Lets Code!

Objects and Classes Part 1

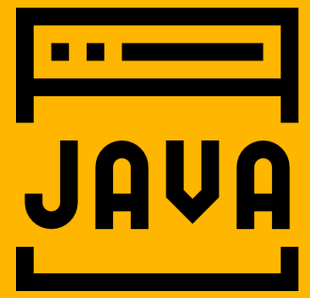
Instructor: Tariq Hook

You can find me on github @code-rhino

Key Terms

- Classes
- Encapsulation
- Private, Static, Final

- Constructors
- Mutators
- Accessors



Procedural Programming

What is procedural programming?

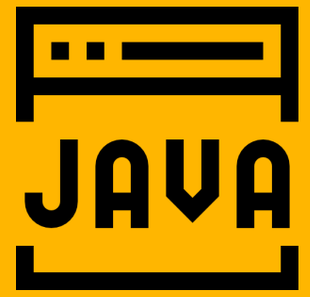
- Procedural (structured) programming consists of designing a set of procedures (algorithms) to solve a problem.
- The procedural paradigm suggests that a programmer will:
    - firstly, identify *which algorithms* to manipulate data
    - secondly, identify *which structures* use which algorithms.

Why do we use procedural programming?

- **Small** problems are easily resolved with a procedural implementation

Why do we not use procedural programming?

- Procedural implementations scale poorly.
- As the program grows in size, its complexity increases.
    - behaviors become tightly-coupled
    - debugging becomes more difficult
    - code-changes and maintenance becomes more difficult
    - testing a single aspect becomes nearly impossible



Why do we use OOP?

Why do we use OOP?

- **Large** problems can more be more easily scaled using the OOP paradigm.
- OOP allows users to break problem down into small logical objects
- OOP allows users to view code-details within the context of a specific object
- OOP allows users to more easily debug code
- OOP allows greater testability of code

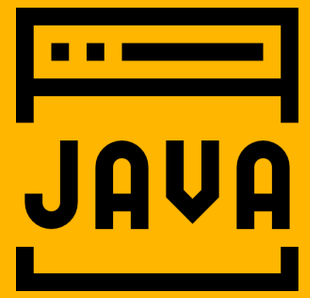
Object Oriented Programming (OOP)

- An object-oriented program is made of objects
- Each object has specific functionalities, which users access via the object's **methods**.
- The OOP paradigm suggests that a programmer will
    - firstly identify *which structures* to manipulate data
    - secondly identify *what algorithms* each structure will use

The 3 aspects of an object

- **Identity** - **What is its location?**
  - How is that object distinguished from other objects of the same type?
- **State** - **What does it store?**
  - What is the value of the internal objects this object contains?
- **Behavior** - **How does it act?**
  - What services or actions this object can perform?

Classes



Classes

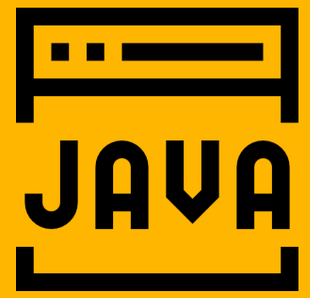
- A **class** is a template, or blueprint from which objects are made
    - it is the cookie-cutter to a cookie
    - it is the *classification* of an object.

Class naming conventions

- Class names must begin with a letter followed by any combination of letters, digits, and underscores.
    - By convention, class names start with a capital letter.
        - You cannot use a Java *reserved word* to name a variable or class.
- Whitespace is irrelevant to the Java compiler

Encapulsation

- Classes *encapsulate* several **data-fields** and behaviors into a single entity.
- **Encapsulation** is combining class-members (methods and variables) in a single scope.



Instance-Fields

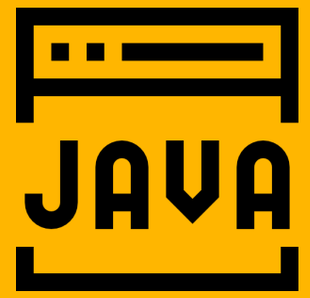
Instance-Fields

- An instance-field, or instance-variable are representative of the *properties* or *attributes* of a `Class`.
- By aggregating the values of an instance's fields, we derive the instance's *state*.

Encapsulation

- "Wraps" several data fields into a single entity

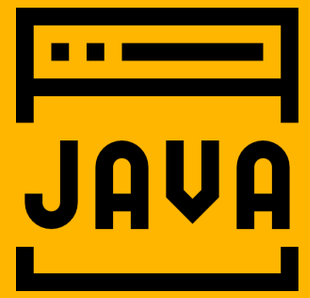
```
// class signature
public class Person {
    // instance variables (fields)
    private String name;
    private Integer age;
    private Boolean isFemale;

    // constructor
    public Person(String name, Integer age, Boolean isFemale) {
        this.name = name;
        this.age = age;
        this.isFemale = isFemale;
    }
}
```



Instance-Methods

Instance-Methods

- behaviors of an object are made available to users via the object's **methods**
- methods which are invoked on an **object** are **instance-methods**
- method-names should describe the intended behavior of the object
- methods of an object have hidden implementation
- methods describe a "can perform" relationship.

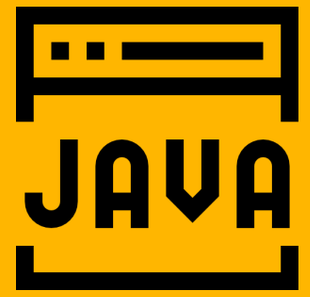


Instance-Variables

Instance-Variables (Fields)

- field-values of an object are made available to users via the object's **getters**
- variable-values of an object can be manipulated by the user via the object's **setters**
- the aggregation of an object's variable's values determines the object's **state**.
- fields describe a "has a" relationship.

Constructors



Constructors

- a method which creates a new instance of a class (an object)
- describes the initial **state** of an object

Constructor (Default)

```
public class Person { // class signature
    private String name; // instance variable

    public Person() { // constructor signature
    }
}
```

Constructor (Non-Default)

```
public class Person { // class signature
    private String myName; // instance variable

    public Person(String name) { // constructor signature
        this.myName = name; // setting instance variable
    }
}
```

Multiple Constructors

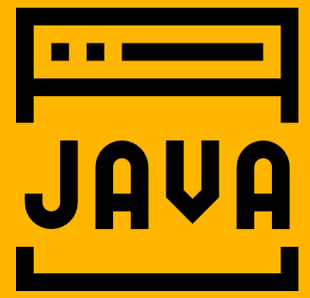
```
public class Person { // class signature
    private String myName; // instance variable

    // no-arg (default) constructor
    public Person() { // constructor signature
        this.myName = "Tariq"; // setting instance variable
    }

    public Person(String name) { // constructor signature
        this.myName = name; // setting instance variable
    }
}
```

Assigning initial values From Constructor

```
public class Person { // class signature
    private String myName; // instance variable
    private Character gender; // instance variable

    // no-arg constructor
    public Person() { // constructor signature
        this.myName = "Tariq"; // setting instance variable
        this.myGender = 'M'; // setting instance variable
    }

    public Person(String name, Character gender) { // constructor signature
        this.myName = name; // setting instance variable
        this.myGender = gender; // setting instance variable
    }
}
```

Calling Constructors From Constructors

```
public class Person { // class signature
    private String myName; // instance variable
    private Character myGender; // instance variable

    // no-arg constructor
    public Person() { // constructor signature
        this("Tariq", 'M'); // nested constructor call
    }

    public Person(String name, Character gender) { // constructor signature
        this.myName = name; // setting instance variable
        this.myGender = gender; // setting instance variable
    }
}
```



Getters and Setters

Setters (Mutators)

```
public class Person { // class signature
    private String myName; // instance variable

    public Person(String name) { // constructor signature
        this.myName = name; // setting instance variable
    }

    public void setName(String differentName) { // method signature
        this.myName = differentName; // setting instance variable
    }
}
```

Setters (Mutators)

```
public class Person { // class signature
    private String myName; // instance variable

    public Person(String name) { // constructor signature
        setName(name); // BAD DESIGN; method can be overriden
    }

    public void setName(String differentName) { // method signature
        this.myname = differentName; // setting instance variable
    }
}
```

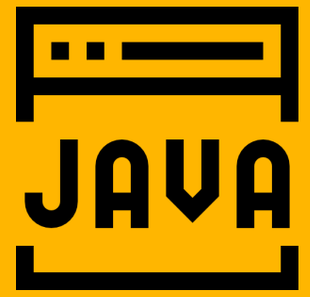
Getters (Accessors)

```
public class Person { // class signature
    private String myName; // instance variable

    public Person(String name) { // constructor signature
        this.myName = name; // setting instance variable
    }

    public void setName(String differentName) { // method signature
        this.myname = differentName; // setting instance variable
    }

    public String getName() {
        return this.myName;
    }
}
```

Wrap Up

- Relationships between Classes
- Objects and Instances
- Single Responsibilty
- Private / Static / Final

- Procedural Programming
- Classes
- Encapsulation
- Methods
- Constructors
- Mutators



Keep Coding !!!

Clean Code is Happy Code