Please Complete The Socrative Pulse Survey

URL: gosocrative.com

Room Name: JAVAGOLD

Objectives

- Review
- Classes

Review

- Maps are data structures with key-value pairs
  - Duplicate keys are NOT allowed
    - Keys are held in a set (see above)
  - Duplicate values are allowed
  - Access to values is through keys
    - A key can not be retrieved from the value
  - Iteration through a Map requires iterating through the keys
  - Commonly used
    - put(), get(), keySet()

```
Map<String, Integer> myMap = new
Map<String, Integer>();

myMap.put("George", 12);
Set<String> keys = myMap.keySet();
```

- Sets are data structures two main attributes:
  - Unordered (no index or ‘get’ method)
  - Contain no duplicates
  - Not commonly used
    - add(), contains()

```
Set<Integer> mySet = new
Set<Integer>();

mySet.add(42);
boolean b = mySet.contains(42);
```

Data Structures: Case Scenario 1

- Becky works as a restaurant manager and constantly has new food orders getting placed. The orders need to be processed in the order they were placed to make sure everyone gets their food on time.

  Which data structure would you tell Becky to use to take and process orders and why?

Array	List<T>	Stack<T>	Queue<T>	Set<T>	Map<K, V>

Data Structures: Case Scenario 2

- Jennifer is coding a phone book program that will store her friend's names and their addresses.

  Which data structure would you tell Jennifer to use and why?

Array	List<T>	Stack<T>	Queue<T>	Set<T>	Map<K, V>

Data Structures: Case Scenario 3

- Luis is managing a large gala event and needs a way to quickly identify if someone’s invitation id is on the guest list.

  Which data structure would you tell Luis use to identify who is a guest and why?

Array	List<T>	Stack<T>	Queue<T>	Set<T>	Map<K, V>

Data Structures: Case Scenario 4

- Janelle is adding code to her game where new, generic enemy objects are constantly being added and removed.

  Which data structure would you tell Janelle to use for the enemies and why?

Array	List<T>	Stack<T>	Queue<T>	Set<T>	Map<K, V>

Data Structures: Case Scenario 5

- John is a teacher trying to sort a very large, fixed number of letter (char) grades in ascending order.

  Which data structure would you tell John to use and why?

Array	List<T>	Stack<T>	Queue<T>	Set<T>	Map<K, V>

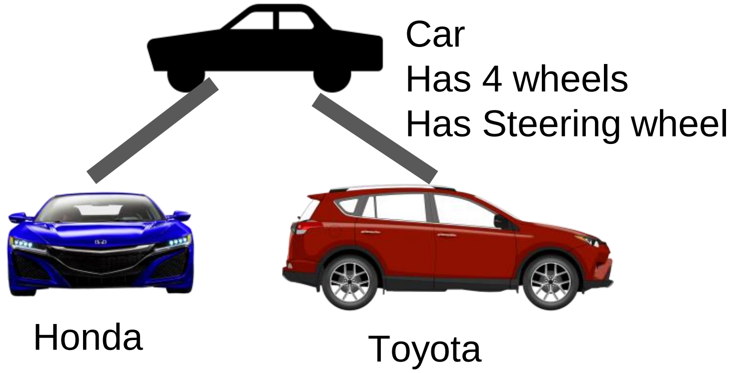
3 Fundamental Principles of Object Oriented Programming (OOP)

Polymorphism - the ability for our code to take on different forms. In other words, we have the ability to treat classes generically and get specific results.

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

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

The key starts the car. The Ignition system is hidden from the user turning the key to start the car.

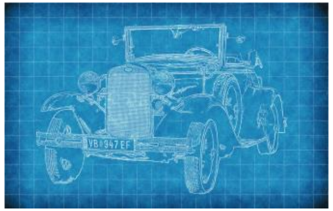


Benefits of Object Oriented Programming (OOP)

- Benefits of OOP are:
  - A natural way of expressing 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
- We will be talking about these more over the next week and a half

Classes

Classes are blueprints to create objects.



A **class** is like a blueprint, it is not the thing you're building, but it describes what you're building

From a class, we can create as many objects of that class we need. **These objects are instances of the class.**

Objects: Properties and Methods

Objects have properties and methods.

Properties of an object are variables that define an object's **state**.

- Number of doors
- Color
- Engine size

Objects also have methods. Methods define the **behavior** of the object.

- Drive
- Switch Gears
- Turn On Headlamp

Class Declaration

Here are the basics on how to declare a class:

```
package te.mobility;

public class Car {
        // body of class
}
```

By convention, class names always start with a capital letter.

Properties: Declaration

Classes have properties. Let’s consider the Car class.

We have declared some properties but not initialized them to anything, they will have default values.

We have declared some properties and initialized them to some values.

```
class Car {
    private String color;
    private Double engineSize;
    private int numberOfDoors;
}
```

```
class Car {
    private String color = "green";
    private double engineSize = 1.5;
    private int numberOfDoors = 2;
}
```

Methods: Declaring

Car.java

```
package te.mobility;

public class Car {
    private String color = "green";
    private boolean engineOn = false;

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public void **goInReverse()** {
        System.out.println("going backwards.");
    }
}
```

Driver.java

```
package te.main;

import te.mobility.Car;

public class Driver {

    public static void main (String args[]) {

        Car shinyNewCar = new Car();
        shinyNewCar.goInReverse();

    }
}
```

Method Signature

- All methods have a name

  usually a verb or verb phrase that describes the action
- All methods have a return type

  what the method will return

  could be anything: Rectangle, boolean, ***void (returns nothing)***
- Methods can be parameterless or can include parameters (or inputs).

public <returnType> name (<List of Arguments>)

```
public boolean isLargerThan(int length, int width) {
    return (this.length * this.width) > (length * width);
}
```

Method Overload

Overloaded methods are methods with the *same name and return type*, and a *different set of parameters*. Java uses the correct overload based on the parameters sent to it.

```
ourRectangle.isLargerThan( otherRectangle );
```

```
public boolean isLargerThan( Rectangle other) {
                return this.area > other.getArea();
}
```

```
ourRectangle.isLargerThan( 10, 12 );
```

```
public boolean isLargerThan( int width, int height ) {
                return this.area > width * height;
}
```

this keyword

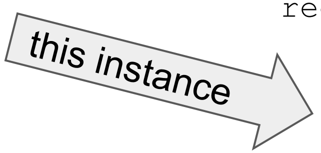
This ***this*** keyword refers to the member variable specific to the instance of an object where the code is run.

```
public class Car {

    private String color;

    public String getColor {
        return this.color;
    }

    public String setColor(String color) {
        this.color = color;
    }

}
```

```
Car blueCar = new Car();
blueCar.setColor( "Blue" );
```

this instance

```
Car redCar = new Car();
redCar.setColor( "Red" );
```



Methods: Getters and Setters

Getters and Setters are special types of methods.

- Data members should **always be private**.
- Access to properties will be provided via getter and setter methods.
- Getter methods allow the outside world to retrieve the value of the data member.
- Setter methods allow the outside world to set the value of the data member.

This practice is known as encapsulation.

Methods: Getters and Setters

Here, a getter and setter have been created for the color data member:

```
package te.mobility;

public class Car {
    private String color = "green";

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}
```

This is a getter, it simply returns the value of the property.

This is a setter, it takes 1 parameter, which will be used to update the property’s value

“this” is used to differentiate the property from the parameter passed in.

Methods: Getters and Setters

Consider the CarSalesMan class. It can now call the getter method to obtain the color,and the setter method to change the car’s color.

Car.java

```
package te.mobility;

public class Car {
    private String color = "green";

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}
```

CarSalesMan.java

```
package te.main;
import te.mobility.Car;

public class CarSalesMan {

    public static void main(String args[]) {
        Car thisCar = new Car();
        System.out.println(thisCar.getColor());  // green

        thisCar.setColor(“blue”);
        System.out.println(thisCar.getColor());  // blue
    }
}
```

Derived Properties

A derived property is a **getter** that, instead of returning a member variable, returns a *calculation* taken from member variables. If we have `firstName` and `lastName`, we don't need to also store `fullName`, we can derive it from what we already have.

```java
package com.techelevator;


public class Rectangle {


    private int length;
    private int width;

    public int getLength() {
        return length;
    }
    public void setLength(int length) {
        this.length = Math.abs(length);
    }
    public int getWidth() {
        return width;
    }
    public void setWidth(int width) {
        this.width = Math.abs(width);
    }

    public int getArea() {

        return this.length * this.width;

    }

}
```

Constructors

Constructors are “method like constructs” in Java designed to help instantiate an object of a class.Consider the following declaration:

```
Car shinyNewCar = new Car();
```

Every class has a default constructor that takes no parameters, in this case it’s Car().

The new keyword instantiates an object and creates space for it in memory.

Constructors Declaration

Custom constructors can be declared following this pattern:

Name of The Class (**parameter1**, **parameter2**) {

… // body of constructor }

Two rules to have in mind:

- The constructor has no return type.
- The constructor’s name must be identical to the class name.

Constructors Declaration Example

A custom constructor with 2 parameters has been created for Car:

```
package te.mobility;

public class Car {
    private String color = "green";
    private int numOfDoors = 4;

    public Car(String color, int numberOfDoors) {
        this.color = color;
        this.numOfDoors = numberOfDoors;
    }
}
```

Constructors Declaration Example

Having defined a constructor in this manner allows for car to be instantiated by providing two parameters.

```
package te.mobility;

public class Car {
    private String color = "green";
    private int numOfDoors = 4;

    public Car(String color, int numberOfDoors) {
        this.color = color;
        this.numOfDoors = numberOfDoors;
   }
}
```

```
package te.main;

import te.mobility.Car;

public class CarSalesMan {

    public static void main(String args[]) {
        Car thisCar = new Car("blue", 4);
    }
}
```

We have now instantiated a blue car with 4 doors.

`public class Car {` `private String color;`	**Default Constructor:** `Car myCar = new Car();`
`public class Car {` `private String color;` `public Car( String color ){` `this.color = color;` `}`	Constructor with argument. The argument value must be passed to instantiate the Car object. `Car myCar = new Car( “red” );`
`public class Car {` `private String color;` `public Car() { }` `public Car( String color ){` `this.color = color;` `}`	No-Argument Constructor with a constructor Overload that allows a value to set the starting state at instantiation. `Car myCar = new Car();` `Car myCar = new Car( “red” );`

Definition of Static in Java

If a method or data member is marked as static, it means **there is exactly one** copy of the method, or one copy of the data member shared across all objects of the class.

One way to think about this, is that the static member is a unique property of the "blueprint" that is the same for all objects created from that blueprint.

FordCar class might have a static data member logo. All FordCar objects will share the same static data member.

The non-static methods and data members we have defined so far are often called Instance members or Instance methods.

Static Members: Declaration

Static members and methods are declared by adding the keyword static.

Be cautious of using static for methods and variables.

```
public class Car {
    public static String carBrand = "Ford";

    public static void honkHorn() {
        System.out.println("beeep?");
    }
}
```

Static: Rules

```
String someInstanceVariable;

public static void someStaticMethod() {
        System.out.printlnString (someInstanceVariable);
        someInstanceMethod();
}

public void someInstanceMethod() {

}
```

This is an instance (non-static data member)

We are inside a static method, but we are referencing an instance member, which is not allowed

We are inside a static method, but we are calling an instance method, which is not allowed.

You have encountered this issue before - recall that any method directly called by public static void main had to also be a static.

Static: Constants

Constants are variables that cannot change. The closest thing to a constant in Java is declaring a data member with **static final**.

```
public class Car {
		public static final String CAR_BRAND = "Ford";
...
}
```

Attempts to change the value of this data member will result in an error. This, for example is invalid:

```
public class CarDealership {

		public static void main(String args[]) {

				Car.CAR_BRAND = "GM";
		}
}
```

Static

Reasons why we'd want to use *static* fields:

• when the value of the variable is independent of objects
• when the value is supposed to be shared across all objects

Reasons why we'd want to use *static* methods:

• to access/manipulate static variables and other static methods that don't depend upon objects.
• *static* methods are widely used in utility and helper classes.

For more information about the static keyword in java, this was the resource I referenced for this slide

Encapsulation

- Makes code extendable
- Made code maintainable
- Promotes “loose coupling”
  - Each of its components has or makes use of little or no knowledge of the definitions of other separate components

Encapsulation

Rule: instance variables (properties, data members) are private and methods are public

```
public class Car {
            private int year;

            public void setYear(int year) {
                this.year = year;
            }
            public int getYear() {
                return this.year;
            }
}
```

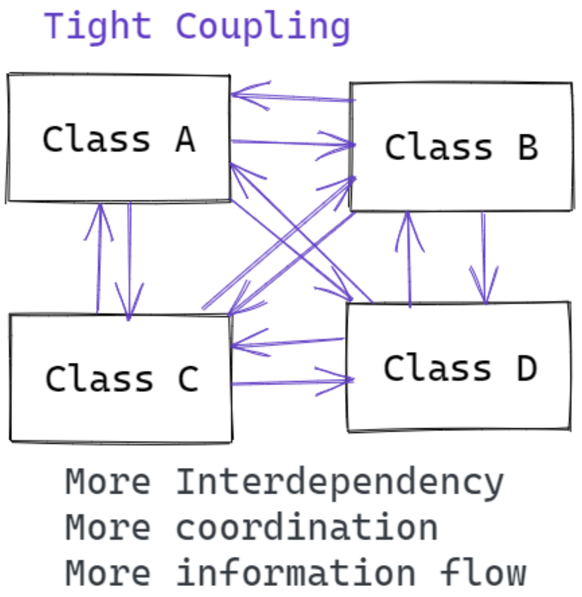
private can only be accessed or used inside the Car class

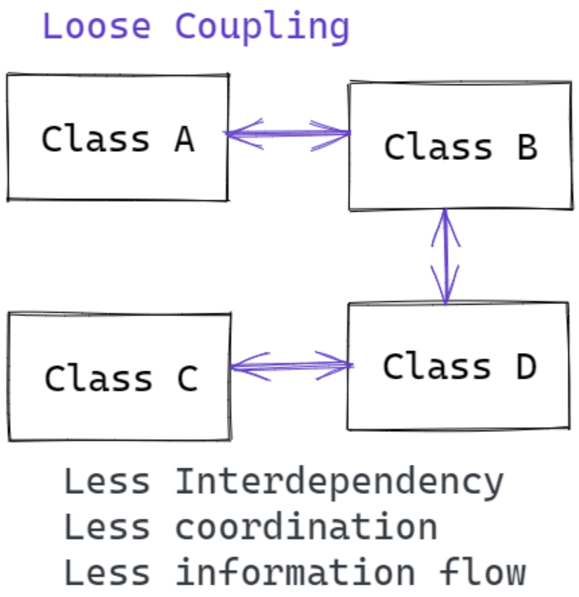
public means this method can be called outside of this class

Loose vs Tight Coupling

Coupling

Coupling refers to the degree of direct knowledge each component of a system or application has of elements that it needs to use. ***Loose Coupling*** is an approach to connect components that depend on each other with the least knowledge possible. Encapsulation promotes Loose Coupling.





Encapsulation & Data Hiding

- **Encapsulation** is the process of combining related data members and methods into a single unit.
    - In Java, encapsulation and data hiding are achieved by putting all related data members and methods in a class.

- **Data hiding** is the process of obscuring the internal representation of an object to the outside world.
    - In Java, data hiding is achieved by setting all members to private and providing getters and setters for said members.

Access Modifiers

Access modifiers control who can access a variable or method.

public	Can be accessed by anyone
private	Can only be accessed from inside the Class.

Getters and Setters

Getters and Setters allow public access to a private member variable while still allowing the class to have full control of the variable.

member variables

```java
private String taskName;
private boolean complete;
```

Getter

```java
public String getTaskName() {
    return this.taskName;
}
```

Setter

```java
public void setTaskName( String taskName ) {
    this.taskName = taskName ;
}
```

Getter

```java
public boolean isComplete() {
    return this.complete;
}
```

Setter

```java
public void setComplete( boolean complete ) {
    this.complete = complete;
}
```

Summary of Class Components

```
package te.mobility;

public class Car {
    private String color = "green";
    private int numOfDoors = 4;
    private int fuelRemaining = 5;
    private int totalFuelCapacity = 10;

    public Car(String color, int numberOfDoors) {
        this.setColor(color);
        this.setNumOfDoors(numberOfDoors);
    }

    public void goForward() {
        System.out.println("going forward");
    }

    public double fuelRemaining() {
        return fuelRemaining/total fuel capacity * 100.0;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public int getNumOfDoors() {
        return this.numOfDoors;
    }

    public void setNumOfDoors(int numOfDoors) {
        this.numOfDoors = numOfDoors;
    }
}
```

These are the properties for the class. (a.k.a instance variables.)

This is a constructor that takes two arguments.

These are methods of the class that perform a task.

These are getters and setters for the two of the data members.

Summary Of Concepts

- Classes are a grouping of variables and methods
  - Variables define the state
  - Methods represent the behavior

- Objects are instances of the class
  - A copy of the variables and methods in memory defined by the class

- Encapsulating a class means converting all instance variables to private and creating methods (getters and setters) to control access to the object’s variables.
  - This helps to decouple the object from other objects that use/refer to it.