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

https://longbaonguyen.github.io

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.

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

declaring the **instance variables**

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

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

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

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

```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

```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) examine 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 void printLocation(){
    System.out.println("(" + center_x + "," +
                    center_y + ")");
  }
  public void moveRight(){
    center_x += 5;
  }
}
```
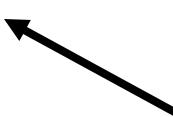
printLocation is an accessor method: it allows client code to examine the data(variables) of the object.

Mutator

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

  public Sprite(double x, double y){
     center_x = x;
     center_y = y;
  }
  public void printLocation(){
     System.out.println("(" + center_x + "," +
                            center_y + ")");
  }
  public void moveRight(){
     center_x += 5;
  }
}
```

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

Private Access

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

  public Sprite(double x, double y){
    center_x = x;
    center_y = y;
  }
  public void printLocation(){
    System.out.println("(" + center_x + "," +
                  center_y + ")");
  }
  public void moveRight(){
    center_x += 5;
  }
}
```
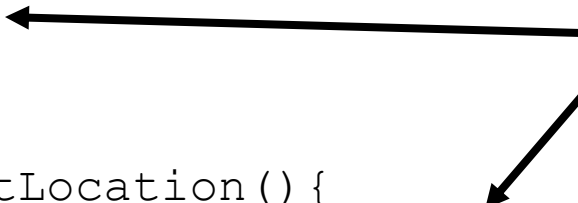
Note that the private variables center_x and center_y are still accessible everywhere in the Sprite class.

Calling Methods

Main.java

```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();



  }
}
```

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;
    }
}
```

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");
}
```

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"):

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

- 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:

```java
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

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

Main.java

```java
public class Main {
  public static void main(String[] args) {
    // ok, uses default constructor
    Point p2 = new Point();

    //error, no such constructor
    Point p1 = new Point(5, 2);
  }
}
```

Overriding Default constructor

Point.java

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

An Implementation of Point

```java
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);
    }
}
```

instance variables

overloaded constructors

mutator method

accessor method

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.

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(double x, double y), translate(double dx, double 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.

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