

# **Unit 5: Writing Classes**

## **Anatomy of a Class**

Adapted from:

- 1) Building Java Programs: A Back to Basics Approach  
by Stuart Reges and Marty Stepp
- 2) Runestone CSAwesome Curriculum

# Classes and Objects

In Unit 2, we learned to use **classes** and **objects** that are built-in to Java or written by other programmers. In this unit, you will learn to write your own classes and objects!

Remember that a **class** in programming defines a new **abstract data type**. When you create **objects**, you create new variables or **instances** of that class data type.

```
String a = new String("hello");  
Scanner input = new Scanner(System.in);  
Sprite player = new Sprite(200, 400);
```

# Declaration

To write your own class, you typically start a class declaration with `public` then `class` then the name of the class. The body of the class is defined inside the curly braces `{}`.

```
public class ClassName {  
    // define class here - a blueprint  
  
}
```

Then, you can create objects of that new type by using:

```
ClassName objectname = new ClassName();
```

# Instance Attributes/Methods

Remember that objects have attributes and behaviors. These correspond to **instance variables** and **methods** in the class definition.

Instance variables hold the **data** for objects whereas the methods code the **behaviors** or the **actions** that can manipulate the data of the object.

A class also has **constructors** which initialize the instance variables when the object is created.

# Point Class

```
public class Point {  
    int x;  
    int y;  
    public Point(int newX, int newY) {  
        x = newX;  
        y = newY;  
    }  
}
```

declare instance  
variables



constructor: initialize  
variables



# Instance Variables

**instance variable:** A variable inside an object that is part of its data. Also called fields, attributes or properties.

- Each object has *its own copy* of each instance variable.

A variable is **public** by default. A public variable can be accessed from outside the class. The keyword **private** can be used to protect access to a variable. A private variable can only be accessed from inside the enclosing class.

Declaration syntax:

**private type name ;**

# An Example

Here's an example of a simple class that we saw in Unit 2: The Sprite class.

Sprite.java

```
public class Sprite{  
    private double center_x;  
    double center_y;
```

declaring the **instance variables**

an instance variable is **private** if it is only accessible/visible inside the class.

```
    public Sprite(double x, double y){  
        center_x = x;  
        center_y = y;  
    }  
}
```

**constructor**

initialize the instance variables.

The Sprite class is not a runnable program. A program always needs a class with the main method: the driver class.

# Driver Class

Here's the driver class(class with the main method). The driver class's main method controls the flow and logic of the entire program.

Main.java

```
public class Main{  
    public static void main(String[] args){  
        Sprite player1 = new Sprite(20, 100);  
        Sprite player2 = new Sprite(10, 50);  
  
        System.out.println(player1.center_x);  
        player1.center_x = 10;  
  
        System.out.println(player1.center_y);  
        player1.center_y = 100;  
    }  
}
```

Use the new operator  
along with calling the  
constructor to create  
objects.

Error! center\_x is  
a private variable,  
cannot be  
accessed outside  
the class!

Since center\_y is  
not declared  
private, this is ok.




# Private Access

If we can't directly work with the private variables of a class from outside the class, then how can we work with variables of the class?

Main.java

```
public class Main{  
    public static void main(String[] args){  
        Sprite player1 = new Sprite(20, 100);  
        Sprite player2 = new Sprite(10, 50);  
  
        System.out.println(player1.center_x);  
        player1.center_x = 10;  
    }  
}
```

Error! Can't  
access private  
variables!



We can create methods that can give us controlled, indirect access to the instance variables. We discuss this next.

# Instance methods

**instance method** (or **object method**): Exists inside each object of a class and gives behavior to each object. Instance method can modify/access instance variables. The keyword `public` means that the method can be called from outside the class.

```
public type name (parameters) {  
    statements;  
}
```

– same syntax as static methods, but **without** `static` keyword

Although we can't directly access private variables, the public methods allow us to indirectly access/manipulate them.

# Encapsulation

**Object-oriented Programming** stresses **data encapsulation** where the data (instance variables) and the code acting on the data (methods) are wrapped together into a single unit and the implementation details are hidden.

The data is protected from harm by being kept private. Anything outside the class can only interact with the public methods and cannot interact directly with the private instance variables.

There are two kinds of methods.

**1) accessor:** A method that lets **client code**(code that uses the class; outside of the class) examines object state.

**2) mutator:** A method that modifies an object's instance variables or state.

# Accessor

```
public class Sprite{
    private double center_x;
    private double center_y;

    public Sprite(double x, double y){
        center_x = x;
        center_y = y;
    }
    public double getCenterX(){
        return center_x;
    }
    public void printLocation(){
        System.out.println("(" + center_x + ", " +
                               center_y + ")");
    }
}
```


accessor method: it allows client code to examine the data(variables) of the object.

printLocation is another accessor method


# Mutator

```
public class Sprite{  
    private double center_x;  
    private double center_y;  
  
    public Sprite(double x, double y){  
        center_x = x;  
        center_y = y;  
    }  
    ... // other code not shown.  
    public void setCenterX(double new_x){  
        center_x = new_x;  
    }  
  
    public void moveRight(){  
        center_x += 5;  
    }  
}
```

setCenterX is a mutator method: it allows client code to modify the data(variables) of the object.



moveRight() is an another mutator method



# Private Access

```
public class Sprite{
    private double center_x;
    private double center_y;

    public Sprite(double x, double y){
        center_x = x;
        center_y = y;
    }
    ... // other code not shown.
    public void setCenterX(double new_x){
        center_x = new_x;
    }

    public void moveRight(){
        center_x += 5;
    }
}
```

Note that the private variables `center_x` and `center_y` are still accessible everywhere in the `Sprite` class.

# Calling Methods

## Main.java

```
public class Main{
    public static void main(String[] args){
        Sprite player1 = new Sprite(20, 100);
        # accessing an object's data
        player1.printLocation(); # (20.0, 100.0)

        # modifying an object's data
        player1.moveRight();

        # accessing an object's data
        player1.printLocation(); # (25.0, 100.0)

    }
}
```

# Constructors

- **constructor**: Initializes the state/variables of new objects;  
**has the same name as the class.**

```
public type(parameters) {  
    statements;  
}
```

- runs when the client uses the `new` keyword
- no return type is specified;  
it implicitly "returns" the new object being created



# Example

Student.java

```
public class Student {  
    private int id;  
    private String name;  
    public Student(int i, String n) {  
        id = i;  
        name = n;  
    }  
}
```

object class



Main.java

```
public class Main {  
    public static void main(String[] args) {  
        // create two Student objects  
        Student s1 = new Student(321, "Sarah Jackson");  
        Student s2 = new Student(462, "Jim Smith");  
    }  
}
```

driver class



# Multiple constructors

- A class can have multiple constructors. These are overloaded constructors.
  - Each one must accept a unique set of parameters.

```
public class Student {  
    private int id;  
    private String name;  
    public Student(int i, String n) {  
        id = i;  
        name = n;  
    }  
    // randomizes the id variable.  
    public Student(String n) {  
        id = (int)(900 * Math.random()) + 100;  
        name = n;  
    }  
}
```

two parameters: an int  
and a string

one parameter: a string

# Client code

```
public class Main {  
    public static void main(String[] args) {  
        Student s1 = new Student(321, "Sarah Jackson");  
  
        // s2 has a randomized id.  
        Student s2 = new Student("Jim Smith");  
    }  
}
```

# Common constructor bugs

## 1. Re-declaring fields as local variables("shadowing"):

```
public Student(int i, String n) {  
    int id = i;  
    String name = n;  
    System.out.println(id); //prints the local id  
}
```

- This declares local variables with the same name as the instance variables, rather than storing values into the instance variables. The instance variables remain 0.

## 2. Accidentally giving the constructor a return type:

```
public void Student(int i, String n) {  
    id = i;  
    name = n;  
}
```

- This is actually not a constructor, but a method named `Student`

# Default Constructor

If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all integer fields to 0, booleans to false, Strings to **null**, etc...

However, if a class has at least one constructor(with or without parameters), this default constructor is overridden by the new constructor(s).

# Default constructor

Point.java

```
public class Point {  
    private int x;  
    private int y;  
    //no constructors  
}
```

Main.java

```
public class Main {  
    public static void main(String[] args) {  
        Point p1 = new Point();  
        // ok, uses default constructor  
  
        Point p2 = new Point(5, 2);  
        //error, no such constructor  
  
    }  
}
```

# Overriding Default constructor

Point.java

```
public class Point {  
    private int x;  
    private int y;  
    //override default constructor  
    public Point(int newX, int newY) {  
        x = newX;  
        y = newY;  
    }  
}
```

Main.java

```
public class Main {  
    public static void main(String[] args) {  
        Point p1 = new Point();  
        //error, no default constructor! (overridden)  
        Point p2 = new Point(5, 2);  
        // ok  
    }  
}
```

# An Implementation of Point

```
public class Point {  
    private int x;  
    private int y;  
    public Point() {  
        x = 0;  
        y = 0;  
    }  
    public Point(int newX, int newY) {  
        x = newX;  
        y = newY;  
    }  
    public void translate(int dx, int dy) {  
        x += dx;  
        y += dy;  
    }  
    public double distanceToOrigin() {  
        return Math.sqrt(x * x + y * y);  
    }  
}
```

Annotations:

- instance variables (points to `x` and `y`)
- overloaded constructors (points to `Point()` and `Point(int newX, int newY)`)
- mutator method (points to `translate`)
- accessor method (points to `distanceToOrigin`)



# OOP/OOD

**Object-Oriented Programming**(OOP) is a programming paradigm based on the concepts of objects data, in the form of instance variables and code, in the form of methods. Many popular languages are object-oriented(C++, Java, Javascript, Python).

In OOP, programs are made up of many objects and a program run is the interaction of these objects.

In **Object-Oriented Design** (OOD), programmers first spend time to decide which classes are needed and then figure out the data and methods in each class.

# Lab 1

Rewrite the Point class with **all** of the methods in this lecture. Make sure it has at least two constructors. Save it as Point.java. Then implement the driver class(Main.java) the main method by creating some Point objects using both constructors. Print out objects' data by accessing its fields and by calling its methods.

All of your variables can be **public for this part of the lab.**

**Note that this is the first time we are working with two different .java files in the same program.**

# Lab 1

Use the same repl as the previous Point class.

Write the Circle class. This class has the following **private** field variables(data/state): int x, int y, double radius. Include at least two constructors to initialize the variables.

It has the following instance methods: getArea(), boolean isInCircle(int a, int b), translate(int dx, int dy), tripleTheRadius().

Use the same driver class from the previous slide to test the Circle class. Create multiple Circle objects using all of the constructors and call and test all of the methods.

# Lab 2(Processing)

This lab requires Processing(<https://processing.org/>) for animation.

We will write the Sprite class which will represent a character object in a game.

This is only the first iteration, but we will build on this class to eventually be able to write a top-down view game like the original Legend of Zelda or a platformer game like Super Mario!

# Lab 2(Processing)

This lab requires Processing(<https://processing.org/>) for animation. We will write the Sprite class which will represent a character object in a game.

This is only the first iteration, but we will build on this class to eventually be able to write a top-down view game like the original Legend of Zelda or a platformer game like Super Mario!

# Lab 2(Processing)

The Sprite class has the following attributes:

**private** variables: `center_x` and `center_y` (used type float, in processing float is the default type for decimal values instead of double).

**public** variables: `change_x` and `change_y` (type float, for velocity components), `w` and `h` (type float, for width and height, don't use width/height as these are reserved) `image` (type `PImage` for the image of the Sprite object).

Download the starter code [here](#) and follow the directions given by the comments.

# References

1) Building Java Programs: A Back to Basics Approach by Stuart Reges and Marty Stepp

2) Runestone CSAwesome Curriculum:

<https://runestone.academy/runestone/books/published/csawesome/index.html>

For more tutorials/lecture notes in Java, Python, game programming, artificial intelligence with neural networks:

<https://longbaonguyen.github.io>