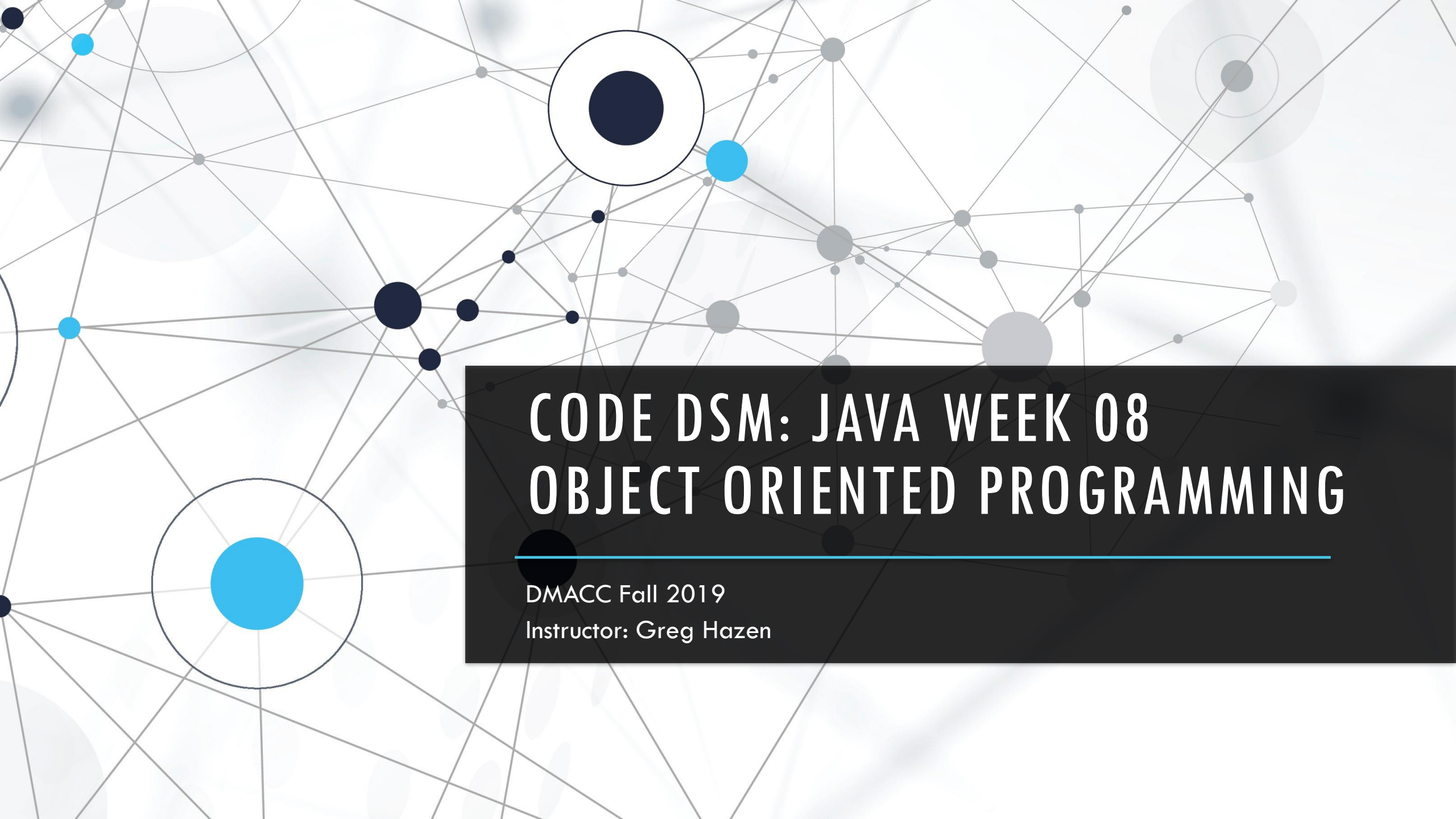


CODE DSM: JAVA WEEK 08 OBJECT ORIENTED PROGRAMMING

DMACC Fall 2019

Instructor: Greg Hazen

AGENDA REVIEW

WEEK 7 QUIZ

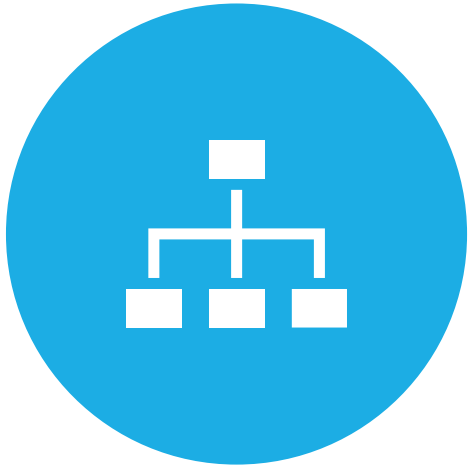
WEEK 7 ASSIGNMENT

OBJECT ORIENTED PROGRAMMING

ENCAPSULATION

AGENDA

ABSTRACTION



INHERITANCE

PROCEDURAL PROGRAMMING

So far we've been doing procedural programming

All written in one file

Instructions run top down

- Enter main
- Do stuff
- Exit

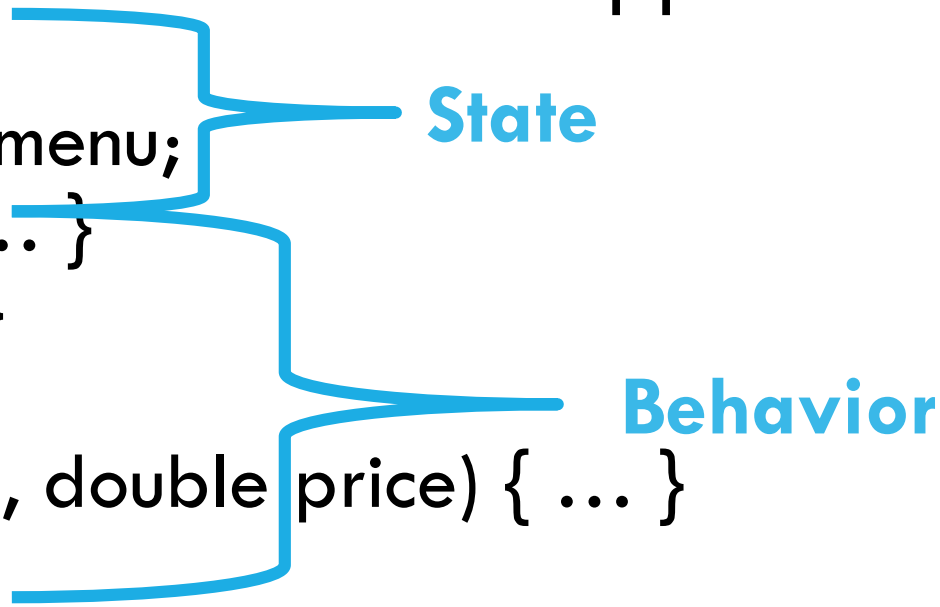
OBJECT ORIENTED PROGRAMMING

Switching to an Object Oriented mindset

- Create a name that describes a concept
  - Restaurant Menu
- List any state that it needs to keep track of
  - Items, Prices
- List behaviors that other can use
  - Add to Menu, Order from Menu
- List behaviors that others don't need to know
  - Load from File, Save to File, Handle if File doesn't Exist…

OBJECTS

```
public class Menu { // Class names start with Uppercase
        File file;
        HashMap<String, Double> menu;
        void loadMenuFromFile() { … }
        void saveMenuToFile() { … }
        void showMenu() { … }
        void addToMenu(String item, double price) { … }
}
```



LET'S REVIEW THE WORDS WE'VE LEARNED…

Class describes the state and behavior of a concept

Objects are an **instance** of a class

- We can have multiple **instances** of a class

Variable stores a value or instance of an object

Instance Variable stores a value specific to an instance

- breakfastMenu.file and dinnerMenu.file return two different files

INSTANCE VARIABLES

```
public class Menu {
    File file;
```

file is an **instance variable** or **member variable,** so *breakfastMenu.file; and dinnerMenu.file;* return two completely separate values

METHODS

```
public class Menu {
    void showMenu() { … }
```

Methods can access the **instance variables**, so breakfastMenu.showMenu() and dinnerMenu.showMenu()

- Use the same behavior, but
- Use completely separate menu values

CONSTRUCTOR

Called when you create a *new* instance of a class

```
new Menu(); // calls the constructor
public class Menu {
	Menu() { // Constructor
		// Do something when instance is created
	}
}
```

CONSTRUCTOR

If you don't create a constructor, the class has a default constructor

```
public class Menu {
    Menu() {}
}
```

‘THIS’ KEYWORD

Use ‘*this*’ to access the instance variable

```
String meal;
Menu(String meal) {
    this.meal = meal;
}
```

‘THIS’ KEYWORD

You can have multiple constructors

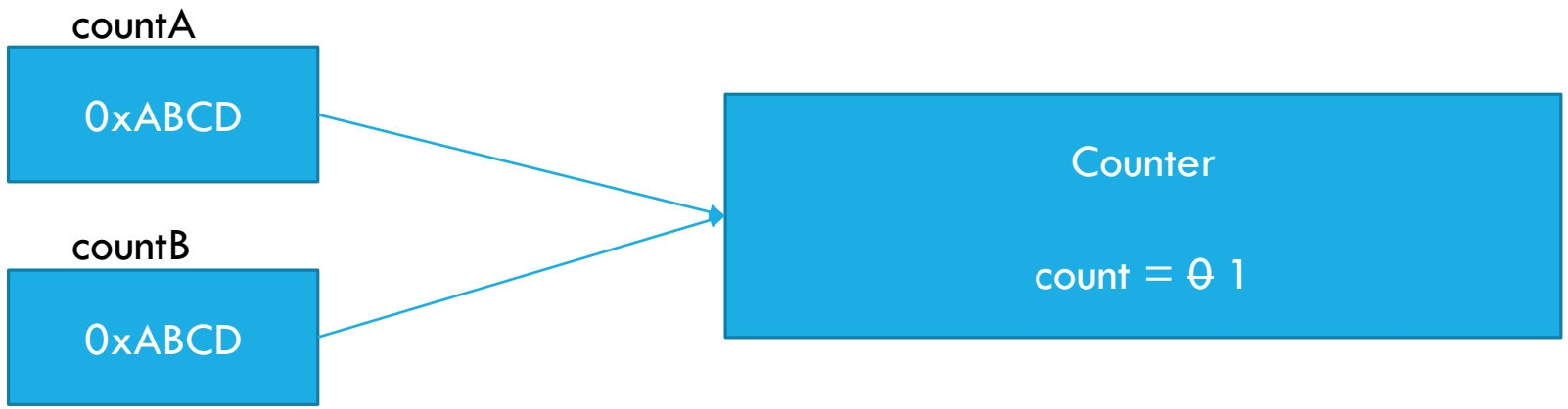
```
public class Counter {
        private int count;
        public Counter() {
                this(0);
        }
        public Counter(int initialCount) {
                this.count = initialCount;
        }
}
```

SHARING REFERENCES TO AN OBJECT

When you update through one reference, both are updated because they point to the same reference

countA.increment();

countB.getCount(); // returns 1



PROBLEM WITH EXPOSING THE DETAILS

What if another object emptied our menu?

What if we changed to using a database instead?

PRIVATE VS ENCAPSULATION

mood, hungry and energy are the **state** of the cat

The fact they're private is not considered **Encapsulation**

```
public class Cat {
    private int mood;
    private int hungry;
    private int energy;

    private void meow() { … }

    public void feed() {
        hungry--;
        mood++;
        meow();
    }
}
```

Adapted from (Petkov, 2018)

PRIVATE VS ENCAPSULATION

mood, hungry and energy are the **state** of the cat

public feed() interacting with the cat is **Encapsulation**

```
public class Cat {
    private int mood;
    private int hungry;
    private int energy;

    private void meow() { … }

    public void feed() {
        hungry--;
        mood++;
        meow();
    }
}
```

Adapted from (Petkov, 2018)

PRIVATE VS ENCAPSULATION

Encapsulation hides things inside a single unit

Method *feed()* is a single unit

Class C*at* is also a single unit

```
public class Cat {
    private int mood;
    private int hungry;
    private int energy;

    private void meow() { … }

    public void feed() {
        hungry--;
        mood++;
        meow();
    }
}
```

Adapted from (Petkov, 2018)

PRIVATE VS ENCAPSULATION

Just because a variable is private doesn't mean it's **Encapsulation**

Encapsulation is the binding of a public method and the hidden state/behavior in a single unit

ENCAPSULATION VS ABSTRACTION

Nearly inseparable and very hard to describe the difference

- **Abstraction** is the **design** of *how objects interact*
- **Encapsulation is** the **implementation** of *how objects interact*

ENCAPSULATION VS ABSTRACTION

Abstraction = Data Hiding + Encapsulation

ABSTRACTION

Occurs when the caller is able to know the goals of an object without knowing the means

An object is an abstraction if:

- only high-level methods of interaction are provided
- the implementation details are hidden from the caller

ABSTRACTION

Abstraction helps us reduce the complexity of systems

As the system grows in size, there may be hundreds of objects

No need to know everything going on in the system, just enough to accomplish the solution

ABSTRACTION EXAMPLE: PAYROLL SALARY

Salary
- Paid a fixed amount
- May not earn overtime
- Always receives benefits

Hourly
- Paid variable amount based on hours worked
- May earn overtime
- May receive benefits

Contractor
- Paid a fixed amount based on contract
- May not earn overtime
- Never receives benefits

ABSTRACTION EXAMPLE: PAYROLL

Together, let's create an abstraction that works for

- Salary
- Hourly
- Contractor

What other abstractions could be made?

ABSTRACTION EXERCISE

- Provide a procedure for baking a cake
- Provide a procedure for baking a pastry
- Provide a procedure for brewing a coffee

Create one or more abstractions for all three procedures

LEAKY ABSTRACTION

An abstraction is 'leaky' if the caller has to know something about the implementation of a method

Almost all abstraction is leaky in some way

Leaks occur

- If the caller has to know which specific input to provide to get a certain result
- If the caller has to know how to deal with specific types of the resulting output

LEAKY ABSTRACTION

Causes tight coupling between objects

- It becomes impossible to reuse the objects without using them all together

Makes the caller responsible for error conditions and performance

Examples:

- Having to know the correct order to iterate through a two-dimension array
- Performance being dependent on the order of operations to be called
- Error conditions that require knowing what a specific code means internal to the called object in order to handle them

LEAKY ABSTRACTION

Fixing a leaky abstraction is not an easy, straightforward process

- Identify the leak
- Create a way to encapsulate anything that might be inadvertently exposed
- Change the public interface to hide the means for arriving at the solution

Let's look for leaky abstractions in our payroll example

OBJECT ORIENTED PROGRAMMING

We've seen

- Encapsulation
- Abstraction

Next, Inheritance. A tool in OO to help us **abstract** out concepts.

INHERITANCE

Inheritance describes an ‘is-a’ relationship

- A Car *is a* Vehicle
- An ElectronicPayrollSystem *is a* PayrollSystem
- An ArrayList *is a* List *is a* Collection

Using this ‘is-a’ style relationship allows us to make type hierarchies

INHERITANCE

A relationship between a more general class and a more specialized class

The general class is an **abstraction** of one or more specialized classes

- General class is the **superclass or parent**
- Specialized class is the **subclass or child**

INHERITANCE

Substitution Principle

- Can replace any **superclass** with its **child**

Vehicle vehicleA = new Car();

Vehicle vehicleB = new Boat();

INHERITANCE

Substitution Principle

- This makes things more reusable

```
List<Vehicle> vehicles = new ArrayList<>();
vehicles.add(new Car());
vehicles.add(new Boat());
```

INHERITANCE

A **subclass** 'extends' a superclass

```
class Car extends Vehicle { … }
class Boat extends Vehicle { … }
```

INHERITANCE

The following will print “On”

```
Car car = new Car();
car.turnOn();
```

```
class Vehicle {
    public void turnOn() {
        System.out.println(“On”);
    }
}

class Car extends Vehicle {
}
```

INHERITANCE

A **subclass** inherits all methods

Subclass can **override** a method by using the same method signature and providing a new implementation

INHERITANCE

The following will print “Vroom”

```
Car car = new Car();
car.turnOn();
```

```
class Vehicle {
    public void turnOn() {
        System.out.println(“On”);
    }
}

class Car extends Vehicle {
    @Override
    public void turnOn() {
        System.out.println(“Vroom”);
    }
}
```

INHERITANCE

Access Modifiers

- public: any object can modify it
- private: can only modify it from inside the instance
- No Modifier: classes in the same folder/package can modify it; **not subclasses**
- **protected: any subclass or any class in the same folder/package can modify it**

INHERITANCE

Private variables of a superclass are inaccessible

Either use

- protected to make them available
- or create a method to access them

INHERITANCE

The following will print “Vroom”

```
Car car = new Car();
car.turnOn();
```

```
class Vehicle {
    protected String onAction = “On”;
    public void turnOn() {
        System.out.println(onAction);
    }
}

class Car extends Vehicle {
    public Car() {
        onAction = “Vroom”;
    }
}
```

EXERCISE

1. Create a Vehicle class with methods
   a. public int milesLeft() // returns 0 by default
   b. public void drive(int miles) // subtract from miles left
2. Create TeslaModelS class that extends Vehicle
   a. Add *public void charge(int hours) //add hours*11.5 to miles*
3. Create GeoPrizm class that extends Vehicle
   a. Add *public void fillGas(int gallons) //add gallons*32 to miles*
4. *Simulate both cars charging/filling up and driving*

‘SUPER’ KEYWORD

Sometimes, we want to interact with the parent implementation of a method

- 'super' allows you to directly access the superclass
- In Vehicle exercise, we could have used

  super.milesLeft

'SUPER' KEYWORD

Use 'super' to call a constructor from a child class

super must be the first statement in the constructor

```
class Vehicle {
    private String name;
    public Vehicle(String name) {
        this.name = name;
    }
}
class Car extends Vehicle {
    private int milesPerGallon;
    public Car() {
        super("Greg's Car");
        this.milesPerGallon = 32;
    }
}
```

‘SUPER’ KEYWORD

Use ‘this’ if you need to interact with something from the subclass in its constructor

```
class Vehicle {
    private String name;
    public Vehicle(String name) {
        this.name = name;
    }
}
class Car extends Vehicle {
    private int milesPerGallon;
    public Car() {
        super(“Greg’s Car”);
        this.milesPerGallon = 32;
    }
}
```

EXERCISE

1. Update your Vehicle class to take a name in the constructor
2. Call it from each of the subclasses

ABSTRACT CLASSES

Java provides a way to model **abstraction** by using the 'abstract' keyword

A method can be 'abstract' and not provide the implementation

public abstract void addMiles(int miles);

ABSTRACT CLASSES

A class **must be** 'abstract' if any of its methods are 'abstract'

```
public abstract class Vehicle {
        public abstract void addMiles(int miles);
}
```

ABSTRACT CLASSES

No all methods need to be ‘abstract’ and it can have instance variables

```
public abstract class Vehicle {
        public int milesLeft;
        public abstract void addMiles(int miles);
        public int milesLeft() {
                return milesLeft;
        }
}
```

ABSTRACT CLASSES

You cannot make an object instance of an **abstract** class

Instead, you must **extend** it (using **Inheritance**) and implement the **abstract** methods

ABSTRACT CLASSES EXAMPLE

```
public abstract class Animal {
    public abstract String communicate();
}
```

```
public class Dog extends Animal
{
    public String communicate() {
        wagTail();
        return "Bark!";
    }
}
```

```
public class Cat extends Animal
{
    public String communicate() {
        return "Meow";
    }
}
```

EXERCISE

Create an abstract class and implementations for the bakery exercise

- Baking a cake
- Baking a pastry
- Brewing a coffee

INTERFACE

Are NOT a class

Cannot be instantiated with 'new'

Cannot have any implementation details

All methods on an interface are public

INTERFACE

An interface is used with the keyword 'implements'

A class can implement multiple interfaces

Example:

```
public interface Drivable { … }
public interface Transportation { … }
public class Vehicle implements Drivable, Transportation {
}
```

INTERFACE

A class must implement all of the methods defined in its interfaces or be declared abstract

Example:

```
public interface Drivable { … }
public interface Transportation { … }
public class Vehicle implements Drivable, Transportation {
    //implementations for all of Drivable's and
Transportation's methods
}
```

EXERCISE

- Create an interface called Openable that has a method openTheStore
- Make a class called Bakery that implements Openable
- Create another type of store that is Openable

ASSIGNMENT

Review assignment for any clarifications

Clone it together

MIDTERM EXAM

Keep making progress!

Review the topics you're weak on

QUIZ

Do NOT use IntelliJ for this week's quiz

SEE YOU DECEMBER 6TH!

You Midterm Exam is due noon before lecture!

Don’t forget to ask question early in the week

REFERENCED MATERIAL

Petkov, A. (2018). *How to explain object-oriented programming concepts to a 6-year-old.* [online] freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/object-oriented-programming-concepts-21bb035f7260/ [Accessed 17 Nov. 2019].