Lets Code!

OOP: Four Pillars of OOP

Instructor: Tariq Hook

You can find me on github @code-rhino

Key Terms

- Four Pillars
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction

Encapsulation

Encapsulation

- also known as *data hiding*.
- mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.
- variables of a class are hidden from other classes, and can be accessed only through the methods of their current class.

Achieving Encapsulation

- Achieved by wrapping several data fields and methods in a single entity

```
public class Person {
  private String name;
  private Integer age;
  public Person(String name, Integer age) {
    this.name = name;
    this.age = age;
  }
  // getters and setters omitted for brevity
}
```

Encapsulation
Managing person-data without Encapsulation

- Unrelated variables coexist in scope

```
// setting name and age of leon
String tariqName = "Tariq";
Integer tariqAge = 41;

// setting name and age of wilhem
String froilanName = "Froilan";
Integer froilange = 41;

// changing state of leon
tariqName = "Tariq";
tariqAge = 42;

// changing state of wilhem
froilanName = "Froilan";
froilanAge = 24;
```

Encapsulation
Managing person-date with Encapsulation

- Related variables coexist in related scope

```
public void demo() {
    // setting name and age of leon
    Person tariq = new Person("Tariq", 41);

    // setting name and age of wilhem
    Person froilan = new Person("Froilan", 41);

    // changing name of tariq
    tariq.setName("Tariq");
    tariq.setAge(42);

    // changing name of wilhem
    froilan.setName("Froilan");
    froilan.setAge(42);
}
```

Inheritance

Inheritance

- the process where one class acquires the members (methods and fields) of another.
- Ensures information is made manageable in a hierarchical order.
- Inheritance is achieved in java by use of the keyword *extends* or *implements*

Inheritance
Designing an Animal class

- the class whose properties are inherited is known as *superclass* (base-class).
  - Here, the `Animal` class is intended to be the super class.

```
public class Animal {
  private Point position;
  protected String phrase;

  public Animal(String phrase) { this.phrase = phrase; }

  public void setPosition(Integer xPosition, Integer yPosition) {
    this.position = new Point(xPosition, yPosition);
  }

  public Point getPosition() { return this.position; }
}
```

Inheritance
Designing a Fox class

- The class which inherits the members of other is known as *subclass* (derived class)
    - Here, the **`Fox`** class inherits members from **`Animal`**
    - Notice, **`Fox`**'s reference to *protected* member of **`phrase`**.

```
public class Fox extends Animal {
  public Fox() {
    super("Bark!");
  }

  public void useSpeech() {
    System.out.println(super.phrase); // inherited member
  }
}
```

Inheritance
Accessing Animal Properties

```
public void demo() {
  Fox fox = new Fox();
  Point foxPosition = fox.getPosition() // method of `Animal` class
  System.out.println(foxPosition);
}
```

Output

```
null
```

Inheritance
Accessing Animal Properties

- *protected* members are only accessible from within the scope of the inheriting *subclass*.

```
public void demo() {
  Fox fox = new Fox();
  String phrase = fox.phrase; // invalid reference
}
```

Polymorphism

Polymorphism

- ability to take on different forms or stages
- Dynamic (run-time) polymorphism: JVM, rather than compiler, decides which method to call
- Static (compile-time) polymorphism: compiler identifies method to call by method-signature

Dynamic Polymorphism

- Also known as *run time* polymorphism
- The Java compiler does not understand which method to call, the decision is deferred to JVM at run-time.
- *Method overriding* of instance methods is an example of dynamic polymorphism in Java

Dynamic (Runtime) Polymorphism
Designing Bird class

- The following design enforces polymorphic behavior of a `Bird`

```
public interface Flyer { void fly(int distance); }
```

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

```
public class Bird extends Animal implements Flyer {
  @Override
  public void fly(int distance) {
    this.setLocation(getX(), getY()+distance);
  }

  @Override
  public String speak() {
    return "chirp!"
  }
}
```

Dynamic (Runtime) Polymorphism
Bird Example

- `Bird` as `Animal` exposes only `speak` method.

```
public void demo() {
  Animal animal = new Bird();
  animal.speak();
}
```

Dynamic (Runtime) Polymorphism
Bird Example

- `Bird` as `Flyer` exposes only `fly` method.

```
public void demo() {
  Flyer flyer = new Bird();
  flyer.fly(10);
}
```

Dynamic (Runtime) Polymorphism
Bird Example

- `Bird` as `Bird` exposes `speak` and `fly` method.

```
public void demo() {
  Bird bird = new Bird();
  bird.speak();
  bird.fly(10);
}
```

Static Polymorphism

- Also known as *compile-time polymorphism* or *static binding*
- At compile time, Java knows which method to invoke by checking the method signatures.

Static (compile-time) Polymorphism
Designing Person class

- The following design enforces polymorphic behavior of a `Person` by overloading the constructor

```
public class Person {
  private String name;
  private Integer age;
  private Color eyeColor;

  public Person(String name, Integer age, Color eyeColor) {
    this.name = name;
    this.age = age;
    this.eyeColor = eyeColor;
  }

  public Person() {
    this("Leroy", 50, Color.BLACK);
  }
}
```

Static (compile-time) Polymorphism
Person Example

- *Primary* constructor of `Person` class

```
public void demo() {
  String personName = "Jamar";
  Integer personAge = 25;
  Color personEyeColor = Color.BROWN;
  Person person = new Person(personName, personAge, personEyeColor);

  System.out.println(person.getName());
  System.out.println(person.getAge());
  System.out.println(person.getEyeColor());
}
```

Output

```
Jamar
25
BROWN
```

Static (compile-time) Polymorphism
Person Example

- *Nullary* constructor of `Person` class

```
public void demo() {
  Person person = new Person();

  System.out.println(person.getName());
  System.out.println(person.getAge());
  System.out.println(person.getEyeColor());
}
```

Output

```
John
50
BLACK
```

Abstraction

Abstraction

- the process of exposing essential features of an entity while hiding other irrelevant detail.
- abstraction reduces code complexity and at the same time it makes your aesthetically pleasant.

Abstraction

- Fetching & printing user input without abstraction

```
public void withoutAbstraction() {
  InputStream inputStream = System.in;
  Scanner scanner = new Scanner(inputStream);
  String promptName = "What is your name?";
  String promptAge = "What is your age?";

  System.out.println(promptName);
  String stringName = scanner.nextLine();

  System.out.println(promptAge);
  String stringAge = scanner.nextLine();
  Integer intAge = Integer.parseInt(stringAge);

  System.out.println("Name = " + stringName);
  System.out.println("Age = " + intAge);
}
```

Abstraction

- Fetching & printing user input with abstraction

```
public void withAbstraction() {
    IOConsole console = new IOConsole();
    String promptName = "What is your name?";
    String promptAge = "What is your age?";
    String userInputName = console.getStringInput(promptName);
    Integer userInputAge = console.getIntegerInput(promptAge);
    Person person = new Person(userInputName, userInputAge);
    console.print(person.toString());
}
```

Wrap Up

- Four Pillars
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction

Keep Coding !!!

Clean Code is Happy Code