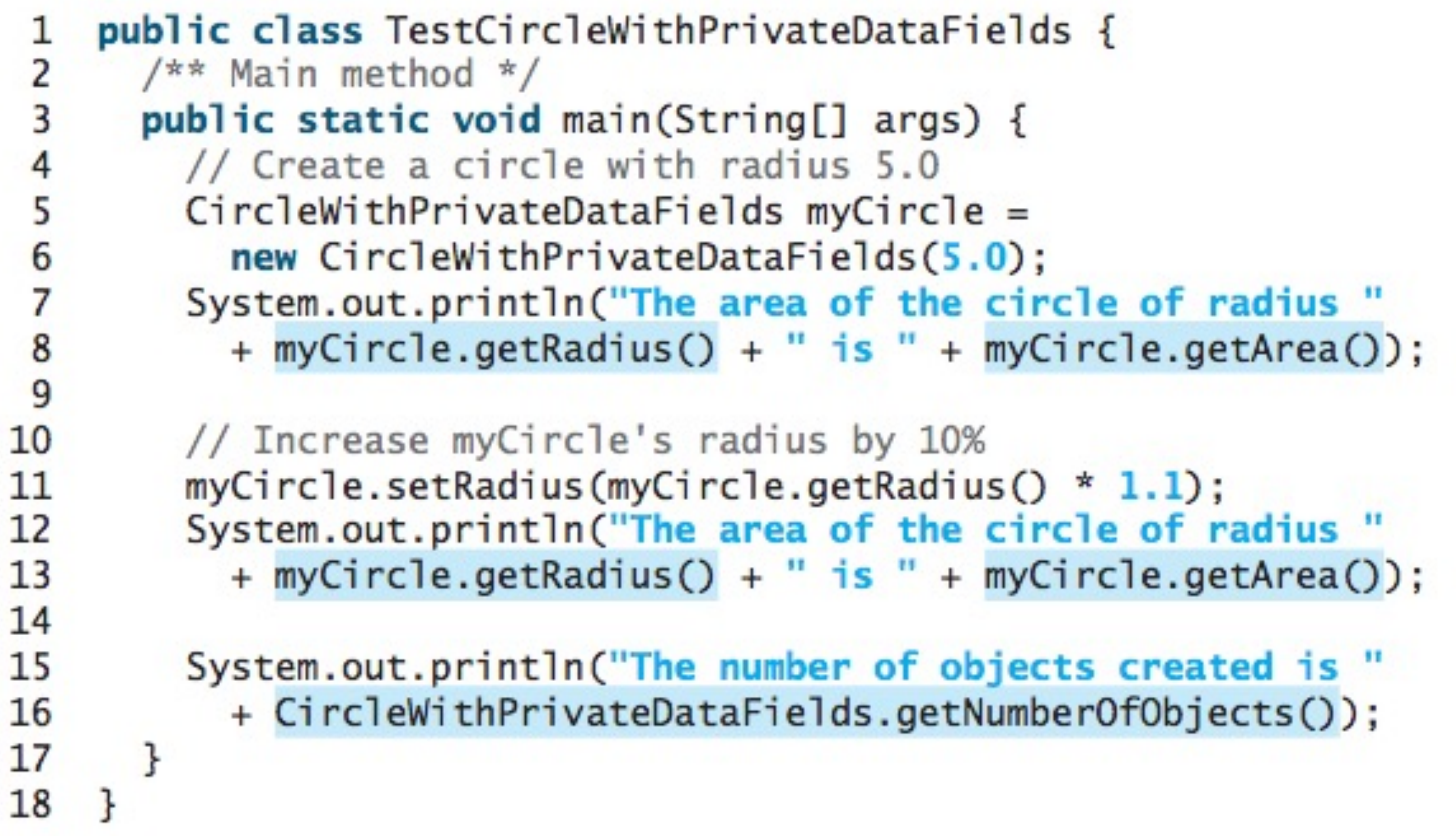
```
public class CircleWithPrivateDataFields {
  /** The radius of the circle */
  private double radius = 1;

  /** The number of objects created */
  private static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  public CircleWithPrivateDataFields() {
    numberOfObjects++;
  }

  /** Construct a circle with a specified radius */
  public CircleWithPrivateDataFields(double newRadius) {
    radius = newRadius;
    numberOfObjects++;
```

Example of Data Field Encapsulation



```java
  }

  /** Return radius */
  public double getRadius() {
    return radius;
  }

  /** Set a new radius */
  public void setRadius(double newRadius) {
    radius = (newRadius >= 0) ? newRadius : 0;
  }

  /** Return numberOfObjects */
  public static int getNumberOfObjects() {
    return numberOfObjects;
  }

  /** Return the area of this circle */
  public double getArea() {
    return radius * radius * Math.PI;
  }
}
```

Example of Data Field Encapsulation



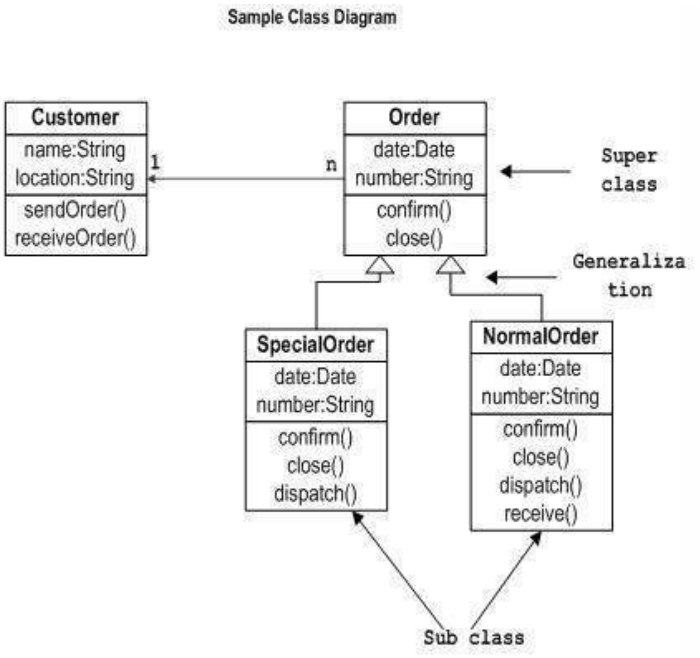
```java
public class TestCircleWithPrivateDataFields {
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 5.0
    CircleWithPrivateDataFields myCircle =
      new CircleWithPrivateDataFields(5.0);
    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    // Increase myCircle's radius by 10%
    myCircle.setRadius(myCircle.getRadius() * 1.1);
    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    System.out.println("The number of objects created is "
      + CircleWithPrivateDataFields.getNumberOfObjects());
  }
}
```

UML Class Diagram

- A UML (Unified Modelling Language) class diagram is a graphical tool that can aid in the design of a class.

- The diagram has three main sections.

Sample Class Diagram



Class Name
Attributes
Methods

UML diagrams are easily converted to Java class files. There will be more about UML diagrams a little later.

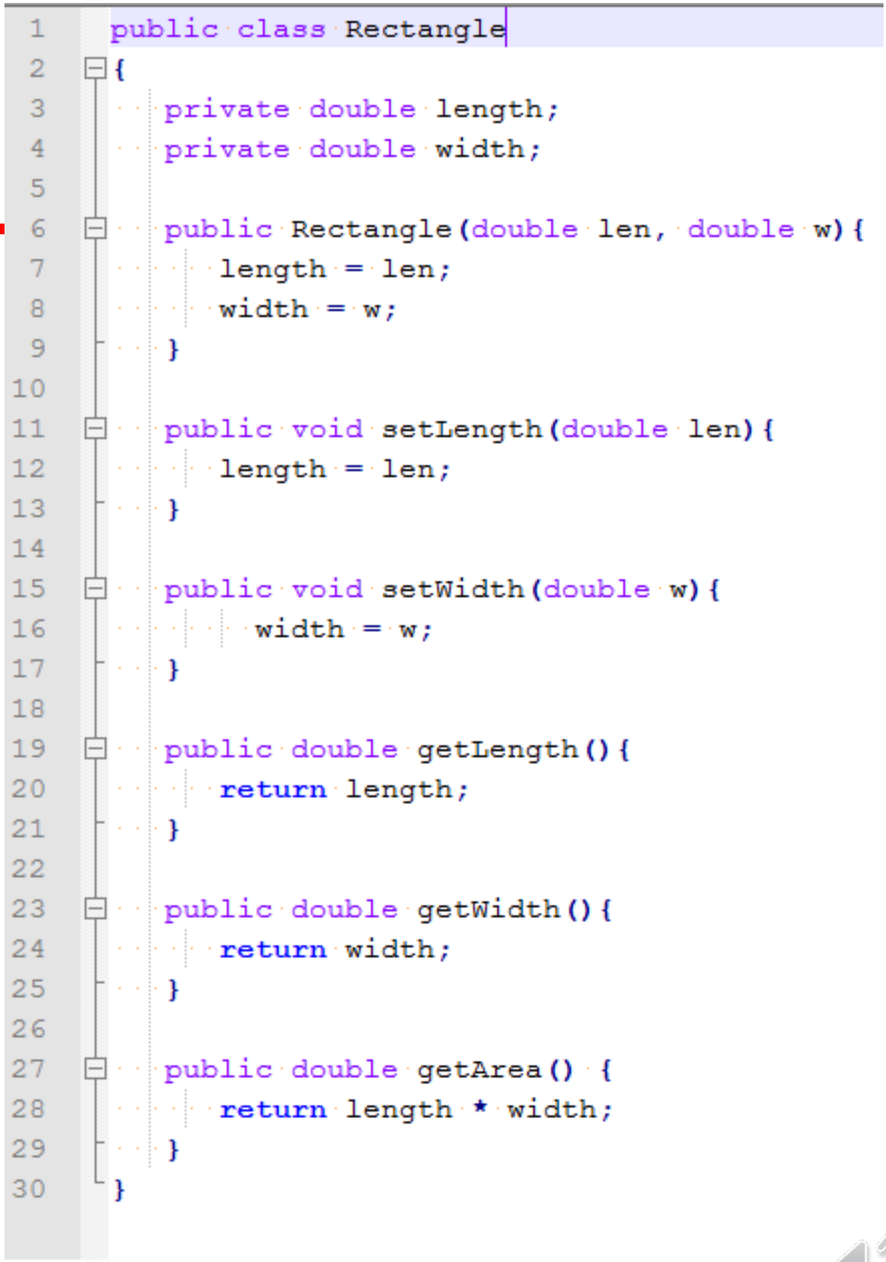
example

Rectangle

- width : double
- length : double

+Rectangle(len:double, w:double)
+ setLength(len : double): void
+ setWidth(w : double) : void
+ getLength() : double
+ getWidth() : double
+ getArea() : double

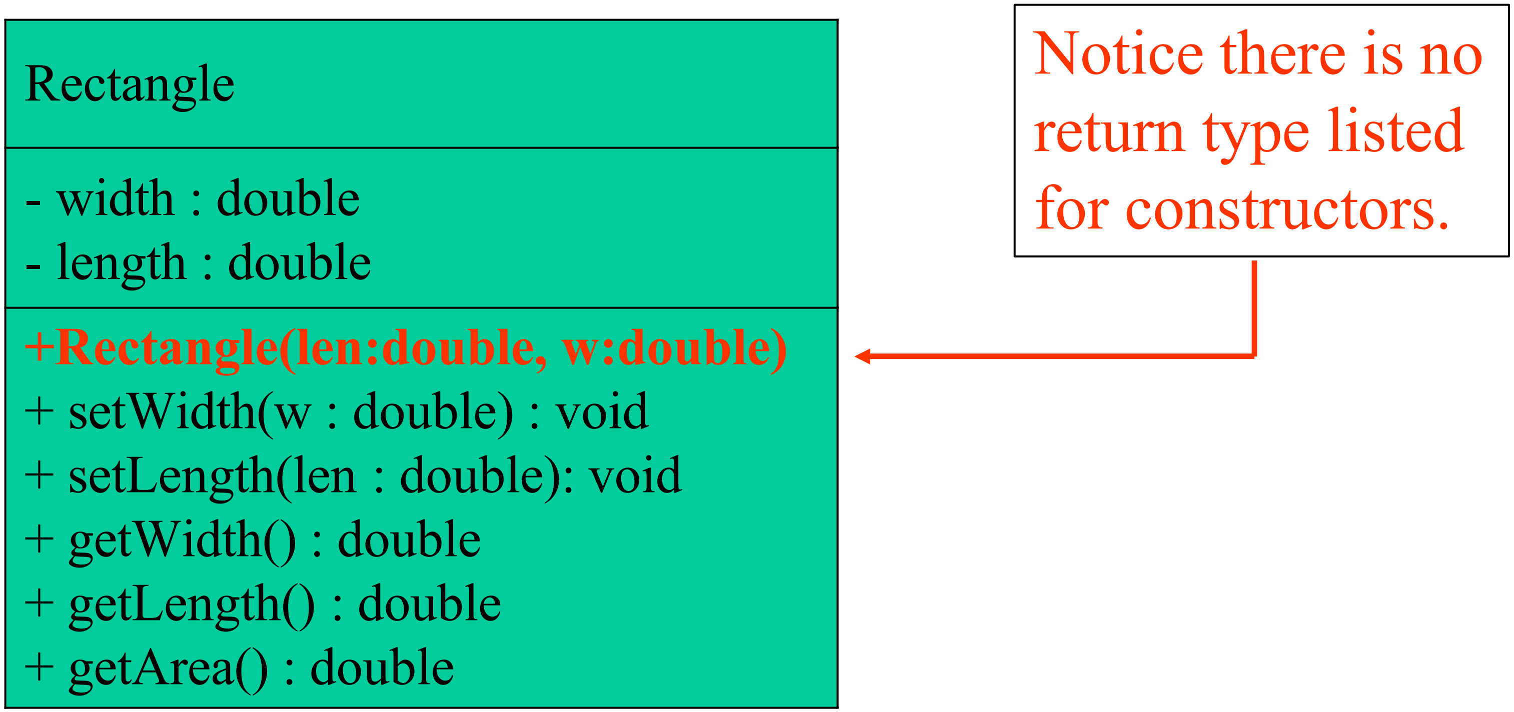
Example: Rectangle.java



```
public class Rectangle
{
    private double length;
    private double width;

    public Rectangle(double len, double w){
        length = len;
        width = w;
    }

    public void setLength(double len){
        length = len;
    }

    public void setWidth(double w){
          width = w;
    }

    public double getLength(){
        return length;
    }

    public double getWidth(){
        return width;
    }

    public double getArea() {
        return length * width;
    }
}
```

Constructors in UML

- In UML, the most common way constructors are defined is:



Rectangle

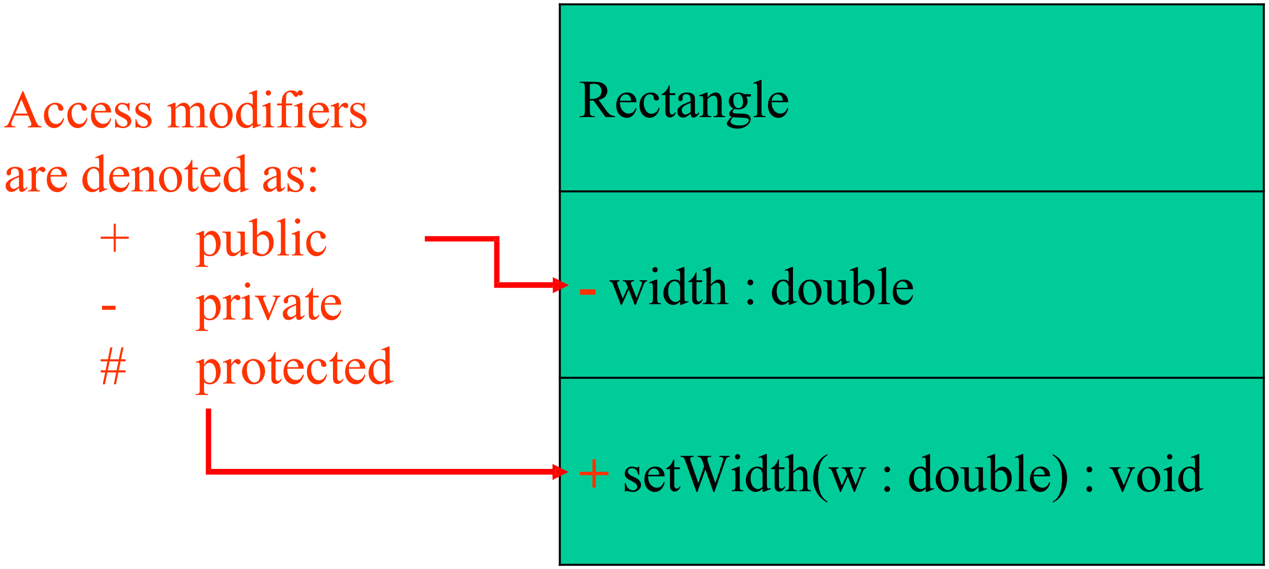
- width : double
- length : double

+Rectangle(len:double, w:double)
+ setWidth(w : double) : void
+ setLength(len : double): void
+ getWidth() : double
+ getLength() : double
+ getArea() : double

Notice there is no return type listed for constructors.

UML Data Type and Parameter Notation

- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.



Access modifiers are denoted as:

+	public
-	private
#	protected

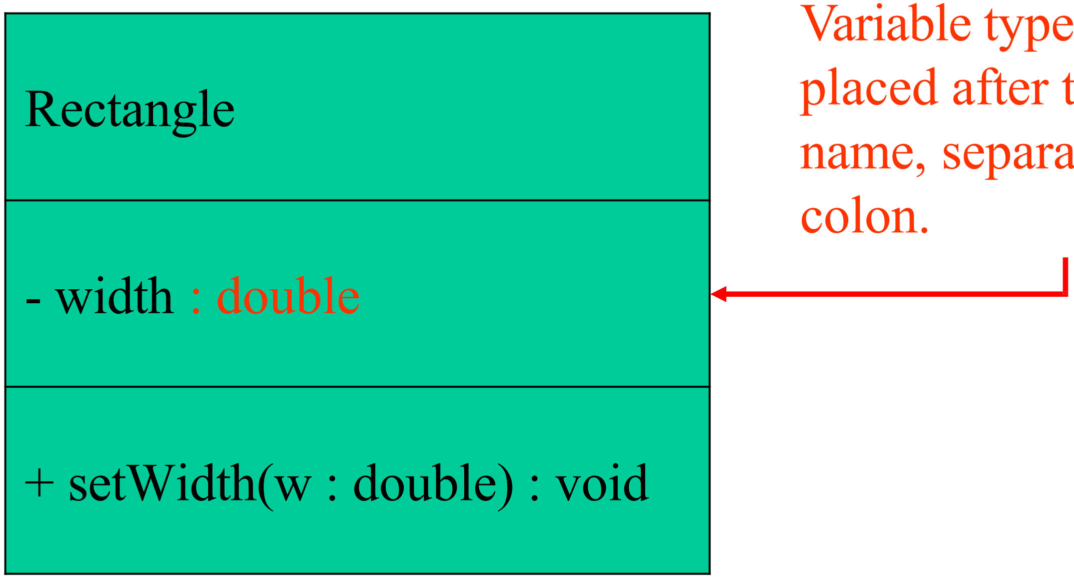
Rectangle

- width : double

+ setWidth(w : double) : void

UML Data Type and Parameter Notation

- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.



Rectangle

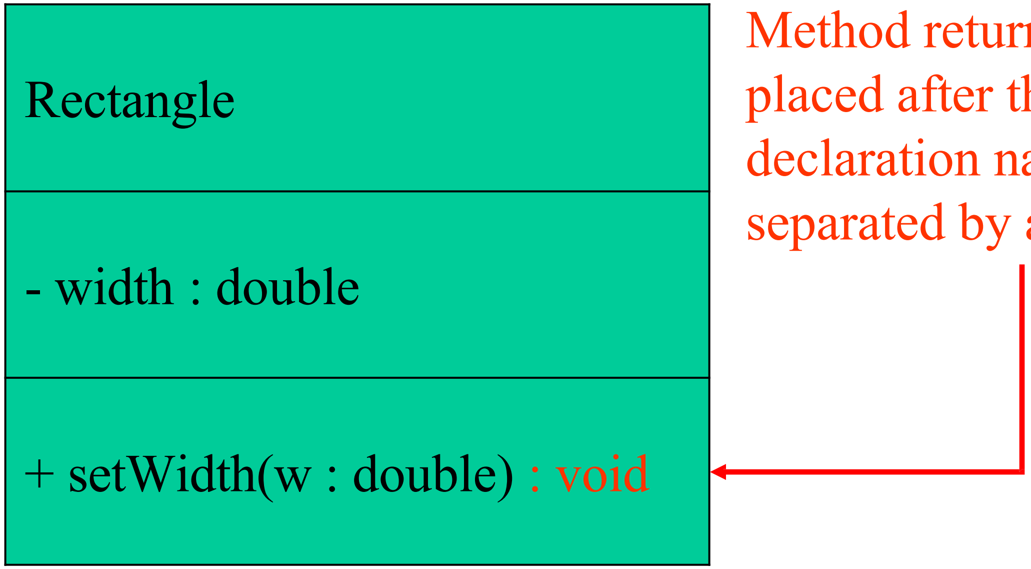
- width : double

+ setWidth(w : double) : void

Variable types are placed after the variable name, separated by a colon.

UML Data Type and Parameter Notation

- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.



Rectangle

- width : double

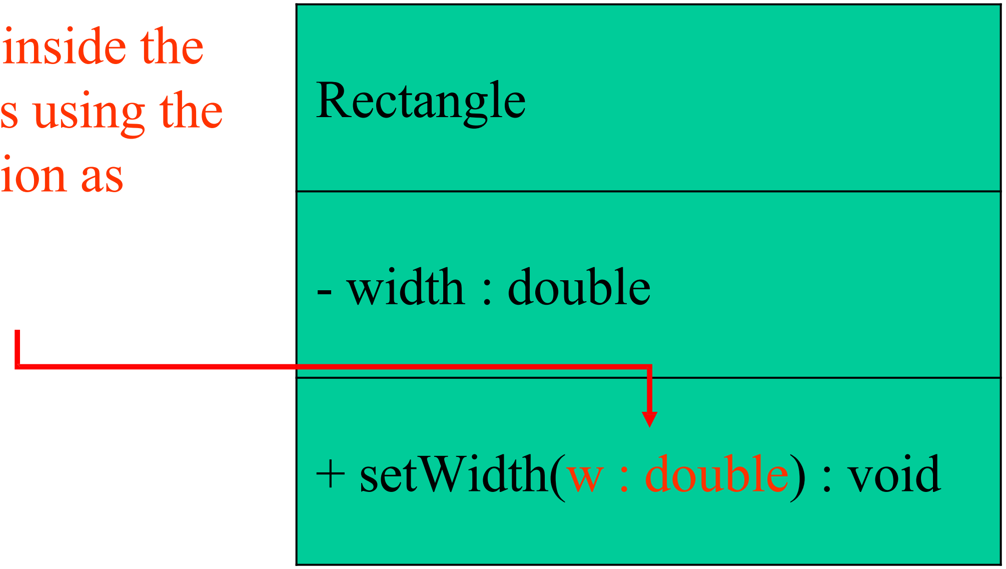
+ setWidth(w : double) : void

Method return types are placed after the method declaration name, separated by a colon.

UML Data Type and Parameter Notation

- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.

Method parameters are shown inside the parentheses using the same notation as variables.



Rectangle

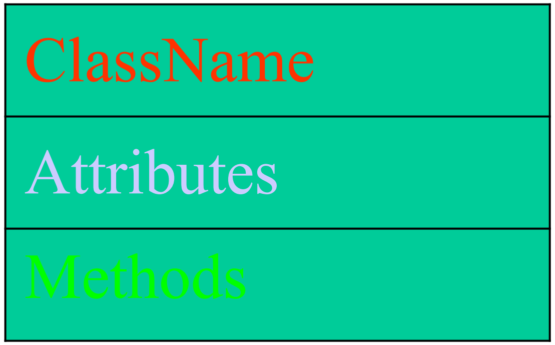
- width : double

+ setWidth(w : double) : void

Converting the UML Diagram to Code

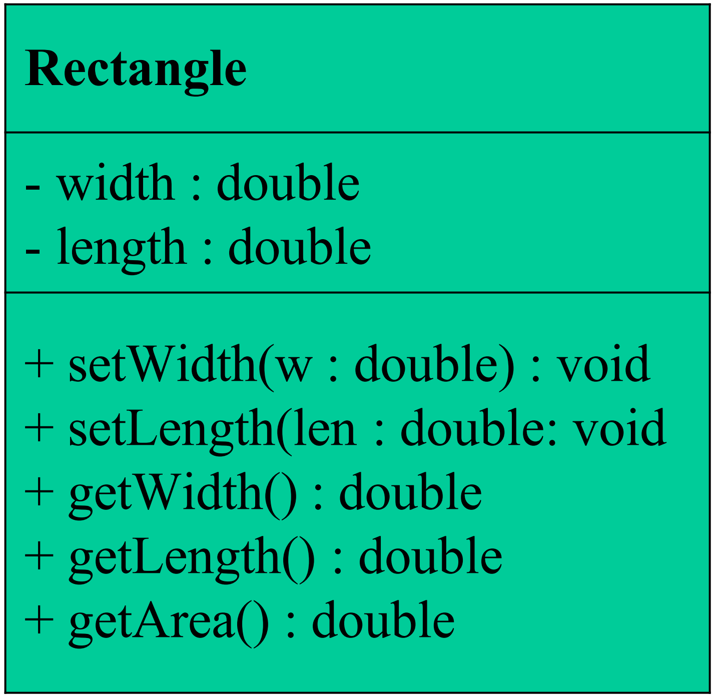
- Putting all of this information together, a Java class file can be built easily using the UML diagram.
- The UML diagram parts match the Java class file structure.

```
class header
{
    Attributes
    Methods
}
```



Converting the UML Diagram to Code

Once the class structure has been tested, the method bodies can be written and tested.



Rectangle

- width : double
- length : double

+ setWidth(w : double) : void
+ setLength(len : double: void
+ getWidth() : double
+ getLength() : double
+ getArea() : double

```
public class Rectangle
{
   private double width;
   private double length;

   public void setWidth(double w)
   {    width = w;
   }
   public void setLength(double len)
   {    length = len;
   }
   public double getWidth()
   {    return width;
   }
   public double getLength()
   {    return length;
   }
   public double getArea()
   {
   }
}
```

Class Layout Conventions

- The layout of a source code file can vary by employer or instructor.
- Typically the layout is generally:
  - Attributes are typically listed first
  - Methods are typically listed second
    - The main method is sometimes first, sometimes last.
    - Accessors and mutators are typically grouped.

A Driver Class

- An application in Java is a collection of classes.

- A *driver class* is a class which contains `main` method.

A Driver Program

Rectangle has no main method.

```
public class RectangleDemo
{
    public static void main(String[] args)
  {
  Rectangle r = new Rectangle();

  r.setWidth(10);

  r.setLength(10);

  System.out.println("Width = "
                 + r.getWidth());

     System.out.println("Length = "
                 + r.getLength());

     System.out.println("Area = "
                 + r.getArea());
  }
}
```

```
public class Rectangle
{
   private double width;
   private double length;

   public void setWidth(double w)
   {     width = w;
   }
   public void setLength(double len)
   {     length = len;
   }
   public double getWidth()
   {     return width;
   }
   public double getLength()
   {     return length;
   }
   public double getArea()
   {     return length * width;
   }
}
```

This RectangeDemo class is a Java application that uses the Rectangle class.
Another Example: LengthWidthDemo.java

the UNIVERSITY of GREENWICH

Summary

We have covered :

- Class Declaration
- Class Constructor, Data Field Encapsulation
- Class Design with UML, UML Diagram to Code