Classes and Encapsulation

Today's Objectives

- Classes
  - What are they?
  - How do we use them in Object Oriented Programming (OOP)?
- Proper class definition
- Create and call Constructors
- Access modifiers: public vs private
- Create an instance of a class
- Overloading, as it relates to classes

Three Fundamental Principles of OOP

- **Encapsulation**: the concept of hiding values or state of data within a class, limiting the points of access

- **Polymorphism**: the ability for our code to take on different forms

- **Inheritance**: the practice of creating a hierarchy for classes in which descendants obtain the attributes and behaviors from other classes

Benefits of OOP

- A natural way to express real-world objects in code
- Modular and reliable, allowing changes to be made in one part of the code without affecting another
- Discrete units of reusable code
- Units of code can communicate with each other by sending and receiving messages and processing data

Classes

- A **class** is a blueprint to create an object
  - Specifies **state**/variables
  - Defines **behavior**/methods
- Class Naming
  - Use singular nouns, not verbs
  - Class must match the file name
  - Use Pascal casing
  - A Fully Qualified Name is unambiguous and includes the package and class name

What is Pascal Case?

- A subset of Camel Case where the first letter is capitalized.
  - Camel Case: **`userAccount`**
  - Pascal Case: **`UserAccount`**

- Use Camel Case for variable names.

- Use Pascal case for Class names and Constructors.

Instance Variables

Instance variable represent the **properties** of a class.

- Each instance of a class will have its own instance variables that represent its internal **state**.

- Instance variables are declared with **access modifiers**.
  - `public`
    - Can be accessed by any other object.
  - `private`
    - Can only be accessed by the current instance of a class.

Encapsulation Using Instance Variables

Encapsulation is the concept of hiding data and controlling access to it.

- Letting other code modify data in an instance can be dangerous because it means an instance is not in control of its internal state.
- Hiding code implementation allows other classes to use a class without knowing anything about how it works
- By declaring instance variables **`private`**, we prevent other code from accessing instance variables directly.
- We use **getters** and **setters** to provide access to internal data to external code.

Goals of Encapsulation

Encapsulation is the concept of hiding data and controlling access to it.

- Encapsulation makes code extendable.
- Encapsulation makes code maintainable.
- Encapsulation promotes "loose coupling."
    - A **loosely coupled** system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components.

Constructors

Every class has a **constructor** which is called when an object is being created.

- Constructors are defined similarly to a method but have
  - The same name as the class
  - No return types

```
public ScratchPad() {

}
```

Constructors

- To create an object, code must call at least one constructor.
- Java provides a built-in no-argument constructor by default so that each class is not required to provide one to allow basic object creation.
- A class may declare alternate constructors with arguments.
- However, as soon as one or more constructors is explicitly declared in the class, the default constructor is no longer available so if the class needs to allow creation of objects with no arguments, the class must explicitly declare a no-argument constructor to replace the default constructor which is no longer available.

Constructors

```
public class ScratchPad
{

}
```

Here, the **`ScratchPad`** class does not explicitly declare a constructor.

```
ScratchPad pad = new ScratchPad()
```

By default, Java provides a no-argument constructor so even though no constructor is explicitly declared for the **`ScratchPad`** class, we can still create a **`ScratchPad`** object using a constructor with no arguments.

Constructors

```
public ScratchPad(String text) {
    // some code
}
```

Here, the **`ScratchPad`** class declares a constructor which takes a **`String`** argument.

We can now create a **`ScratchPad`** object using this constructor.

```
// Valid
ScratchPad pad = new ScratchPad("Test text");

// No longer valid
ScratchPad pad = new ScratchPad();
```

However, because we have declared our own constructor the default no-argument constructor is no longer available so **this is NOT valid**.

Constructors

```
public class ScratchPad {

    public ScratchPad() {

    }

    public ScratchPad(String text) {
        // some code
    }
}
```

Here, the **`ScratchPad`** class declares a constructor which takes a **`String`** argument but also explicitly declares a no-argument constructor to replace the default constructor which is no longer available because the class has declared its own constructor.

Since we have added our own no-argument constructor we can once again create a **`ScratchPad`** object using a no-argument constructor.

```
ScratchPad pad = new ScratchPad()
```

Objects and this

- The **`this`** keyword is used to refer to the current object.
- We can use **`this`** to avoid name collisions.

```
public class ScratchPad {

    private String text;
    private String data;

    public ScratchPad(String text, String sampleData) {
        this.text = text;
        data = sampleData;
    }
}
```

The constructor argument **`text`** has the same name as one of the instance variables.

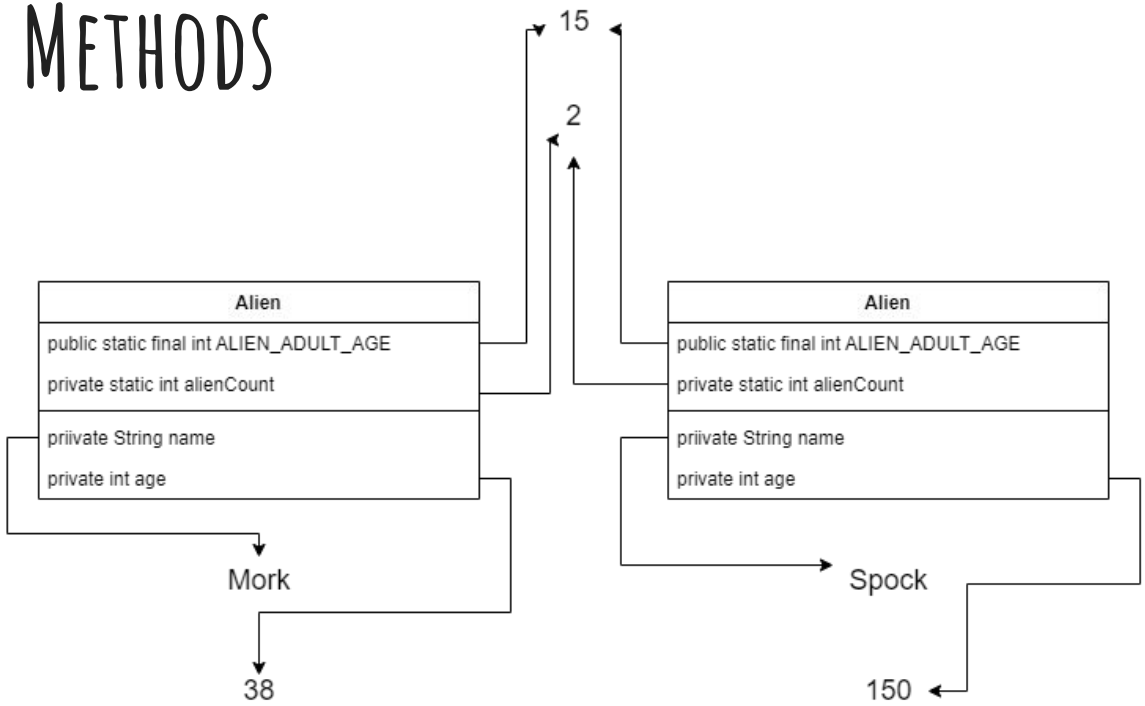
We use the **`this`** keyword to differentiate between the instance variable and the constructor argument.

Because the argument being assigned to **`data`** has a different name, the **`this`** keyword is not required.

Static Attributes & Methods

Static Attributes & Methods

- Attributes which are marked **`static`** share one value across all instances.
- Attributes and methods which are marked **`static`** are accessed using the class name rather than an object.




```
System.out.println(Alien.ALIEN_ADULT_AGE);  // static attribute
System.out.println(Alien.getAlienCount());  // static method
mork.sayMyName();  // instance method
```