

# Programming 1

Week 10 – User-defined Classes

# Object Oriented Programming (OOP)

- So far, we learned procedure-oriented programming where the main emphasis is on solving a specific task. It consists of a series of computational steps to be carried out.
- Java is an object-oriented programming language (OOP).
- Object-oriented programming is one of the most effective approaches for writing software.
- OOP is about creating classes that represent real-world things and situations, and you create objects based on these classes.

# Object Oriented Programming (OOP)

- OOP is not different from what we have learned so far, it is just another way to program. We still need to declare variables, and use the control structures, etc.,
- Understanding OOP will help you know your code, not just what's happening line by line, but also the bigger concepts behind it.
- Knowing the logic behind classes will train you to think logically so you can write programs that effectively address almost any problem you encounter.

# Why OOP

- OOP provides a clear structure for the programs.
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug.
- OOP makes it possible to create full reusable applications with less code and shorter development time.

# What Is an Object?

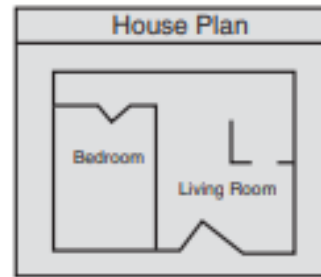
- Object is a software model of something that exists in the real world.
  - Software objects are often used to model the real-world objects that you find in everyday life.
  - Real-world objects share two characteristics: They all have state and behavior.
  - Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.
- Objects have two general capabilities:
  - Objects can store data. The pieces of data stored in an object are known as **fields**.
  - Objects can perform operations. The operations that an object can perform are known as **methods**.

# What Is a Class?

- A class is a blueprint or prototype from which objects are created.
  - *In the real world, you'll often find many individual objects all of the same kind.*
  - *There may be thousands of other bicycles in existence, all of the same make and model.*
  - *Each bicycle was built from the same set of blueprints and therefore contains the same components.*
  - *In object-oriented terms, we say that your bicycle is an instance of the class of objects known as bicycles.*
- A class is code that describes a particular type of object. It specifies the data that an object can hold (the object's fields), and the actions that an object can perform (the object's methods).

# A blueprint and houses built from the blueprint

Blueprint that describes a house



Instances of the house described by the blueprint



# Declaration of Instance Variables

- Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables.
  - Thus, the data for one object is separate and unique from the data for another.

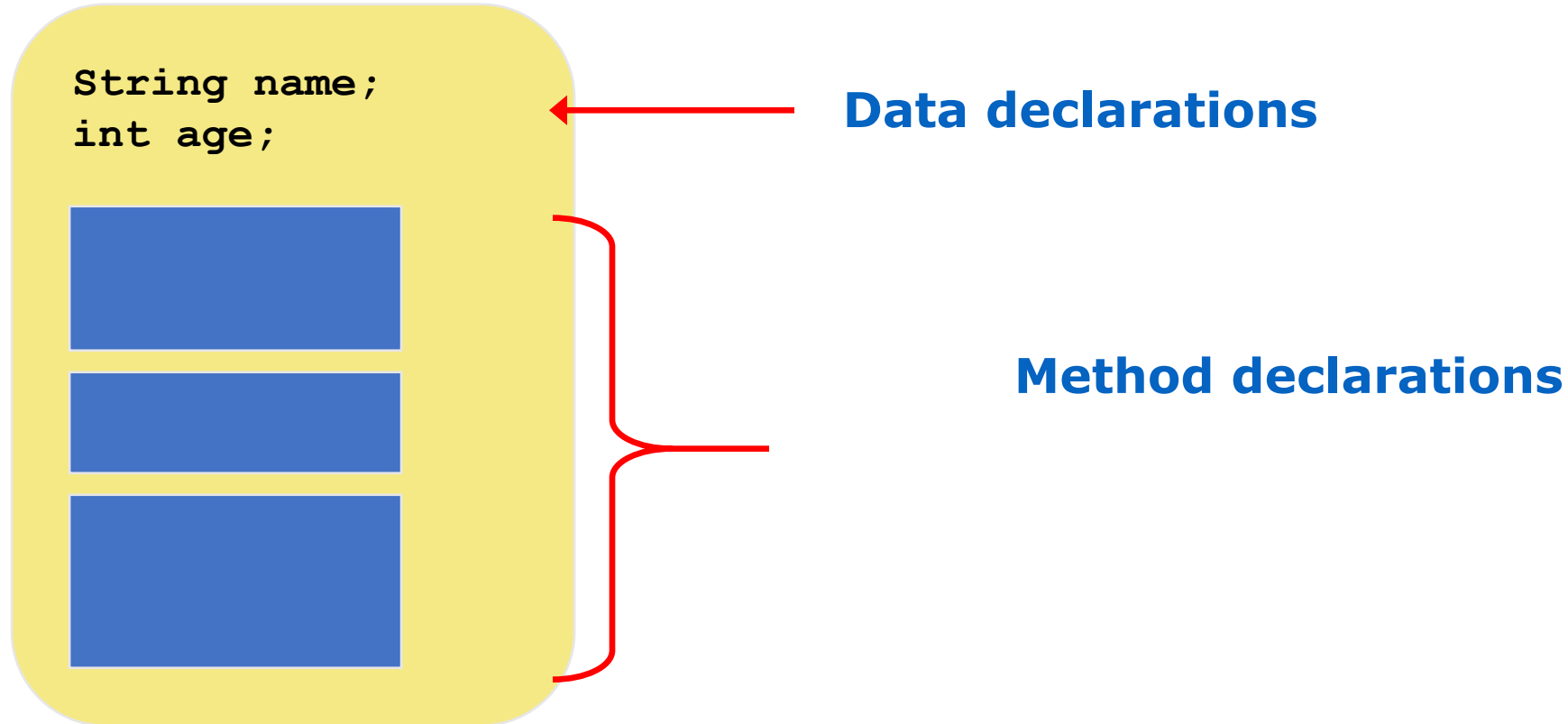


# Declaration of Methods

- *A method is a program module that contains a series of statements that carry out a task.*
- *To execute a method, you invoke or call it from another method; the calling method makes a method call, which invokes the called method.*
  - *Any class can contain an unlimited number of methods, and each method can be called an unlimited number of times.*
- Class methods are called **instance methods**.
- Instance methods are methods that are **not** declared with a special keyword, `static`.

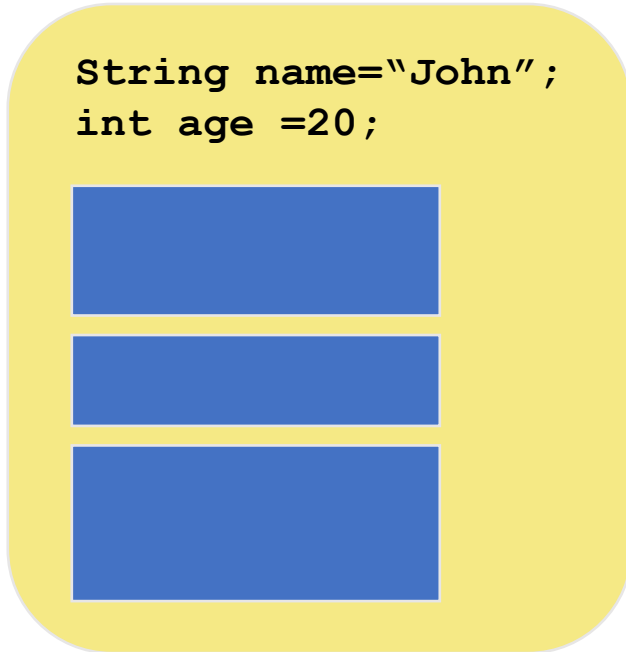
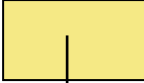
# Classes

- A class can contain data declarations and method declarations

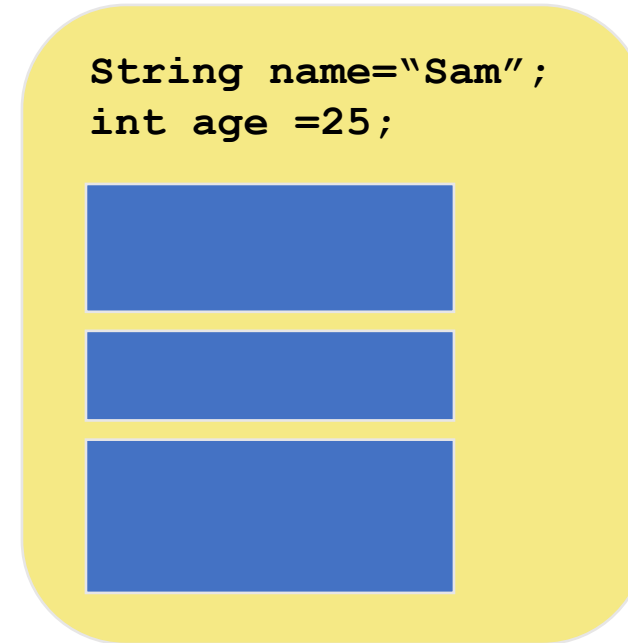
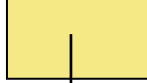


# Objects – instances of classes

obj1



obj2



Note that the variables can have different values in the two objects

# A first look at classes

**A class will look like this:**

```
<Access-Modifier> class MyClass {  
    // field, constructor, and method declarations  
}
```

**To instantiate an object, we will do:**

```
MyClass instance = new MyClass(<constructor params>);
```

# Access Modifiers

- An access modifier is a Java keyword that indicates how a field or method can be accessed.
- `public`
  - When the `public` access specifier is applied to a class member, the member can be accessed by code inside the class or outside.
- `private`
  - When the `private` access specifier is applied to a class member, the member cannot be accessed by code outside the class. The member can be accessed only by methods that are members of the same class.

# Demo

# Data Hiding (1 of 2)

- An object hides its internal, private fields from code that is outside the class that the object is an instance of.
- Only the class's methods may directly access and make changes to the object's internal data.
- Code outside the class must use the class's public methods to operate on an object's private fields.

## Data Hiding (2 of 2)

- Data hiding is important because classes are typically used as components in large software systems, involving a team of programmers.
- Data hiding helps enforce the integrity of an object's internal data.



# Accessor and Mutator Methods

- Because of the concept of data hiding, fields in a class are private.
- **An *accessor* method** is a special kind of method that returns a value - it returns the value of an *instance variable* of the object for which it is called
- Accessor methods are commonly called “get” methods because they allow us to access or “get” the value of an instance variable
- Unlike *accessor* methods, which simply *return* the value of an instance variable, ***mutator* methods** allow us to *change* the value(s) of one or more of the instance variables of the object
- Since mutator methods *modify* or *change* values, they are commonly called “set” methods
- Mutator methods do not return a value
- Other names for these methods are **getters** and **setters**.

# Accessor and Mutator Methods

- Each field that the programmer wishes to be viewed by other classes needs an accessor.
  - Can declare a write-only properties by removing the getter method.
- Each field that the programmer wishes to be modified by other classes needs a mutator.
  - Can declare read-only properties by removing the setter method.
- For example, the names of the methods `getScore()` and `setScore(value)` are obtained by concatenating "get" or "set" with the name of the property score with the first character capitalized
- This naming convention is easily recognizable, both to human programmers and to editing tools, such as the NetBeans IDE.

# Class Layout Conventions

- The layout of a source code file can vary:
- A common layout is:
  - Fields listed first
  - Methods listed second
    - Accessors and mutators are typically grouped.

# Demo

1. Create a “Car” class with . . .
  - Instance variables “name” & “mpg (Miles Per Gallon)” (OK to make public for now)
  - No need for a main() method!
2. In a separate file, create a “CompareCars” class with a main() method
  - Create 2 instances of “Car” using ‘new’
  - Set the name & mpg for each (e.g. Civic with 35 mpg and Mazda with 15 mpg)
  - Since your instance variables are public, you can set them directly using “.name” and “.mpg”
3. Compare the fuel efficiency of your 2 cars and print out which is more efficient to the screen
  - e.g. “The Civic is more fuel efficient than the F350”
4. Now change your variables from public to private – Try to run it again and notice the compiler errors – Fix these errors by adding ‘get’ methods (getName & getMpg) to your Car class that return these variables & modify your “CompareCars” class to call these methods – This is how we “hide data” – a key component of the encapsulation concept

# Providing Constructors for Your Classes

- *A class contains constructors that are invoked to create objects from the class blueprint.*
- *Constructor declarations look like method declarations—except that they use the name of the class and have no return type.*
  - *You don't have to provide any constructors for your class, but you must be careful when doing this.*
  - *The compiler automatically provides a no-argument, default constructor for any class without constructors.*

# Constructors (1 of 2)

- A constructor is a method that is **automatically** called when an object is created.
- Constructors are used to perform operations at the time an object is created.
- Constructors typically initialize instance fields and perform other object initialization tasks.
  - We use constructors to initialize the object with the default or initial state

# Constructors (2 of 2)

- 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.
- **Default Constructor**
  - If you do not write a constructor, Java provides one when the class is compiled. The constructor that Java provides is known as the **default constructor**.
  - It sets all of the object's numeric fields to 0.
  - It sets all of the object's `boolean` fields to `false`.
  - It sets all of the object's reference variables to the special value **null**

# Writing a Class, Step by Step (1 of 2)

- A `Rectangle` object will have the following fields:
  - `length`. The length field will hold the rectangle's length.
  - `width`. The width field will hold the rectangle's width.

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



## Writing a Class, Step by Step (2 of 2)

- The `Rectangle` class will also have the following methods:
  - **`setLength`**. The `setLength` method will store a value in an object's `length` field.
  - **`setWidth`**. The `setWidth` method will store a value in an object's `width` field.
  - **`getLength`**. The `getLength` method will return the value in an object's `length` field.
  - **`getWidth`**. The `getWidth` method will return the value in an object's `width` field.
  - **`getArea`**. The `getArea` method will return the area of the rectangle, which is the result of the object's `length` multiplied by its `width`.

# Writing and Demonstrating the `setLength` Method

```
/**  
    The setLength method stores a value in the  
    length field.  
    @param len The value to store in length.  
*/  
public void setLength(double len)  
{  
    length = len;  
}
```

# Writing the `getLength` Method

```
/**  
    The getLength method returns a Rectangle  
    object's length.  
    @return The value in the length field.  
*/  
public double getLength()  
{  
    return length;  
}
```

Similarly, the `setWidth` and `getWidth` methods can be created.

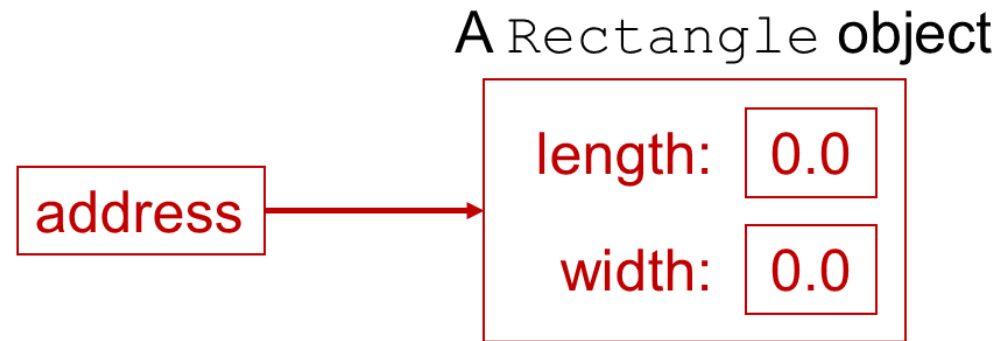
# Writing and Demonstrating the `getArea` Method

```
/**  
    The getArea method returns a Rectangle  
    object's area.  
    @return The product of length times width.  
*/  
public double getArea()  
{  
    return length * width;  
}
```

# Creating a Rectangle object

```
Rectangle box = new Rectangle();
```

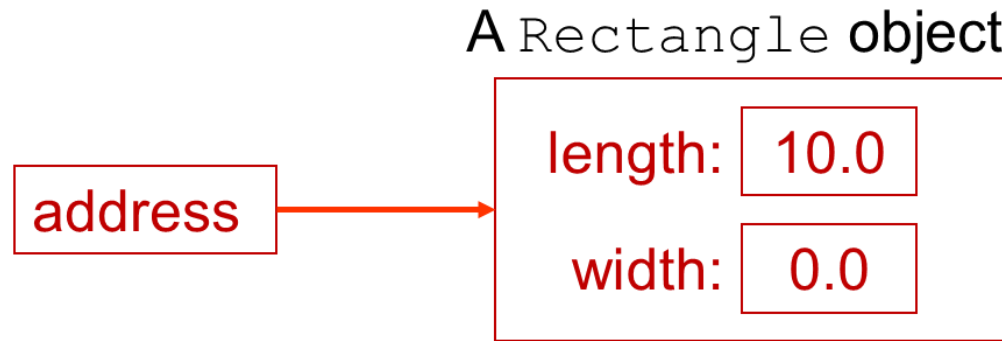
The `box` variable  
holds the address  
of the  
Rectangle  
object.



# Calling the `setLength` Method

```
box.setLength(10.0);
```

The `box` variable  
holds the address  
of the  
Rectangle  
object.



**This is the state of the `box` object after the `setLength` method executes.**

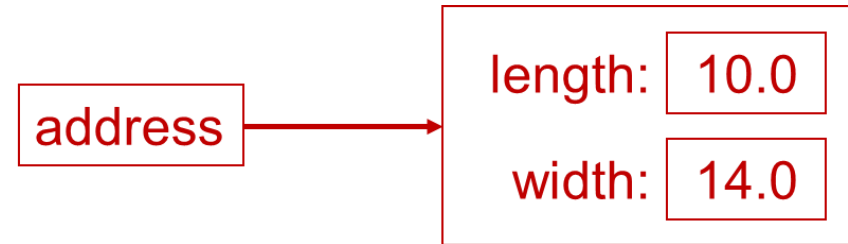
## Instance Fields and Methods (2 of 2)

- Instance fields and instance methods require an object to be created in order to be used.
- Note that each room represented in this example can have different dimensions.

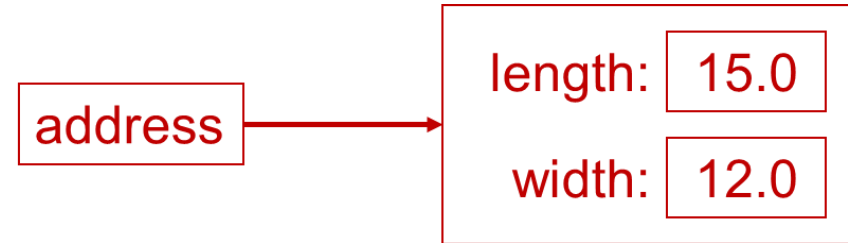
```
Rectangle kitchen = new Rectangle();  
Rectangle bedroom = new Rectangle();  
Rectangle den = new Rectangle();
```

# States of Three Different Rectangle Objects

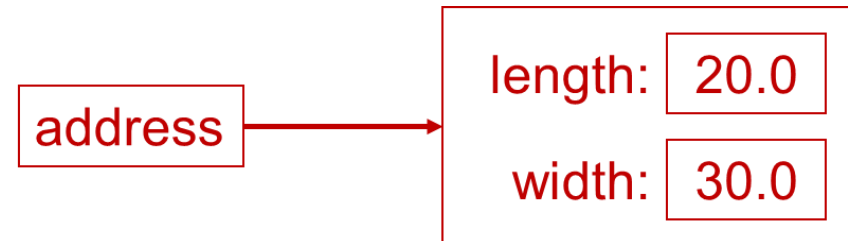
The `kitchen` variable holds the address of a Rectangle Object.



The `bedroom` variable holds the address of a Rectangle Object.



The `den` variable holds the address of a Rectangle Object.





# Constructor for Rectangle Class

```
/**
 * Constructor
 * @param len The length of the rectangle.
 * @param w The width of the rectangle.
 */
public Rectangle(double len, double w)
{
    length = len;
    width = w;
}
```

# No-Arg Constructor

- The default constructor (provided by Java) is a no-arg constructor.
- We can write our own no-arg constructor

```
public Rectangle()  
{  
    length = 1.0;  
    width = 1.0;  
}
```

# Uninitialized Local Reference Variables

- Reference variables can be declared without being initialized.

```
Rectangle box;
```

- This statement does not create a `Rectangle` object, so it is an uninitialized local reference variable.
- A local reference variable must reference an object before it can be used, otherwise a compiler error will occur.

```
box = new Rectangle(7.0, 14.0);
```

- `box` will now reference a `Rectangle` object of length 7.0 and width 14.0.

## Rectangle Class Constructor Overload

If we were to add the no-arg constructor we wrote previously to our `Rectangle` class in addition to the original constructor we wrote, what would happen when we execute the following calls?

```
Rectangle box1 = new Rectangle();  
Rectangle box2 = new Rectangle(5.0, 10.0);
```

The first call would use the no-arg constructor and `box1` would have a length of 1.0 and width of 1.0.

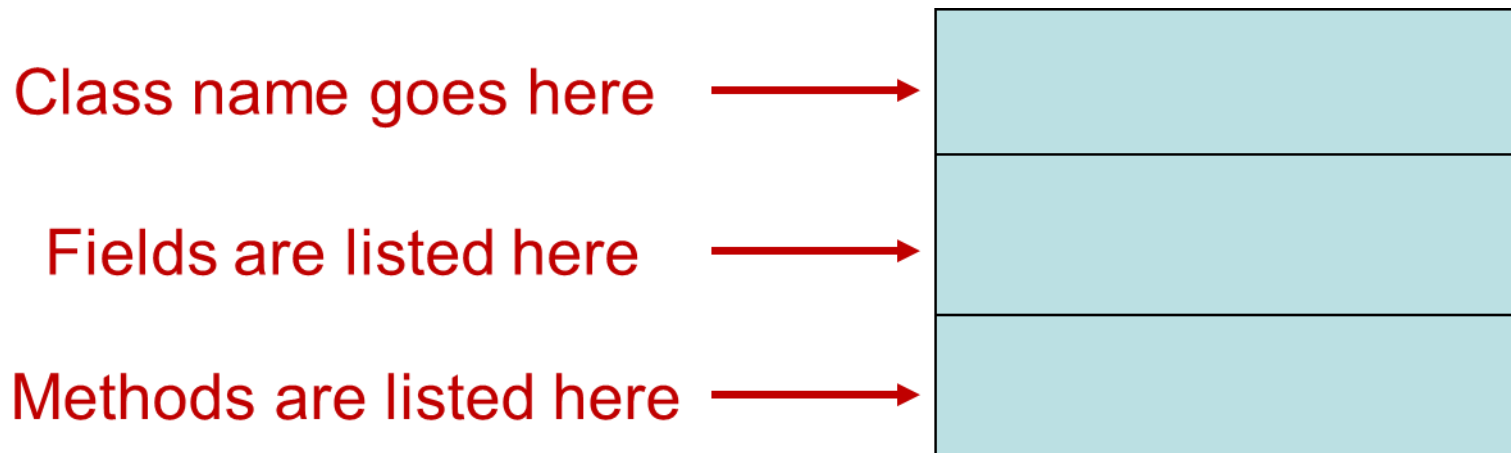
The second call would use the original constructor and `box2` would have a length of 5.0 and a width of 10.0.

# Scope of Instance Fields

- Variables declared as instance fields in a class can be accessed by any instance method in the same class as the field.
- If an instance field is declared with the `public` access specifier, it can also be accessed by code outside the class, as long as an instance of the class exists.

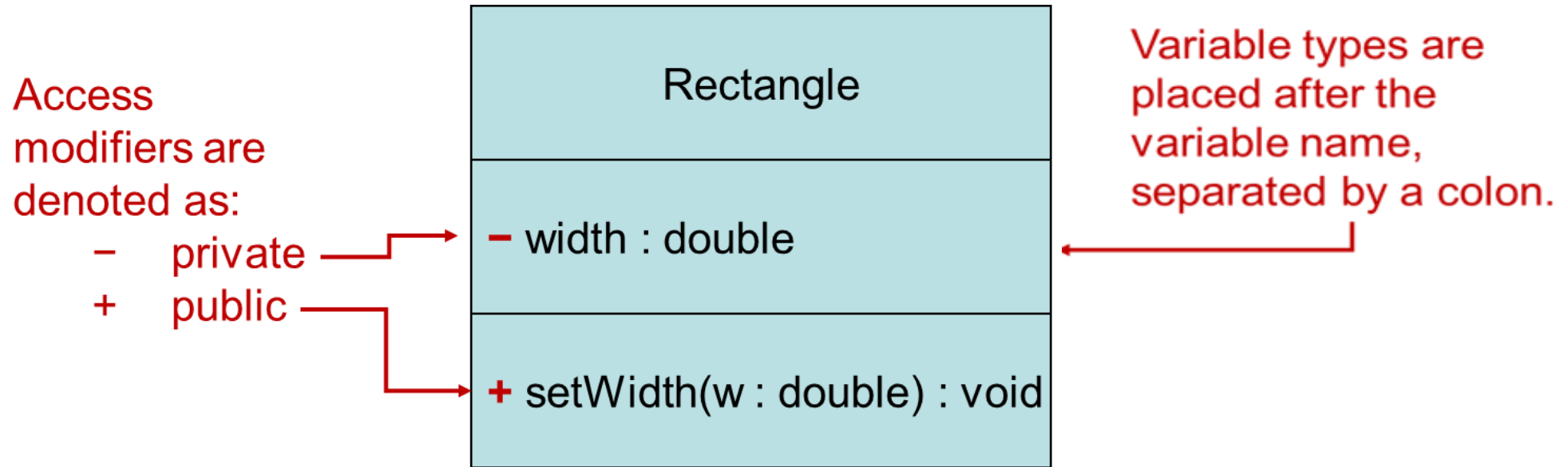
# UML Diagram

- Unified Modeling Language (UML) provides a set of standard diagrams for graphically depicting object-oriented systems.
- A class diagram is a diagram used in designing and modeling software
- Class diagrams enable us to model software in a high level of abstraction and without having to look at the source code.
- Classes in a class diagram correspond with classes in the source code.



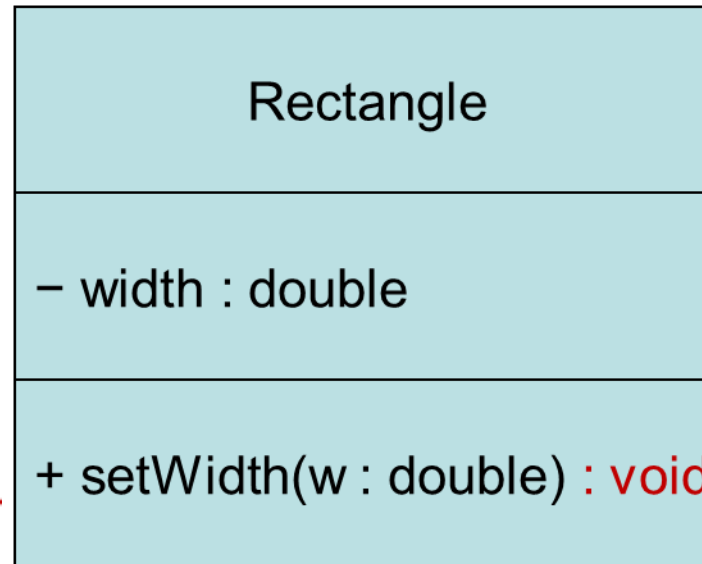
# UML Data Type and Parameter Notation

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



# UML Data Type and Parameter Notation

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

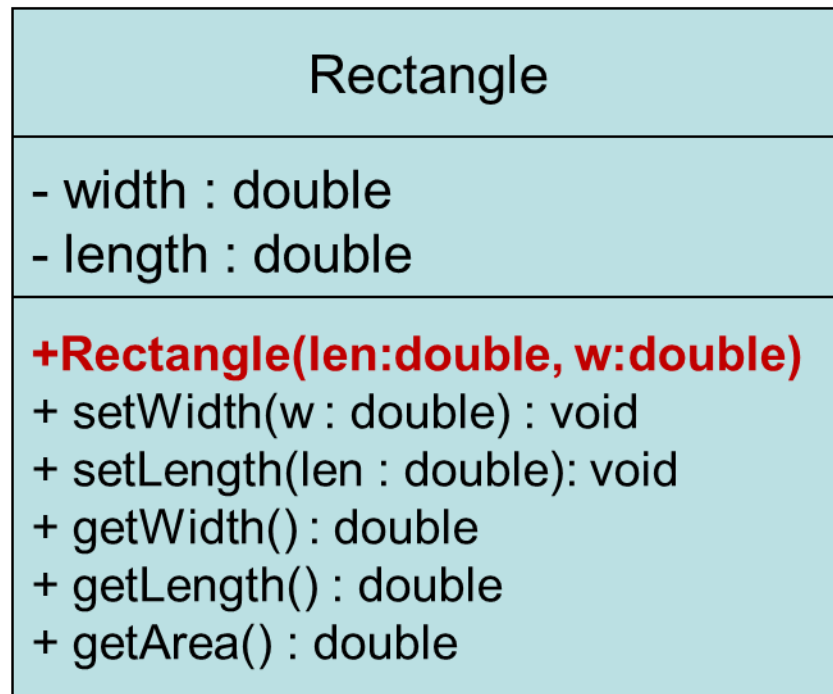


Method return types are placed after the method declaration name, separated by a colon.



# Constructors in UML

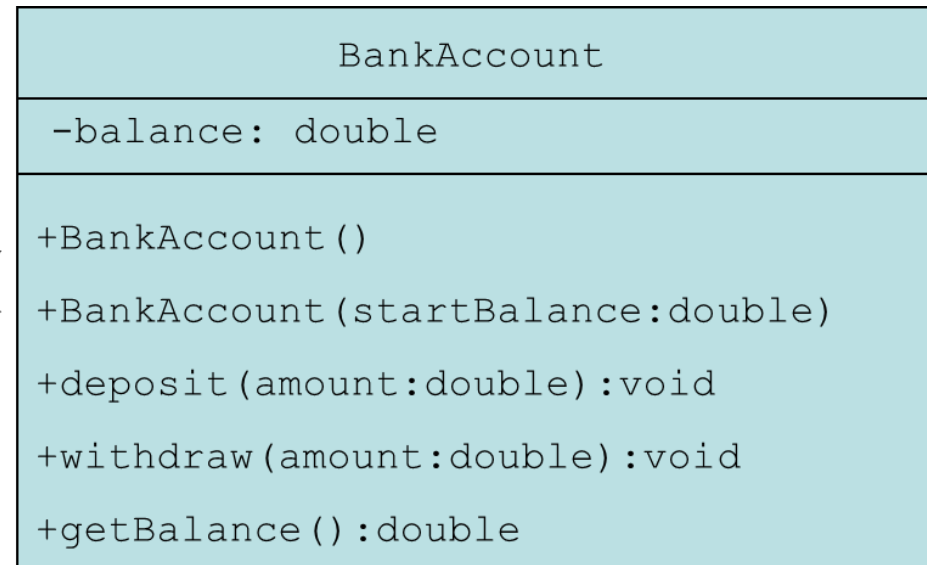
- In UML, the most common way constructors are defined is:



Notice there is no  
return type listed  
for constructors.

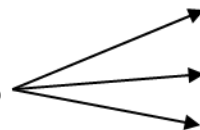
# The BankAccount Class Example

Overloaded Constructors



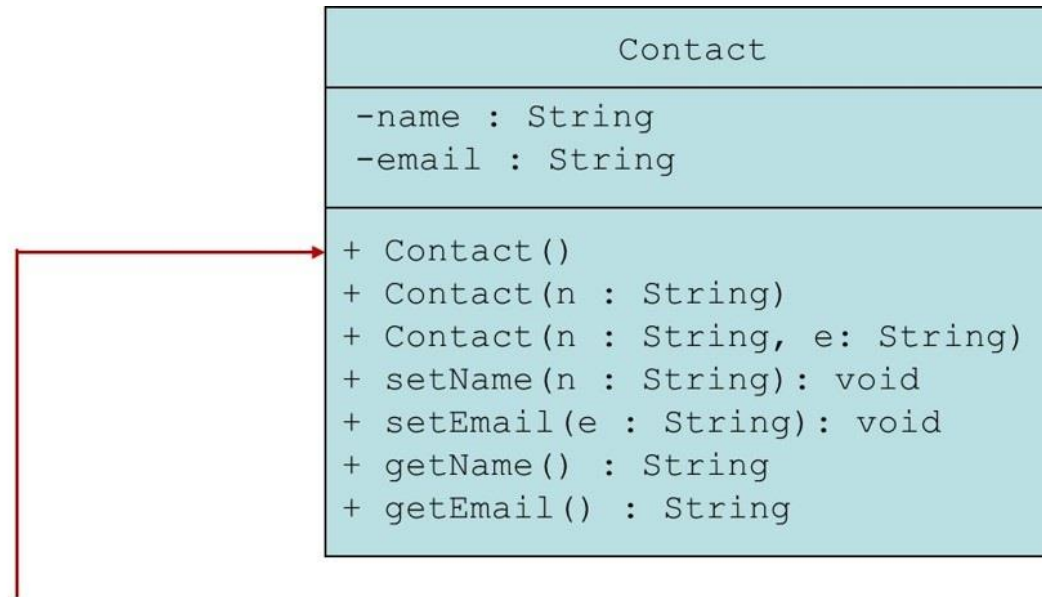
# The Contact Class Example (1 of 4)

Overloaded Constructors



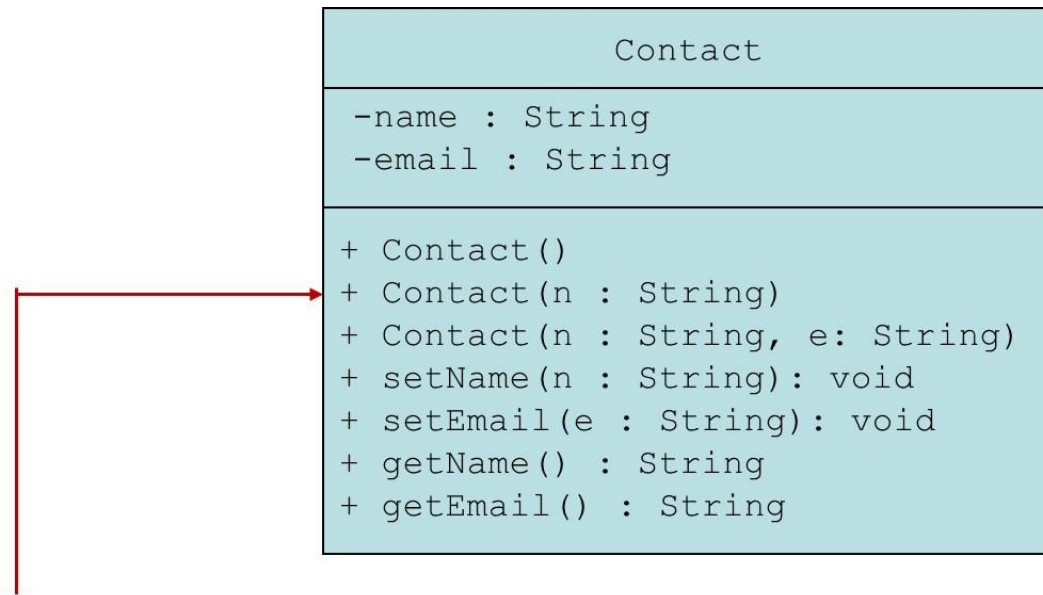
Contact
-name : String -email : String
+ Contact() + Contact(n : String) + Contact(n : String, e: String) + setName(n : String): void + setEmail(e : String): void + getName() : String + getEmail() : String

# The Contact Class Example (2 of 4)



The no-arg constructor can be used when we simply need a blank *Contact* object. Later we can use the *setName* and *setEmail* methods to insert a name and email address.

# The Contact Class Example (3 of 4)



The second constructor can be used when we have a contact's name, but no email address. Later, we can use the *setEmail* method to update the email address.

# Hands-on

Write a class named Employee that has the fields and methods presented in the UML:

Employee
<ul style="list-style-type: none"><li>- name : String</li><li>- idNumber : int</li><li>- department : String</li><li>- position : String</li></ul>
<ul style="list-style-type: none"><li>+ Employee()</li><li>+ Employee(n : String, id : int, dept : String, pos : String)</li><li>+ Employee(n : String, id : int)</li><li>+ setName(n : String) : void</li><li>+ setIdNumber(num : int) : void</li><li>+ setDepartment(d : String) : void</li><li>+ setPosition(p : String) : void</li><li>+ getName() : String</li><li>+ getIdNumber() : int</li><li>+ getDepartment() : String</li><li>+ getPosition() : String</li></ul>

# The toString() Method

- Once we create primitive variable, we can print it to check its value.  

```
int num = 3;  
System.out.print(num);
```
- However, we cannot directly print an object of a class. This will print an address instead of a value.  

```
Student s = new Student();  
System.out.print(s);
```
- To understand this, we need to first understand what happens when we create a primitive variable and an object. When you create a primitive variable, one piece of memory is allocated to it.

```
int num = 3;
```

memory	address	name
3	AA	num

When you want to read the value or write the value, it will directly check that piece of memory (address: AA) and get its value.

# The toString() Method

- When you create an object, it may contain more than one data members, (for example, a student contains name, age, email three data members)
- In this case, we cannot simply check one piece of the memory the get one value to represent an object.
- Most classes can benefit from having a method named toString, which is implicitly called under certain circumstances. Typically, the method returns a string that represents the state of an object.
- toString() method generates a string to represent the object, usually it returns the value of each data member of the object.

```
public String toString() {  
    return String.format("%02d:%02d:%02d", hr, mi, se);  
}
```