Lecture 02

Outline

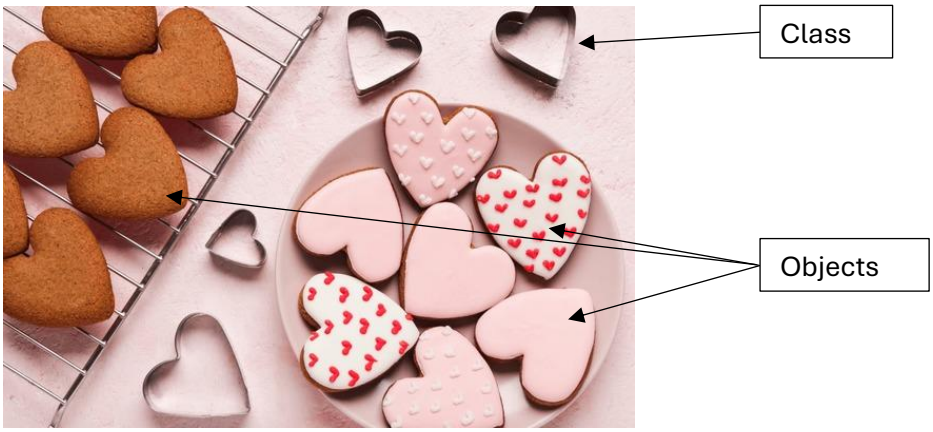
1. Objects
    a. User Defined Objects
    b. Predefined Objects (Libraries)
2. Classes
    a. Properties – Usually *private*
        - Representing the state of data
    b. Methods – Usually *public*
        - Representing the behavior
3. Abstraction
    a. What?
    b. Why?
    c. How?
4. Access Modifiers
    a. public
    b. protected
    c. default
    d. private
5. Getters and Setters
    a. What?
    b. Why?
    c. How?
6. Method Overloading
    a. What is it?
    b. Rules
        - Same name
        - Differ in number, type, or order of parameters.
7. Initializing the Attributes of an Object
    a. Constructors
        - Default Constructors
        - Overloaded Constructors

Objects

Objects are the instances of classes. In real world examples it has **attributes** and **functions**.

Analogy: Cookie Cutter Analogy

Imagine a cookie cutter as a **class**. If you do have a cookie cutter you can create the same shape and sized cookies (instances of cookie cutter: **objects**) with some dough. However, the dough (**parameter**) that you use will affect the color, taste, texture. But the shape will be the same.



There are two kinds of objects.

1. **User Defined Objects**
   In this case user creates their own classes and make out objects from that.

2. **Predefined Objects**
   In this scenario users create objects from the predefined classes (Libraries).

Calc.java × CalcDemo.java

```java
package Lecture02;

public class Calc {  2 usages
    private final int input;  4 usages

    Calc(int input){  1 usage
        this.input = input;
    }

    public int add(int input){  1 usage
        return this.input + input;
    }

    public int power(){  1 usage
        return input * input;
    }
}

```

Figure 1: Creating a user defined class

Calc.java CalcDemo.java ×

```java
package Lecture02;

public class CalcDemo {

    public static void main(String[] args) {
        Calc CalcValue = new Calc( input: 10); //Create the new object "CalcValue"

        System.out.println(CalcValue.add(20));
        System.out.println(CalcValue.power());
    }
}

```

*Figure 2: Creating an object named **CalcValue** from class **Calc***

scannerDemo.java

```java
package Lecture02;

import java.util.Scanner;

public class scannerDemo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.print("Enter a value: ");
        int input = scan.nextInt();

        System.out.println("Value: " + input);
    }
}
```

*Figure 3: Create **scan** object from the **predefined class "Scanner"***

Classes

Properties

- Properties are the ***characteristics*** of the object.
- They basically represent the **state** or **data** of the object.
- Synonyms: **Attributes**, **Data Members.**
- Usually these properties are private, which means only they could be used within the class itself.

Methods

- Methods describe the ***behaviors*** of the object.
- Synonyms: **Behaviors**, **Member Functions**
- Usually methods are public, which means can be accessed both inside and outside the class.
- However, it's possible objects have ***private methods*** as well.

Analogy: Lego Car Analogy

Imagine you have a LEGO instruction manual (**Class**). It'll guide you through building a car. Now, the instruction manual is basically a ***blueprint*** of a car.

Assume that, to build the simplest car, you need these five things (**Attributes/ Properties**). These are the LEGO pieces that you need to assemble into the blueprint of the car.

- 4 Wheels
- 1 Steering Wheel
- 1 Body
- 1 Windshield
- 1 DC Motor

You might get these pieces from different sizes, colors and textures. These attributes will literally define **what the car looks like**.

Now, once we initialized these properties (assembled pieces), the car should be able to:

- Drive forward
- Turn left or right
- Stop

It should be clear to you now; these are the ***behaviors*** of the car. Which means in programing context these are the ***methods/ functions*** of the car.

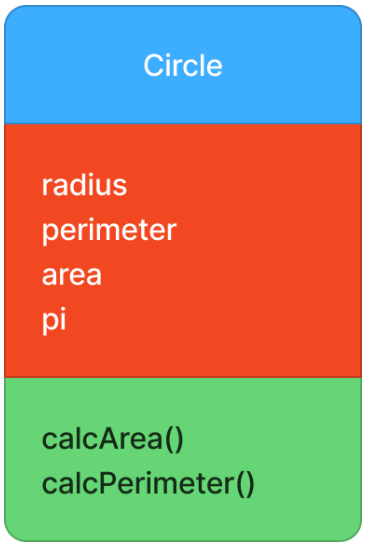
In Simple Terms:

- **Class = LEGO manual 📖 (the plan)**
- **Attributes = LEGO pieces 🧩 (the parts of the car)**
- **Methods = What the car can do 🚙 (moving, stopping, turning)**

Let's look at a real-world example. Assume we are going to create a **Circle** class, and it has four **attributes**.

1. Radius
2. Perimeter
3. Area
4. Pi

And it should be able to **calculate the area** and **calculate the perimeter**. These two would be the methods that we want to use. It's easier to understand this with a UML class diagram.



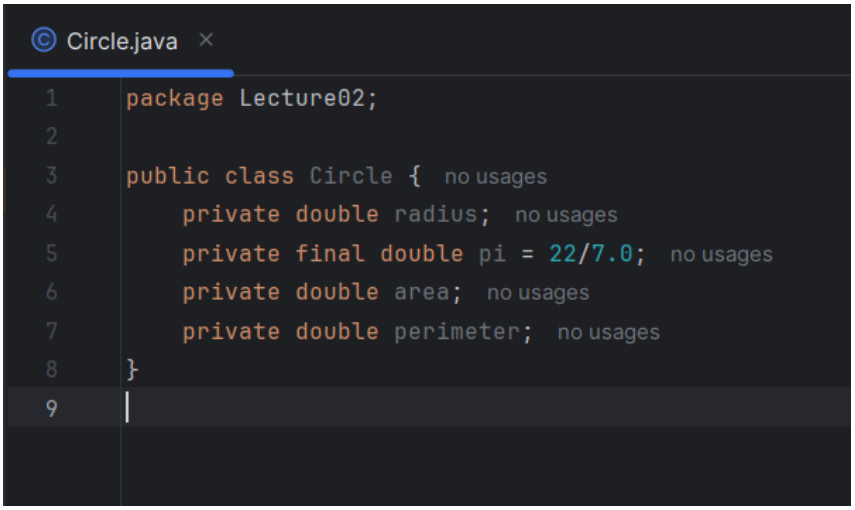
Blue: Class Name

Orange: Attributes

Green: Properties

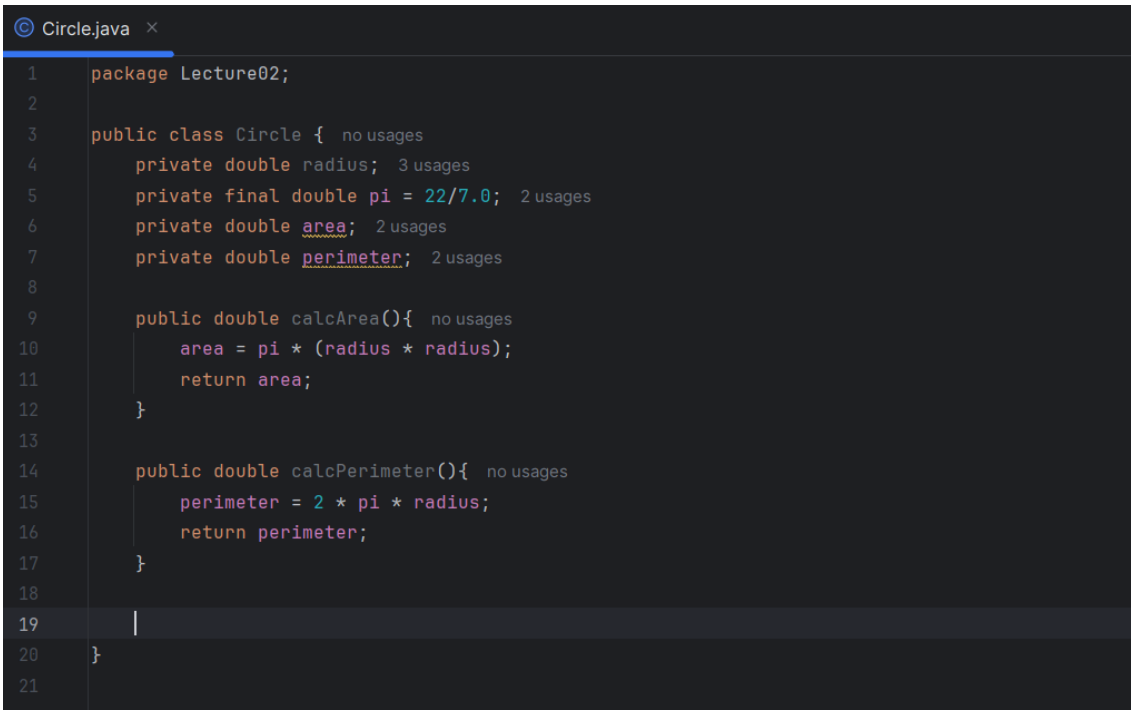
Now let's code this.

First let's just define the class itself and attributes.



Notice that we've defined all attributes as private, as we want to restrict direct access of the attributes from outside the class.

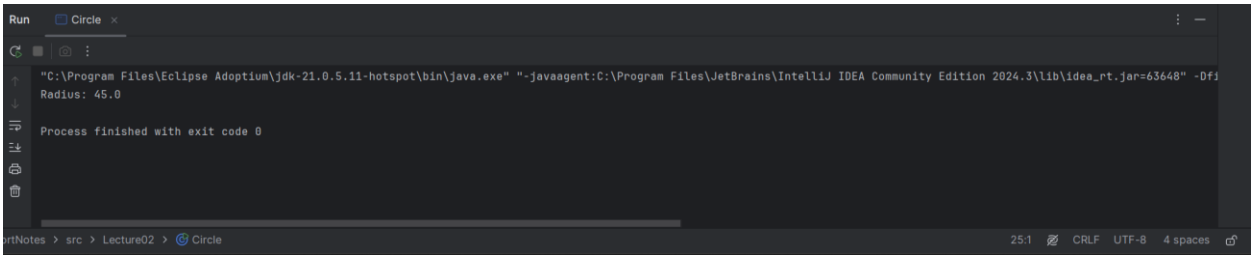
However, we want to implement methods in a way that can be accessed both in and out of the class. Let's implement both calcArea() and calcPerimeter() methods, which return the area and the perimeter of our circle, respectively.



So, we used **private** to define attributes and **public** to define methods. Now, let's see what happens if we try to access these attributes within the class.

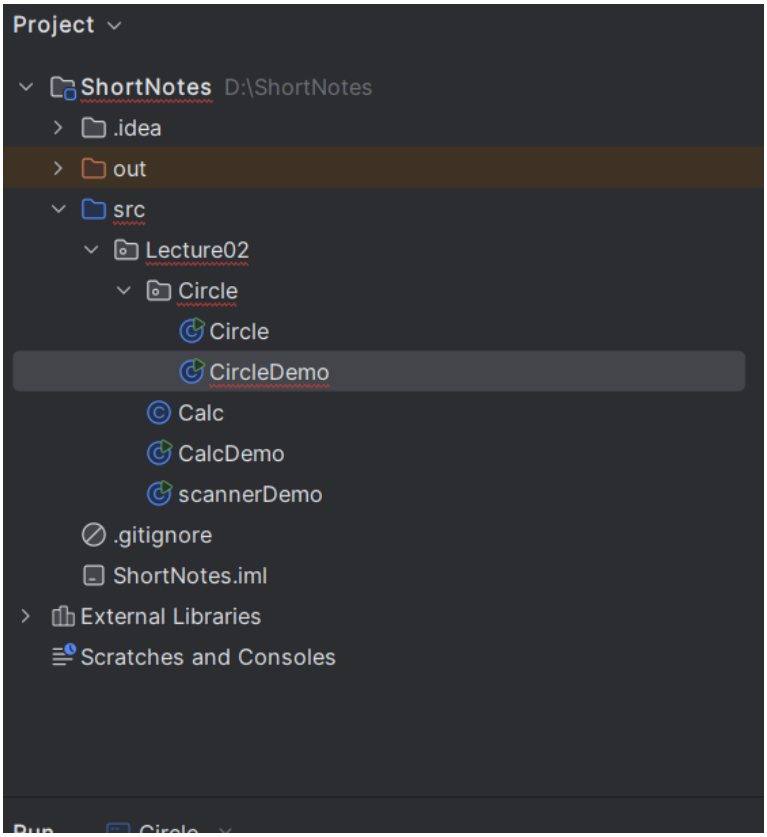
```java
package Lecture02;

public class Circle {
    private double radius;  5 usages
    private final double pi = 22/7.0;  2 usages
    private double area;  2 usages
    private double perimeter;  2 usages

    public double calcArea(){  no usages
        area = pi * (radius * radius);
        return area;
    }

    public double calcPerimeter(){  no usages
        perimeter = 2 * pi * radius;
        return perimeter;
    }

    public static void main(String[] args) {
        Circle circle1 = new Circle();
        circle1.radius = 45;
        System.out.println("Radius: " + circle1.radius);
    }
}
```

In this main method inside the class, we've accessed an instance of Circle class directly and it gives us no errors. Let's try to run that code.

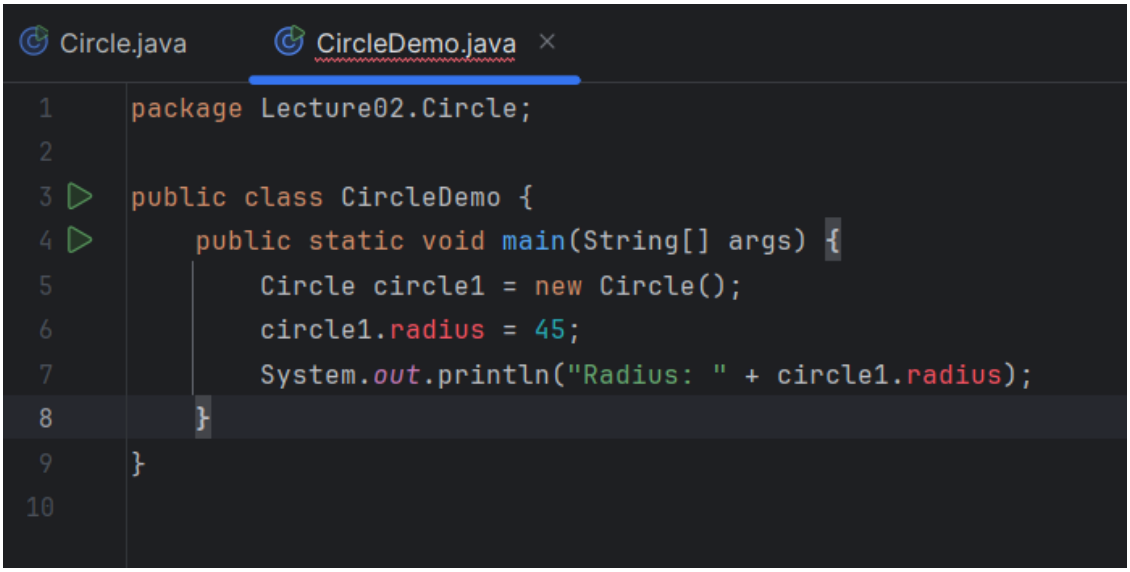


It works just fine inside the class. Now, let's create another file inside the same folder (**package**) and see what will happen if we try to do the same.

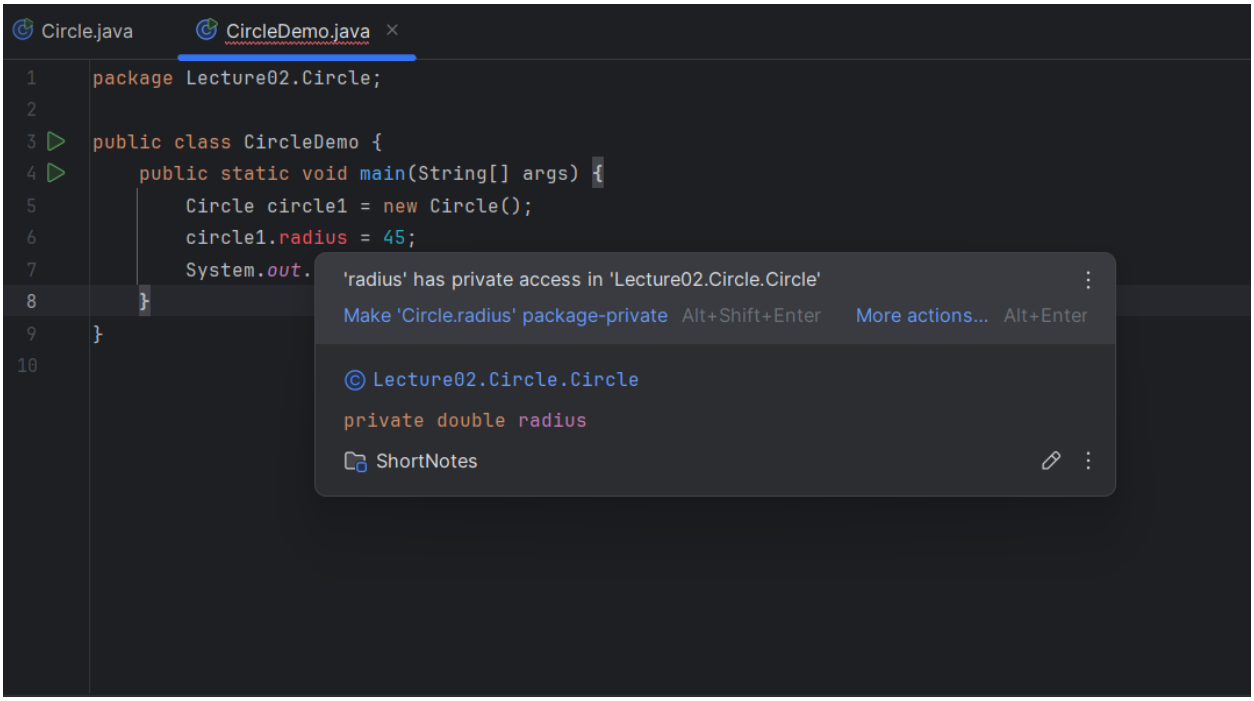
First take a look at the file structure.



And now let's code in **CircleDemo** file.

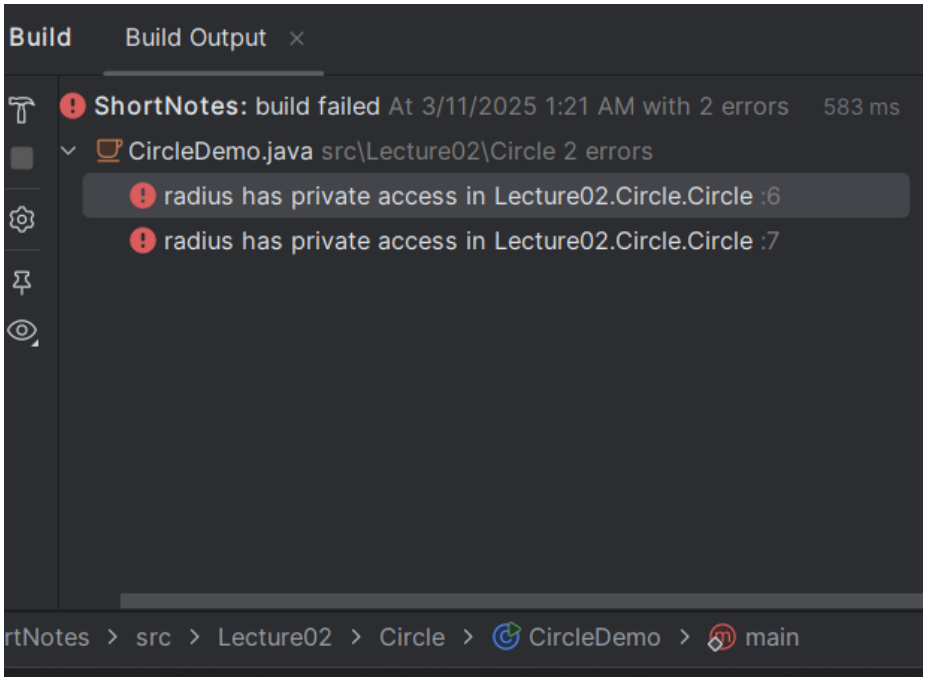


You can see there are errors, which are shown in red color. Let's hover over and see what's the problem.



Here, you can see it says radius has private access in Circle class. Which means it cannot be directly accessed out of the class.

Let's try to run this code ignoring the error.



It runs into an error. Now I hope you can understand the difference between private and public keywords. To avoid these errors, we can use **setters** and **getters**. More on that later. Let's return to this same problem in an upcoming chapter.

Abstraction

- Please notice that the person who created this did not use any drugs before getting into abstractions.
- Abstraction is the process of removing unnecessary **characteristics** from something in order to reduce it to a set of **essential characteristics** that are needed for the particular system.
- Assume you want to create a system for a school which stores students' and teachers' details.
- However, the IT ~~professional~~ who created it decided to include the following properties into the student class.
    - Admission Number
    - Student Name
    - Guardian' Telephone Number
    - Student' Ex GFs phone number
    - Relationship Status
    - Pet's name

  Bruh 💀
- And he forgot to include the following properties.
    - Marks
    - Class Rank
    - Current Grade

- The above issue happened because he did not know about the concept of abstraction (or was on drugs).

Let's ask:

1. **What** is abstraction**?**
   Abstraction is the process of removing unnecessary characteristics from something, to reduce it to set of essential characteristics that are needed for the particular system.

   According to the system the necessary characteristics would be different.
    a. **Facebook:** Wants to know **who you dated, who you'll date next, and how bad your last breakup was.**
    b. **School:** Only cares about **your grades, your class rank, and whether you're secretly using school Wi-Fi to watch Netflix.**
2. **Why?**
   Why do you need abstraction? It's pretty much okay to have your relationship status on the school student database right (Unless you are in a third world developing country)?

   Bro, we're calculating your grade, not your dating history! Why on earth do we need your ex's number to compute your final marks?

Even the worst school database in the world shouldn't mix **relationship status** with **class rank**—unless it's tracking how breakups affect your GPA

That's why! Get rid of all unnecessary information which makes our code complex and messy.

3. **How?**
   How do we do that?
   >>**First read the question.**
   >>**Identify the necessary properties**
   >>**Only define the necessary properties**
   >>**If you did it right at the end of code there should not be any** no usage **code segments**.

🚨 **How to Use Abstraction Like a pro:**
✅ Keep **useful** stuff (marks, grade, rank)
❌ Remove **nonsense** (ex-GF's number, relationship drama, pet names)
🤦 If your code **looks like a gossip column**, you're doing it wrong!

Access Modifiers

Return to private and private part in **Class** topic if you don't understand following table. That's all we have to discuss under this topic.

	Default	Private	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package Subclass	Yes	No	Yes	Yes
Same Package Non-Subclass	Yes	No	Yes	Yes
Different Package Subclass	No	No	Yes	Yes
Different Package Non-Subclass	No	No	No	Yes

Getters and Setters

Okay! Back to our circle class. What happened when we tried to access properties in CircleDemo file?

We understand that we cannot access private property outside the class (Check the above table. That's true.).

To access an attribute (property) we can implement two methods.

1. A getter: To get the value of the attribute
2. A setter: To set (initialize) the value of the attribute

Back to your questions.

1. **What** are those getters and setters?
   Simple! As attributes in a class generally described as private, we can not use them directly in another class (another java file).
   So:
   Setters will assign values to private attributes.
   Getters will return values from private attributes.

2. **Why** do we need those getters and setters?
   Why don't we just make everything public like a true democracy? Because, **bro**, you don't want some random hacker messing with your private data like it's an open buffet!

   Therefore, please implement getters and setters instead of making everything public. You won't make your GF public, will you? 💀

3. **How** do we implement them?
   Easy! Just implement them like I did below.

Circle.java × CircleDemo.java

```java
package Lecture02.Circle;

public class Circle {
    private double radius;  5 usages
    private final double pi = 22/7.0;  2 usages
    private double area;  2 usages
    private double perimeter;  2 usages

    public void setRadius(double radius) {  no usages
        this.radius = radius;
    }

    public double getRadius() {  no usages
        return radius;
    }

    public double calcArea(){  no usages
        area = pi * (radius * radius);
        return area;
    }

    public double calcPerimeter(){  no usages
        perimeter = 2 * pi * radius;
        return perimeter;
    }

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

Okay you did something. How do we use them outside the class?

>> Wait a minute. Not finished yet! Take a look at the structure of getters and setters.

Set

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

- Method is **public**.
- Return type is **void** (By assigning we do not return a value)
- In parenthesis we mention the **parameters**. In this example (**double radius**)**.** Which is basically the input for radius.
- **this.radius** means attribute **radius** in the class itself.
- **this.radius = radius; //**What this line basically says is (from left to right),
    - "Okay! There are two radiuses in this class right now (Like 3 Peter Parkers in **No Way Home**). Our radius (OG radius), let's call him **this.radius.** So, assign the new radius value to **this.radius** (OG radius).

Get

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

- Method is **public**.
- The return type is **double** (As we defined radius as double).
- To get the radius we need no parameters. So, this is all fine.
- In here you don't have to use **this** keyword, as there are no multiple instances of spider man.

FOR DC FANS

"When you see this.radius = radius;, think of it like this:"

1. **First "radius" (right side) is the new value you received as input.**
2. **"this.radius" (left side) is the actual attribute inside your class.**
3. **We are simply updating the OG radius with the new value.**

💡 **Think of it like renaming a contact:**

- You had **"Bae ❤️‍🩹"** saved in your phone.
- But now you need to update it to **"Ex 💀"**
- You don't create a new contact—you just **update the existing one** (same with this.radius = radius;)

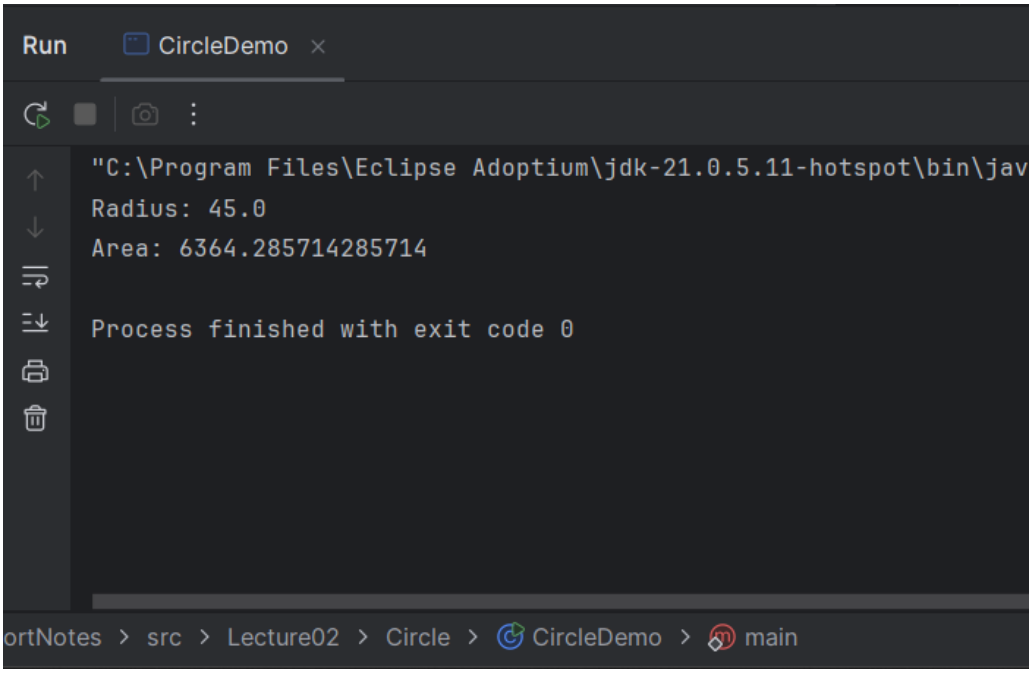
Ah! Almost forgot. Let's use getters and setters in CircleDemo.java now.

Circle.java CircleDemo.java

```
1  package Lecture02.Circle;
2
3  public class CircleDemo {
4      public static void main(String[] args) {
5          Circle circle1 = new Circle();
6          circle1.setRadius(45);
7          System.out.println("Radius: " + circle1.getRadius());
8          System.out.println("Area: " + circle1.calcArea());
9      }
10 }
11
```

No red words. We've replaced circle1.radius with **getRadius()** and **setRadius()** methods. Also we used our **calcArea()** method here!

Let's just try to run it 🏃



If it works: **don't touch it** 🚫

Method Overloading

- Method Overloading is basically having the same method, but with different parameters it'll return different output values.
- Java should be able to understand ***how*** you want to use the method, just looking at the parameters. So, the parameter order, type or number should be different.
- Now let's look into a simple example to understand this.

  Assume you are dumb and became a university dropout. Now you are working in a coffee shop. In your coffee shop there is a coffee machine, which makes coffee (*What else the freaking machine supposed to do?!*).

  There is a button to make coffee in that machine which says *pour*. Once you click this button machine will add two TBSP of coffee powder and 1TBSP of sugar to it.

  However, there are some customers (*definitely your collegemates who got the degree*) who would like to have a coffee with 3 TBSP of coffee powder within it. For that, the machine has a keyboard. After inserting how many coffee TBSPs you want, you can press the ***pour*** button.

Now, finally there are customers who would like to have their own favorite coffee in the shop. To do that you have to pass all two values (sugar count and coffee powder count) and press ***pour*** button.

As you can see, all these times you have to press ***pour*** button. But as you pass some value to it, it'll function differently. It will **return** different types of coffee based on the parameters.

Let's create a class that accepts no parameters. We'll name this class as objectMath. This object should be able to,

1. Calculate the volume of a cube.
2. Calculate the volume of a cuboid.

First let's create the initial class.

```
package Lecture02.solidObjects;

public class objectMath {
    public int calcVolume(int side){
        return side * side * side;
    }

    public int calcVolume(int length, int width, int height){
        return length * width * height;
    }

}
```

Now let's try to create an object from this and measure the volume of a cube and a cuboid.

```
package Lecture02.solidObjects;

public class calcVolumeDemo {
    public static void main(String[] args) {
        objectMath calculator  = new objectMath();

        System.out.println("Volume of cube: " + calculator.calcVolume(10));

        System.out.println("Volume of cuboid: " +
calculator.calcVolume(5,10,15));
    }
}
```

Now what would be the output of this code?

Java knows which function to execute based on the parameters.

Initializing the Attributes of an Object

The **constructor** is what that helps us to create a new object using classes.

So, the constructer with no parameters is known as **default** constructor.

- If you use default constructor to create an object it will use default attribute values to create the object.
- If you create a class but do not create a constructor, Java automatically provides a constructor.
- Here are the default values that default constructor will initialize:
    - For integers: 0
    - For doubles and floating-point numbers: 0.0
    - For objects: null

Assume we implemented the following class “Employee”. We have not created any constructors. So, java will use the default constructor.

```
package Lecture02.employee;

public class Employee {
    private int empID;
    private String empName;
    private double empSal;
    private float empBonus;


    public void displayDetails(){
        System.out.println("Employee ID: " + empID);
        System.out.println("Employee Name: " + empName);
        System.out.println("Employee Salary: " + empSal);
        System.out.println("Employee Bonus: " + empBonus);
    }
}
```

Can you guess the initial values for the following properties?

1. empID
2. empName
3. empSal
4. empBonus

Here we created another java file and created emp1 object from it.

```java
package Lecture02.employee;

public class employeeDemo {
    public static void main(String[] args) {
        Employee emp1 = new Employee();
        emp1.displayDetails();
    }

}
```

And here is the output.

Overloaded Constructors

Now let's move on to the overloaded constructors. The constructors that accept parameters are known as **overloaded constructors**.

Same as **overloaded functions** you can have more than one overloaded constructor to implement a single class.

```
package Lecture02.OLCons;

public class Driver {
    private int driverID;
    private String driverName;
    private double salary;
    private int rating;

    Driver(int driverID, String driverName, double salary, int rating){
        this.driverID = driverID;
        this.driverName = driverName;
        this.salary = salary;
        this.rating = rating;
    }

    Driver(int driverID, String driverName, double salary){
        this.driverID = driverID;
        this.driverName = driverName;
        this.salary = salary;
    }

    Driver(int driverID, String driverName){
        this.driverID = driverID;
        this.driverName = driverName;

    }

    public void displayDetails() {
        System.out.println("Driver ID: " + driverID);
        System.out.println("Driver Name: " + driverName);
        System.out.println("Salary: " + salary);
        System.out.println("Rating: " + rating);
        System.out.println("---------------------------------");
    }


}
```

Here all the constructors are overloaded constructors. Let's create three objects using these three constructors and print the output.

```java
package Lecture02.OLCons;

public class DriverDemo {
    public static void main(String[] args) {
        // Creating objects using different constructors
        Driver driver1 = new Driver(101, "Alice", 5000, 4);
        Driver driver2 = new Driver(102, "Bob", 4500);
        Driver driver3 = new Driver(103, "Charlie");

        // Displaying details of each driver
        driver1.displayDetails();
        driver2.displayDetails();
        driver3.displayDetails();
    }
}
```