Revised: 12/23/2025

Object-Oriented Programming II

Michael C. Hackett

Associate Professor, Computer Science

COMMUNITY COLLEGE OF PHILADELPHIA

Lecture Topics

- Declaring Classes
- Fields
- Creating Instances of an Object
- Methods
- Class Diagrams
- Constructors
- Constructor Overloading
- Method Overloading
- Encapsulation

Colors/Fonts

Local Variable Names	– Brown
Primitive data types	– Fuchsia
Literals	– Blue
Keywords	– Orange
Object names	– Green
Operators/Punctuation	– Black
Field Names	– Lt Blue
Method Names	– Purple
Parameter Names	– Gold
Comments	– Gray
Package Names	– Pink

Source Code	– **`Consolas`**
Output	– `Courier New`

🟥 Boolean expression is false

🟩 Boolean expression is true

Class Declaration

- Objects are defined using a **class**.
  - The class contains the object’s fields and methods.

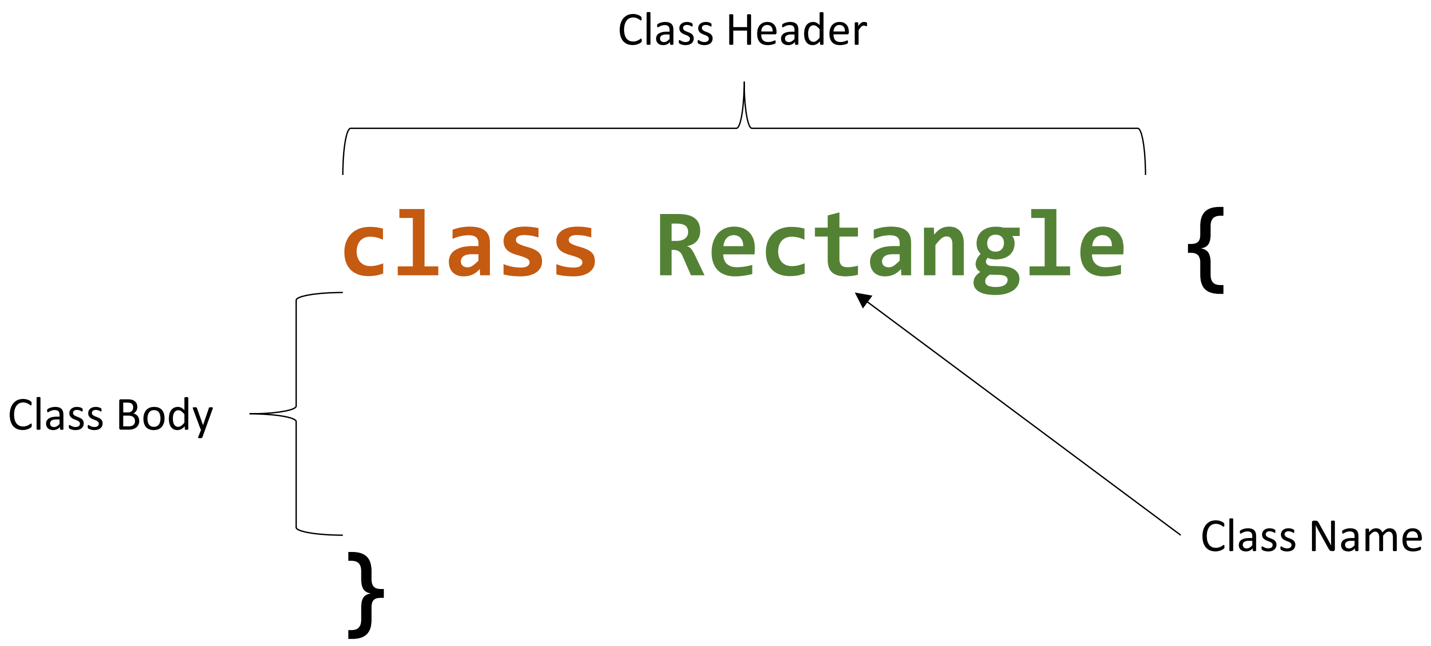
- A class is the blueprint from which unique instances of objects are created.
  - It is the actual source code of the object.
  - The object is the idea, the code in the class is the implementation of the idea.

Class Declaration

- Classes are declared using the class keyword followed by the desired name of the class.
  - Braces identify the beginning and end of the **class body**

```
class Rectangle {

}
```

Class Declaration



Class Header

```
class Rectangle {

}
```

Class Body

Class Name

Fields

- The fields and methods (*attributes and behaviors*) of a class are referred to as **class members**

- A ***field*** is simply a variable that stores data for the class.
  - Sometimes called an ***instance variable***.

- The data stored in a field is unique to each instance of an object.

Fields

```
class Rectangle {

   int width;
   int length;

}
```

- This Rectangle class now contains length and width fields, which may only reference integer data.
- The data stored in these fields is unique to each instance of an object.

Creating an Instance of an Object

- In a second class named TestProgram, we will declare a variable of the Rectangle type in the main method (shown below).
  - Note that declaring a variable within a method (called a **local variable** as opposed to an **instance variable**/field) is the same as how we declared the fields in the Rectangle class.

```
class TestProgram {

    public static void main(String[] args) {
       Rectangle demo;
    }
}
```

Declares a variable named demo which will only be able to reference a Rectangle object

Creating an Instance of an Object

- To use a Rectangle object, the object must be instantiated.
  - ***Instantiation*** is the term used when you create an ***instance*** of an object from a class.
- In the example below, a new instance of a Rectangle object is constructed and is assigned to the "demo" variable.

```
class TestProgram {

    public static void main(String[] args) {
      Rectangle demo;
      demo = new Rectangle();
    }
}
```

Instantiates a new Rectangle object and assigns it to the demo variable

Creating an Instance of an Object

- Objects are instantiated using a special method called a ***constructor***.
  - Constructors are used to “set up” or *construct* an instance of an object.
- When there are no constructors present in a class, the compiler automatically adds a ***default constructor***.
  - This guarantees every object has a constructor.
  - Constructors are covered later in this lecture.

```
class TestProgram {

     public static void main(String[] args) {
        Rectangle demo;
        demo = new Rectangle();
    }
}
```

Using an Object's Fields

- We can access an object's fields using dot notation.

```
class TestProgram {

    public static void main(String[] args) {
      Rectangle demo;
      demo = new Rectangle();

      demo.width = 3;
      demo.length = 7;
   }
}
```

Assigns 3 to the width field and 7 to the length field

Using an Object’s Fields

- This example demonstrates how the object’s fields can be updated:

```java
class TestProgram {

    public static void main(String[] args) {
        Rectangle demo;
        demo = new Rectangle();

        demo.width = 3;
        demo.length = 7;
        System.out.println(demo.width);
        System.out.println(demo.length);
        demo.length = 18;
        System.out.println(demo.width);
        System.out.println(demo.length);
    }
}
```

Output:

```
3
7
3
18
```

Creating Multiple Instances of an Object

- A major benefit of object-oriented programming and classes is the ability to reuse parts of our program.

- For example, we can create multiple instances of Rectangle objects, each with their own unique lengths and widths.

- Since they all draw from the same class, all Rectangle objects will have the same types attributes and the same behaviors

Creating Multiple Instances of an Object

- A quick note:
  - You can declare and initialize a variable on the same line.

```
class TestProgram {

    public static void main(String[] args) {
      Rectangle demo = new Rectangle();
    }

}
```

Creating Multiple Instances of an Object

- Four distinct Rectangle objects are instantiated:

```
class TestProgram {

    public static void main(String[] args) {
       Rectangle demo = new Rectangle();
       Rectangle demo2 = new Rectangle();
       Rectangle demo3 = new Rectangle();
       Rectangle demo4 = new Rectangle();
    }

}
```

Creating Multiple Instances of an Object

- The four Rectangle objects can each have different values assigned to their width and length fields (only using the width field in this example):

```
class TestProgram {

     public static void main(String[] args) {
        Rectangle demo = new Rectangle();
        Rectangle demo2 = new Rectangle();
        Rectangle demo3 = new Rectangle();
        Rectangle demo4 = new Rectangle();
        demo.width = 4;
        demo2.width = 7;
        demo3.width = 2;
        demo4.width = 8;
    }

}
```

Creating Multiple Instances of an Object

- Printing the values of the width fields of these objects shows they each store a unique value in their width fields
- The takeaway here is we easily created four objects that have the same exact abilities (*abstraction*) and maintain their own data (*encapsulation*)

```java
class TestProgram {

    public static void main(String[] args) {
        Rectangle demo = new Rectangle();
        Rectangle demo2 = new Rectangle();
        Rectangle demo3 = new Rectangle();
        Rectangle demo4 = new Rectangle();
        demo.width = 4;
        demo2.width = 7;
        demo3.width = 2;
        demo4.width = 8;
        System.out.println(demo.width);
        System.out.println(demo2.width);
        System.out.println(demo3.width);
        System.out.println(demo4.width);
    }

}
```

Output:

```
4
7
2
8
```

Methods

- A ***subroutine*** is a group of self-contained instructions that are executed when called.
  - A **function** is a subroutine that returns data when called.

    ```
    answer = Math.sqrt(x);
    ```

  - A **procedure** is a subroutine that does not return data when called.

    ```
    System.out.println("Hello World!");
    ```

  - A **method** is a subroutine (function or procedure) that is part of a software object.

    ```
    someString.toLowerCase();
    ```

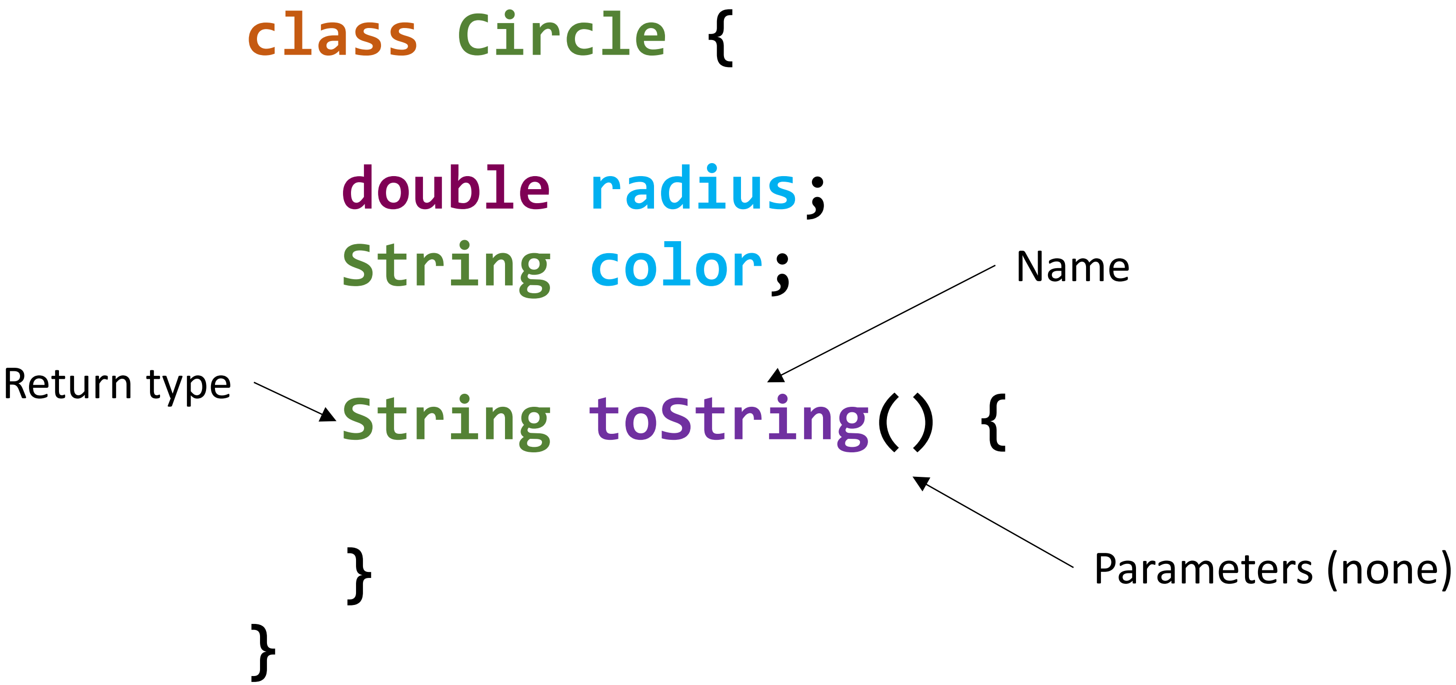
- All four terms are generally used interchangeably, though these are the correct definitions.

Methods

- The first example of defining a method in a class will be to prepare and return a descriptive String about a Circle object.
  - The String will contain (in its text) the current values of the radius and color fields.

- At a minimum, methods must specify
  - **The type of data they return**
    - This method will return a String
  - **The name of the method**
    - We will name this method “toString”
  - **A parameter list**
    - This method will have no parameters
    - We’ll see the use of parameters in the next lecture

Methods



```
class Circle {

    double radius;
    String color;

    String toString() {

    }
}
```

Return type

Name

Parameters (none)

Methods

- Let’s have this method return a String that is in the following format:

```
Circle radius: X and color: Y
```

- Where X is replaced with the current value of the radius field and Y is replaced with the current value of the color field

Methods

```
class Circle {

    double radius;
    String color;

    String toString() {
        String text = "Circle radius: ";
        text = text + radius;
        text = text + " and color: ";
        text = text + color;
    }
}
```

Methods

- Finally, we will need to explicitly state when (and what) to return from the method using a **return statement**.
- Be sure the type of data returned matches the return type that is specified in the method’s declaration.

```
class Circle {

    double radius;
    String color;

    String toString() {
        String text = "Circle radius: ";
        text = text + radius;
        text = text + " and color: ";
        text = text + color;
        return text;
    }
}
```

Methods

- This example calls the Circle object’s toString method, which stored the returned value to a variable named “description”
- The String referenced by “description” is printed

```
class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle();
        demo.radius = 40;
        demo.color = "Blue";

        String description = demo.toString();
        System.out.println(description);
    }

}
```

Output:

```
Circle radius: 40 and color: Blue
```

Methods

- This example essentially does the same thing, but it calls the object's toString method *within* the call to the println method.

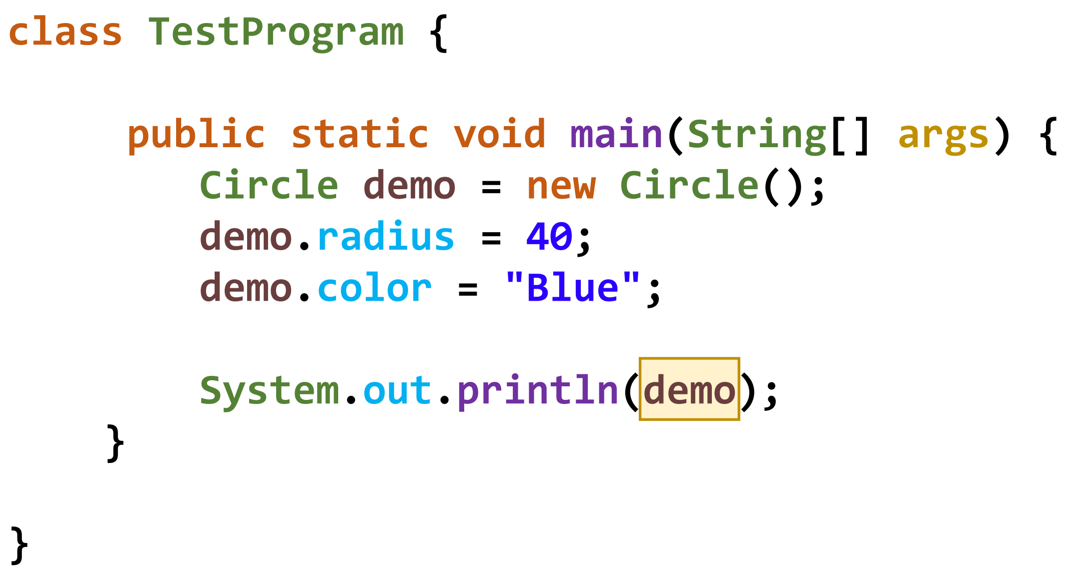
```
class TestProgram {

     public static void main(String[] args) {
        Circle demo = new Circle();
        demo.radius = 40;
        demo.color = "Blue";

        System.out.println(demo.toString());
    }

}
```

Output:

```
Circle radius: 40 and color: Blue
```

Methods

- When an object’s variable is used in the context of a String, the toString() method is called implicitly.



```
class TestProgram {

     public static void main(String[] args) {
        Circle demo = new Circle();
        demo.radius = 40;
        demo.color = "Blue";

        System.out.println(demo);
    }

}
```

Output:

```
Circle radius: 40 and color: Blue
```

Methods

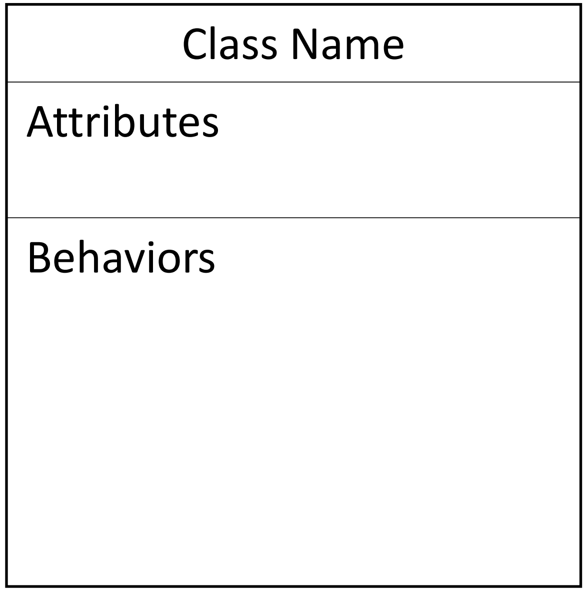
```
class TestProgram {

     public static void main(String[] args) {
        Circle demo = new Circle();
        Circle demo2 = new Circle();

        demo.radius = 40;
        demo.color = "Blue";

        demo2.radius = 25;
        demo2.color = "Green";

        System.out.println(demo.toString());
        System.out.println(demo2.toString());

    }
}
```

Output:

```
Circle radius: 40 and color: Blue
Circle radius: 25 and color: Green
```

Class Diagrams

- Unified Modeling Language (UML) provides a set of standard diagrams for graphically depicting an object-oriented system.
- In UML, each class is shown as a box, with three sections:
  - The Class Name
  - Class Attributes (Fields)
  - Class Behaviors (Constructors and Methods)



Class Name

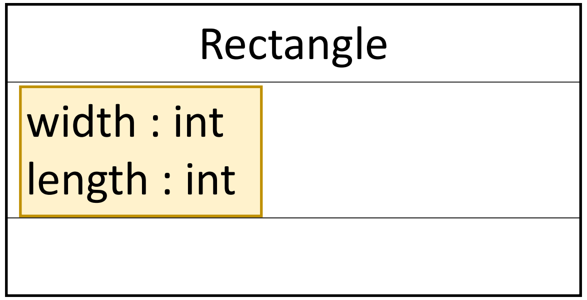
Attributes

Behaviors

Class Diagrams

- When displaying fields in a class diagram, the format to use is:
  - name : type

```
class Rectangle {

   int width;
   int length;

}
```



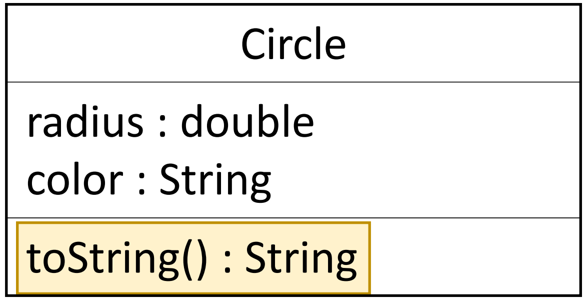
Rectangle

width : int
length : int

Class Diagrams

- When displaying a method in a class diagram, the format to use is:
  - name(arg : type, …) : returnType

```
class Circle {

     double radius;
     String color;

     String toString() {
          String text = "Circle radius: ";
          text = text + radius;
          text = text + " and color: ";
          text = text + color;
          return text;
     }
}
```



Circle

radius : double
color : String

toString() : String

Methods

- Methods are generally categorized as accessor methods or mutator methods.
  - An **accessor method** retrieves (or “gets”) data from an object.
  - A **mutator method** changes (or “sets”) the object’s state.
  - Some may be a hybrid of both, where the method changes the object’s data *and* returns information from the object.

- Some methods neither alter the object’s state nor return information from the object.
  - These are often referred to as **utility methods**.

Utility Methods

- The Circle class’s toString method is mostly a utility method.

```
String toString() {
    String text = "Circle radius: ";
    text = text + radius;
    text = text + " and color: ";
    text = text + color;
    return text;
}
```

- Though, it is somewhat an accessor since it returns information about the object’s state (its current radius and color)
    - However, those data aren’t usable unless it is properly parsed out of the String

Utility Methods

- Suppose we want to add a method that calculates and return the Circle object’s area, based on its current radius.
  - Formula for the area of a circle: $\pi \times r^2$

```
double area() {
    double result = Math.PI * Math.pow(radius, 2);
    return result;
}
```

- This is primarily a utility method but is somewhat an accessor since it returns information about the object’s state.
  - No explicit information about the radius or color, though.

Utility Methods

Output

```
Circle radius: 100.4 and color: Blue
The circle’s area is 31667.756603
```

```
class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle(100.4, "Blue");

        System.out.println(demo.toString());

        double a = demo.area();
        System.out.println("The circle's area is " + a);
    }
}
```

Utility Methods

- Let’s also add a method that calculates and returns the Circle object’s circumference, based on its current radius.
  - Formula for the circumference of a circle: $2\pi \times r$

```
double circumference() {
    double result = 2 * Math.PI * radius;
    return result;
}
```

- This is also primarily a utility method but is somewhat an accessor since it returns information about the object’s state.
  - No explicit information about the radius or color, though.

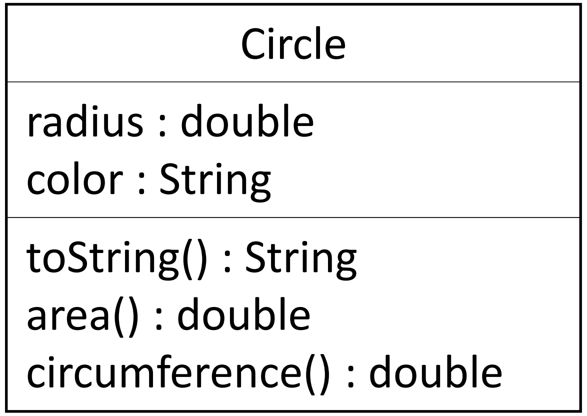
Utility Methods

Output

```
Circle radius: 100.4 and color: Blue
The circle’s area is 31667.756603
The circle’s circumference is 630.831804841
```

```
class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle(100.4, "Blue");

        System.out.println(demo.toString());

        double a = demo.area();
        System.out.println("The circle's area is " + a);

        double c = demo.circumference();
        System.out.println("The circle's circumference is " + c);

    }
}
```

Utility Methods



Circle

radius : double
color : String

toString() : String
area() : double
circumference() : double

Access Modifiers

- **Access modifiers** specify how classes, fields, and methods can be accessed by other objects.
  - This limits other objects from making changes to the data in the fields or calling methods that pertain only to the object's internal implementation.

`public`

- Allows a field or method to be accessible to all other objects.

`private`

- Does not allow a field or method to be accessible to other objects.

Access Modifiers

- When there is no access modifier present (like in the examples shown thus far) then the class, field, or method is said to be **package private**.
  - Other classes within the same package may access that class, field and method; classes outside of that package cannot.

- From this point forward, we will use an access modifier for all classes, fields, constructors, and methods.

Access Modifiers

```
public class Sphere {


}
```

- In most cases, a class will be either public or package private.
- Occasionally, a class might be private in the case of an inner class.
  - Inner classes will be one of the last topics we cover in this course.

Access Modifiers

```
public class Sphere {

   private double radius;


}
```

- Normally, the fields of a class will be private.
  - This better encapsulates the object’s data.
  - Only the object itself can change the value of a private field.
- Constructors are typically public, as we want other objects to be able to instantiate objects of this class.
  - Constructors are sometimes private in situations where inheritance is involved (not covered until CSCI 112)

Access Modifiers

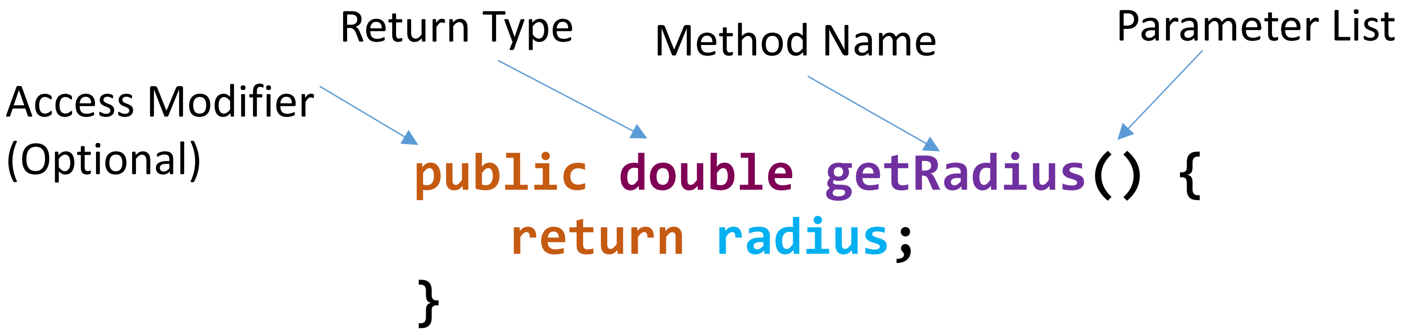
```
class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere();

        demo.radius = 5.6;
        System.out.println("The sphere's radius is " + demo.radius);
  }
}
```

COMPILE ERROR

- A private field is not accessible outside of the class.
- The Sphere class can contain methods that either retrieve or change the value of a private field.

Accessor Methods

- An **accessor method** retrieves (or “gets”) data from an object.
  - Colloquially called a “getter” method.



```
public double getRadius() {
    return radius;
}
```

- This method will return the current value of our radius field.
- The type of data returned must match the method’s return type.
  - Since this method’s return type is double, the method can only return a double value.

Accessor Methods

```
public class Sphere {

    private double radius;

    public double getRadius() {
        return radius;
    }

}
```

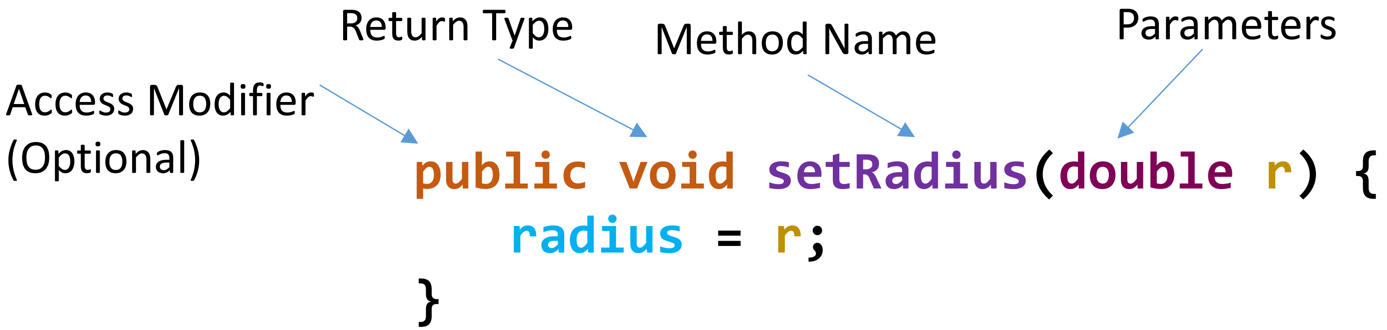
Mutator Methods

- ***Parameters*** are variables that represent data that is given (or *passed*) to a method.
  - The data given to the parameters are called ***arguments***.
  - Though, *parameter* and *argument* are often used interchangeably.

- If a method declares one or more parameters, a value for each parameter must be present.
  - The parameters must indicate the data type for each parameter.

Mutator Methods

- A **mutator method** alters (or “sets”) the data of an object.
  - Colloquially called a “setter” method.



Return Type

Method Name

Parameters

Access Modifier (Optional)

```
public void setRadius(double r) {
    radius = r;
}
```

- This method will change the current value of the object’s radius field.
- A method that does not return data (no return statement) must have a **void** return type.
  - Mutators typically do not return data. They “set” data- they don’t “get” data.

Mutator Methods

```
public class Sphere {

    private double radius;

    public double getRadius() {
        return radius;
    }

    public void setRadius(double r) {
        radius = r;
    }

}
```

Accessor and Mutator Methods

```java
class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere();

        demo.setRadius(104.5);
        System.out.println("The sphere's radius is " + demo.getRadius());
    }

}
```

Output

```
The sphere’s radius is 104.5
```

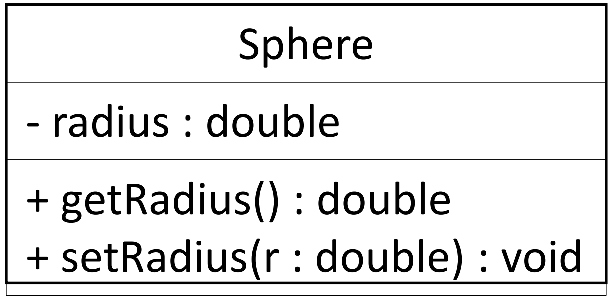
Class Diagrams (Access Specifiers)

- Access specifier symbols:

+	public fields/methods
-	private fields/methods
None (or ~)	No access specifier.

Class Diagrams (Access Specifiers)

```
public class Sphere {

    private double radius;

    public double getRadius() {
        return radius;
    }

    public void setRadius(double r) {
        radius = r;
    }

}
```



Sphere

- radius : double

+ getRadius() : double
+ setRadius(r : double) : void

Constructors

- Constructors are a special type of method that prepares the instance of the object.
- Constructors always have the same name as the class.
- Classes may have multiple constructors.
  - This is referred to as ***overloading*** or having ***overloaded constructors***.

The No Argument (No-Arg) Constructor

```
class Circle {

    double radius;
    String color;

    Circle() {
        radius = 1;
        color = "Red";
    }

}
```

- Replaces the default constructor added by the compiler.
  - A class only has a default constructor when the class has no constructors defined.
- The code in the no-arg constructor’s body will be executed when the no-arg constructor is called.
- In this example, the radius field will be set to 1 and the color field is set to Red when the no-arg constructor is called.

The No Argument (No-Arg) Constructor

```java
class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle();

        System.out.println("The circle's radius is " + demo.radius);
        System.out.println("The circle's color is " + demo.color);

    }
}
```

Output

```
The circle’s radius is 1
The circle’s color is Red
```

Overloaded Constructors

- A class can have more than one constructor.

- *Overloading* is the term used to describe two or more constructors or methods that have the same name but different parameter lists.

- A constructor is always overloaded when there are at least two constructors present.
  - Constructors will always have the same name (the class's name) but each can have a varying number of parameters.

Overloaded Constructors

```
class Circle {

    double radius;
    String color;

    Circle() {
        radius = 1;
        color = "Red";
    }

    Circle(double r) {
        radius = r;
        color = "Red";
    }

}
```

- There are now two ways to instantiate a Circle object
  - Without an argument (radius is set to 1; color is set to Red)
  - With a double-type argument (radius is set to the value passed to the r parameter; color is set to Red)

Overloaded Constructors

```java
class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle();
        Circle demo2 = new Circle(76.5);

        System.out.println("The first circle's radius is " + demo.radius);
        System.out.println("The second circle's radius is " + demo2.radius);
    }
}
```

Output

```
The first circle’s radius is 1
The second circle’s radius is 76.5
```

Overloaded Constructors

```
class Circle {

    double radius;
    String color;

    Circle() {
        radius = 1;
        color = "Red";
    }

    Circle(double r) {
        radius = r;
        color = "Red";
    }

    Circle(double r, String c) {
        radius = r;
        color = c;
    }

}
```

- There is now a third way to instantiate a Circle object
    - With a double-type argument (radius is set to the value passed to the r parameter) followed by a String-type argument (color is set to the value passed to the c parameter)

Overloaded Constructors

Output

```
The first circle’s radius is 1
The first circle’s color is Red

The second circle’s radius is 76.5
The second circle’s color is Red

The third circle’s radius is 100.4
The third circle’s color is Blue
```

```java
class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle();
        Circle demo2 = new Circle(76.5);
        Circle demo3 = new Circle(100.4, "Blue");

        System.out.println("The first circle's radius is " + demo.radius);
        System.out.println("The first circle’s color is " + demo.color);
        System.out.println();
        System.out.println("The second circle's radius is " + demo2.radius);
        System.out.println("The second circle’s color is " + demo2.color);
        System.out.println();
        System.out.println("The third circle's radius is " + demo3.radius);
        System.out.println("The third circle’s color is " + demo3.color);
    }
}
```

Overloaded Constructors

- There is no limit to the number of constructors in a class.
- However, each constructor must have a unique ***signature***.
    - A constructor signature consists of its name and parameter list data types.

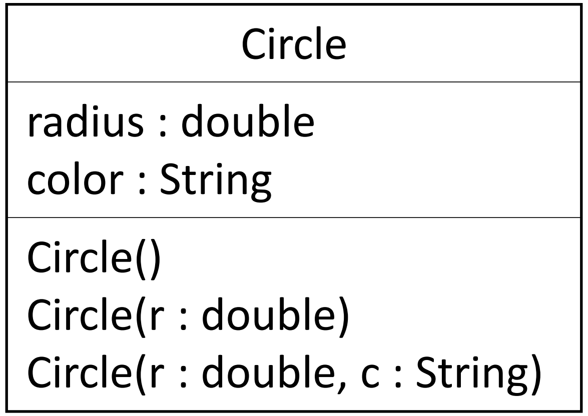
```
Circle()
Circle(double r)
Circle(double r, String c)
```

Signatures:

```
    Circle()
    Circle(double)
    Circle(double, String)
```

Overloaded Constructors

```
class Circle {

    double radius;
    String color;

    Circle() {
        radius = 1;
        color = "Red";
    }

    Circle(double r) {
        radius = r;
        color = "Red";
    }

    Circle(double r, String c) {
        radius = r;
        color = c;
    }

}
```



Circle

radius : double
color : String

Circle()
Circle(r : double)
Circle(r : double, c : String)

Method Overloading

```
public class Sphere {

    private double radius;

    public Sphere(double r) {
        radius = r;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double r) {
        radius = r;
    }

    public void setRadius(String r) {
        radius = Double.parseDouble(r);
    }

}
```

- Like constructors, methods can also be overloaded.
- Like overloaded constructors, overloaded methods must have unique signatures.
  - setRadius(double)
  - setRadius(String)

Method Overloading

```
public class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere(65.4);

        System.out.println("The sphere’s radius is " + demo.getRadius());

        demo.setRadius(14.5);
        System.out.println("The sphere’s radius is " + demo.getRadius());

        demo.setRadius("83.214");
        System.out.println("The sphere’s radius is " + demo.getRadius());
  }
}
```

Output

```
The sphere’s radius is 65.4
The sphere’s radius is 14.5
The sphere’s radius is 83.214
```

Encapsulation

- Limiting access to an object’s fields gives the object greater control over its encapsulated data.
    - Consider the example below, where the Sphere constructor allows any value (even values that don’t make sense, like a negative value) to be set to the radius field

```
public class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere(-73.65);
        System.out.println("The sphere’s radius is " + demo.getRadius());
    }
}
```

Output

```
The sphere’s radius is -73.65
```

Encapsulation

```
public class Sphere {

    private double radius;

    public Sphere(double r) {
        if(r > 0) {
            radius = r;
        }
        else {
            radius = 1;
        }
    }

    ... (Other methods)
}
```

- The constructor now checks that the parameter’s value is greater than 0
  - If it is, it assigns that value to the radius field
  - If it is not (zero or negative), it assigns a default value of 1 to the radius field.

Encapsulation

```java
public class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere(-73.65);
        System.out.println("The sphere’s radius is " + demo.getRadius());
    }
}
```

Output

```
The sphere’s radius is 1
```

Encapsulation

- While the constructor performs a check, the setter methods both still allow invalid data to be assigned to the radius field

```
public class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere(-73.65);
        System.out.println("The sphere’s radius is " + demo.getRadius());
        demo.setRadius(-9.25);
        System.out.println("The sphere’s radius is " + demo.getRadius());
    }
}
```

Output

```
The sphere’s radius is 1
The sphere’s radius is -9.25
```

Encapsulation

```java
public class Sphere {

    private double radius;

    public Sphere(double r) {
        if(r > 0) {
            radius = r;
        }
        else {
            radius = 1;
        }
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double r) {
        if(r > 0) {
            radius = r;
        }
        else {
            radius = 1;
        }
    }

}
```

- We could add the same logic to the setter method, but this adds:
  - More source code to maintain
    - A benefit of object-oriented programming is that it is supposed to reduce the amount of source code to maintain
  - More of the *same* source code
    - A benefit of object-oriented programming is that it is supposed to make code more reusable

Encapsulation

```
public class Sphere {

    private double radius;

    public Sphere(double r) {
        validateRadius(r);
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double r) {
        validateRadius(r);
    }

    private void validateRadius(double r) {
        if(r > 0) {
            radius = r;
        }
        else {
            radius = 1;
        }
    }
}
```

- A solution is to create a utility method that specifically handles validating the data to be assigned to the radius field:
  - The method is private, so that only the other methods in the Sphere class can call on it
  - This one method can contain all necessary logic for error-checking data prior to assigning anything to the object’s radius field.
  - Any new methods or constructors in the class can simply call this method, and it will ensure that the radius field is set correctly.

Encapsulation

```java
public class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere(-73.65);
        System.out.println("The sphere's radius is " + demo.getRadius());
        demo.setRadius(-9.25);
        System.out.println("The sphere's radius is " + demo.getRadius());
    }
}
```

Output

```
The sphere’s radius is 1
The sphere’s radius is 1
```