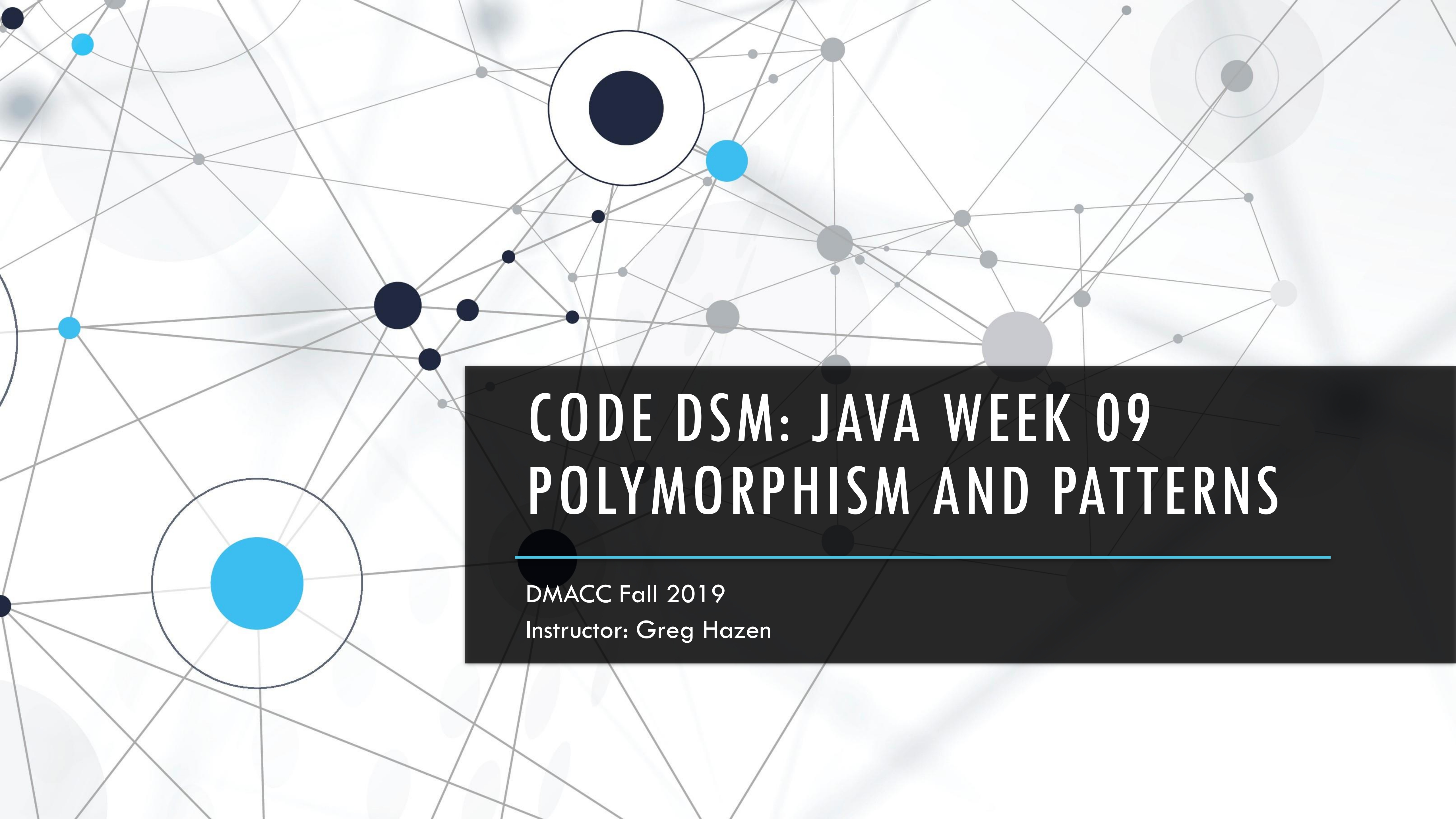

# CODE DSM: JAVA WEEK 09 POLYMORPHISM AND PATTERNS

DMACC Fall 2019

Instructor: Greg Hazen

# AGENDA REVIEW

WEEK 8 ASSIGNMENT

ENCAPSULATION

ABSTRACTION

INHERITANCE

# AGENDA

POLYMORPHISM

OBJECT PATTERNS

# REVIEW WEEK 8 ASSINGMENT

# SOME OBJECT ORIENTED TERMS

**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

# 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) { … }
}
```

State

Behavior

# 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 *Cat* 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)*

# 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

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

# 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

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

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

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() {
        super.onAction = “Vroom”;
    }
}
```

# ‘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;
    }
}
```

# ABSTRACT CLASS

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

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

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

# ABSTRACT CLASS

```
public abstract class Animal {
  public String speak() {
    return “”;
  }
}
```

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

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

# INTERFACE

```
public interface Animal {
  public String speak();
}
```

```
public class Dog implements Animal {
    @Override
    public String communicate() {
        return "Bark!";
    }
}

public class Cat implements Animal {
    @Override
    public String communicate() {
        return "Meow";
    }
}
```

# OBJECT ORIENTED PROGRAMMING

We've seen

- Encapsulation
- Abstraction
- Inheritance

Next, Polymorphism.

# POLYMORPHISM - DEFINITIONS

Sharing behaviors between objects even though they are used in different ways

Two ways to express this in Java

- Static polymorphism – compile time
- Dynamic polymorphism – run time

# STATIC POLYMORPHISM

Methods can have the same name as long as:

- The return type is the same
- The input parameters are different for each

This is called method **overloading**

```
public interface Animal {
    public void feed(String food);
    public void feed(String food, int
                             amount);
}
```

# STATIC POLYMORPHISM

Works for constructors too

Can have differing access modifiers

This is also called constructor **overloading**

```
public class Cat {
    public Cat(String name) {
        this(name, 0);
    }
    public Cat(String name, int age) {
        File fileToSave = new File(...);
        this(fileToSave);
    }
    private Cat(File file) {
        // Initialize private variables
    }
}
```

# STATIC POLYMORPHISM

At compile-time

- Check every reference to an overloaded method
- Check every reference to an overloaded constructor

# EXERCISE: BAKERY EQUIPMENT

Create an abstract class called Equipment that has two String member variables: equipmentType and equipmentLocation

Create a default constructor and a constructor that loads all of the member variables

Add an abstract method called useEquipment that returns a Boolean for the following scenarios

- For an integer duration
- For an integer duration at an integer intensity

# DYNAMIC POLYMORPHISM

Determines which method in a parent/child to execute

```
Animal animal = new Cat();
animal.speak(); // “Meow”
Cat cat = new Cat();
cat.speak(); // “Meow”
```

```
public class Animal {
    public void speak() {
        System.out.println(“I can’t speak!”);
    }
}

public class Cat extends Animal {
    @Override
    public void speak() {
        System.out.println(“Meow”);
    }
}
```

# DYNAMIC POLYMORPHISM

A runtime

- Determines whether to call the parent or child method

# EXERCISE: BAKERY EQUIPMENT

Using your exercise from earlier, create a Mixer class and Oven class that both extend the Equipment class

- Use the constructor to set the equipmentType and equipmentLocation
- Override both useEquipment methods to print something and return a boolean

# PREVENTING UNWANTED POLYMORPHISM

Sometimes, it may be a really good idea to prevent other developers from overloading methods or creating subclasses

- Security code
- Insuring a certain value is only ever calculated a certain way

This can be accomplished by using ‘final’

# PREVENTING UNWANTED POLYMORPHISM

To prevent a class from being extended, use ‘final’ in the class definition

public final class SecurityContext { … }

public class SuperSekritSecurityContext extends SecurityContext { … } //compilation error!!

# PREVENTING UNWANTED POLYMORPHISM

To prevent a method from being overriden, use ‘final’ in the method signature

```
public class PasswordGenerator {
    public final String generatePassword(Integer
characterCount) {
            //cannot be overridden by a subclass
    }
}
```

# OBJECT PATTERNS

Developers frequently solve the same problems over and over again

Great developers recognize these patterns and have a solution already in mind

# OBJECT PATTERNS

There are lots of patterns

- The goal isn’t to memorize them all
- Know the most common and how to Google them to remind yourself

# CREATIONAL DESIGN PATTERNS

Comes from the Design Patterns and commonly referred to as the Gang of Four (GoF) for the four authors

- **Singleton**
- **Factory**
- Abstract
- Builder

# SINGLETON DESIGN PATTERN

Ensure only one instance of the object exists throughout the JVM

Provide a global class to access that one instance

**Note**, the constructor has the private access modifier

```
public class Singleton  {
    private Singleton() {}

    private static class SingletonHolder {
        public static final Singleton instance =
                   new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonHolder.instance;
    }
}
```

Baeldung.com. (2019).

# SINGLETON DESIGN PATTERN

When to Use:

- For resources that are expensive to create (like database connection objects)
- References to application configuration settings
- Classes that access shared resources

Baeldung.com. (2019).

# FACTORY DESIGN PATTERN

Class for creating objects so that other classes don’t have to know how to

```
public interface Polygon { }
public class Triangle implements Polygon { }
public class Square implements Polygon { }

public class PolygonFactory {
    public Polygon getPolygon(int sides) {
        if(sides == 3) {
            return new Triangle();
        }
        if(sides == 4) {
            return new Square();
        }
        … // additional complexity
    }
}
```

Baeldung.com. (2019).

# FACTORY DESIGN PATTERN

When to Use Factory Method Design Pattern

- When implementation of an interface or an abstract class is expected to change frequently
- When the initialization process is relatively simple, and the constructor only requires a handful of parameters

# CONTINUED DESIGN PATTERNS

Read more about design patterns to become a better designer and developer

In course contents this week, I reference multiple links for further reading. These are optional but recommended when you have time!

# QUIZ

Available in Blackboard

Do not use IntelliJ

# ASSIGNMENT

Review assignment for any clarifications

Clone it together

# SEE YOU DECEMBER 13TH!

Don’t forget to ask question early in the week

# REFERENCES

Baeldung.com. (2019). *Introduction to Creational Design Patterns*. [online] Available at: https://www.baeldung.com/creational-design-patterns [Accessed 30 Nov. 2019].