SWEN20003

Workshop 10, Week 11

Eleanor McMurtry, University of Melbourne

Part 1: Event-driven programming

A modern paradigm

- Programs can be thought of as responding to requests and actions

A modern paradigm

- Programs can be thought of as responding to requests and actions
- **Event-driven:** behaviour defined by responses to **events**

Example

Event	Response
User clicks “submit”	Data sent to server
Response received from server	Display result to user
User clicks “save”	Data saved to disk
Network disconnected	Retry connection

Example

Event	Response
User clicks “submit”	Data sent to server
Response received from server	Display result to user
User clicks “save”	Data saved to disk
Network disconnected	Retry connection

- Event-driven programming de-couples **event creation** from **event handling**.

Example

Event	Response
User clicks “submit”	Data sent to server
Response received from server	Display result to user
User clicks “save”	Data saved to disk
Network disconnected	Retry connection

- Event-driven programming de-couples **event creation** from **event handling**.
- In Java: often use the **Observer** design pattern.

Asynchronous programming

- Handle many events at the same time in any order

Asynchronous programming

- Handle many events at the same time in any order
- Tasks can be paused (e.g. while waiting for a server response) to be resumed later

Asynchronous programming

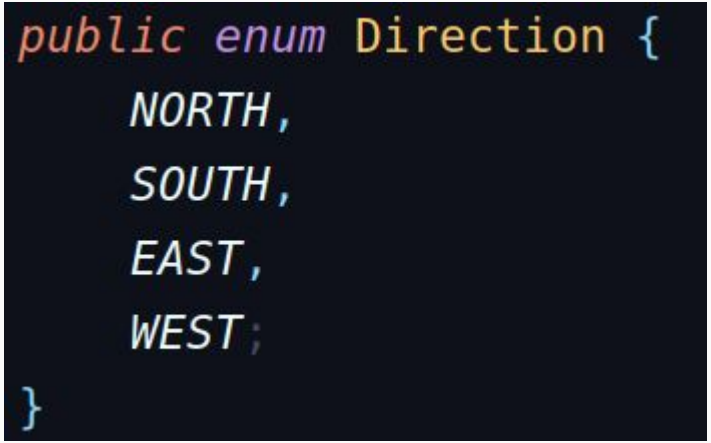
- Handle many events at the same time in any order
- Tasks can be paused (e.g. while waiting for a server response) to be resumed later

- Not well-supported in Java; see `java.util.concurrent`

Part 2: Enumerated types

What is an enum?

- A class with **a set number of** instances

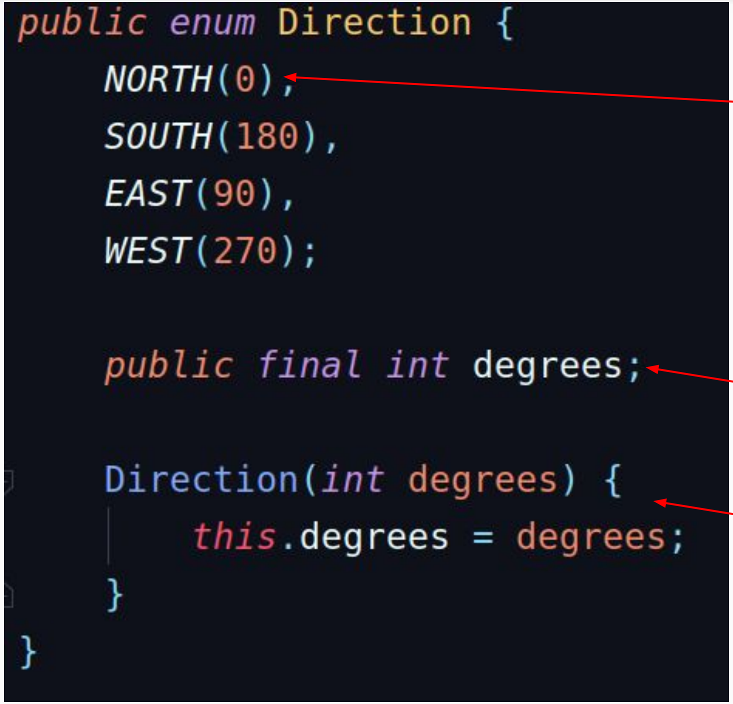


```java
public enum Direction {
    NORTH,
    SOUTH,
    EAST,
    WEST;
}
```

Why enum?

- More clearly express intent
- Better than `String` with constants: invalid values are syntax errors

Enums are classes



```java
public enum Direction {
    NORTH(0),
    SOUTH(180),
    EAST(90),
    WEST(270);

    public final int degrees;

    Direction(int degrees) {
        this.degrees = degrees;
    }
}
```

calling the constructor

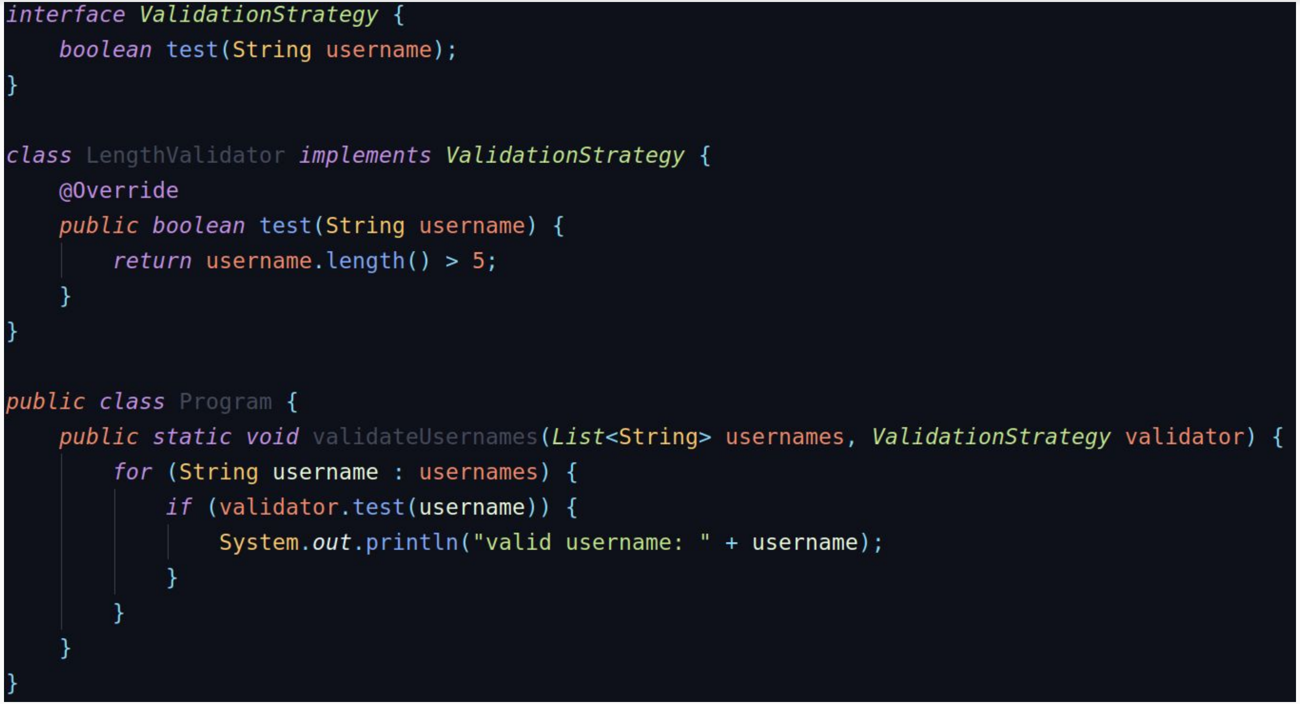
attribute

constructor

Can you think of examples?

Part 3: Functional Java with streams

The Strategy pattern

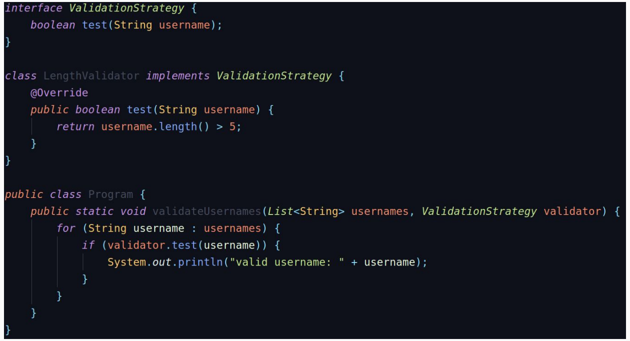


```java
interface ValidationStrategy {
    boolean test(String username);
}

class LengthValidator implements ValidationStrategy {
    @Override
    public boolean test(String username) {
        return username.length() > 5;
    }
}

public class Program {
    public static void validateUsernames(List<String> usernames, ValidationStrategy validator) {
        for (String username : usernames) {
            if (validator.test(username)) {
                System.out.println("valid username: " + username);
            }
        }
    }
}
```

The Strategy pattern



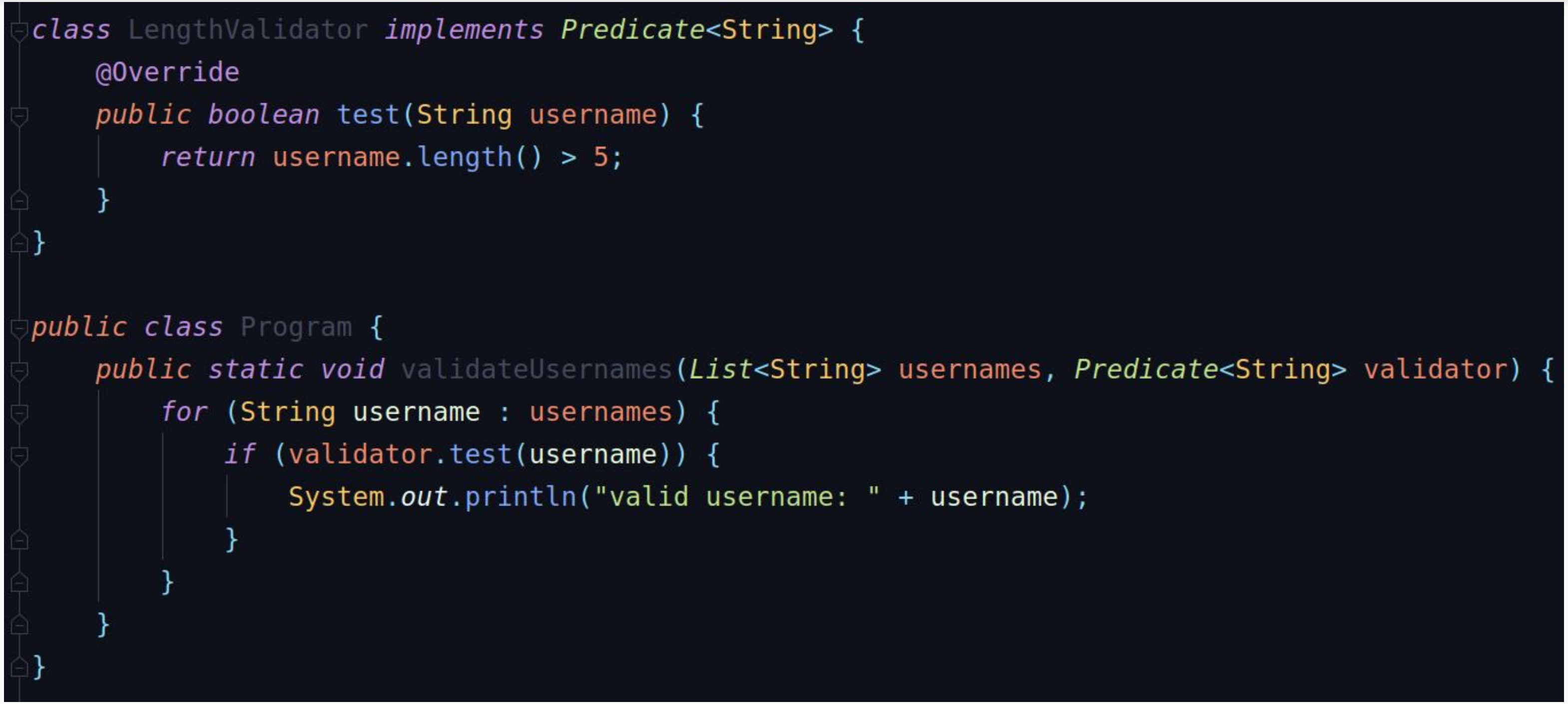
```java
interface ValidationStrategy {
    boolean test(String username);
}

class LengthValidator implements ValidationStrategy {
    @Override
    public boolean test(String username) {
        return username.length() > 5;
    }
}

public class Program {
    public static void validateUsernames(List<String> usernames, ValidationStrategy validator) {
        for (String username : usernames) {
            if (validator.test(username)) {
                System.out.println("valid username: " + username);
            }
        }
    }
}
```

- A strategy to test a condition is so common that it’s built in: `Predicate<T>`

The Strategy pattern

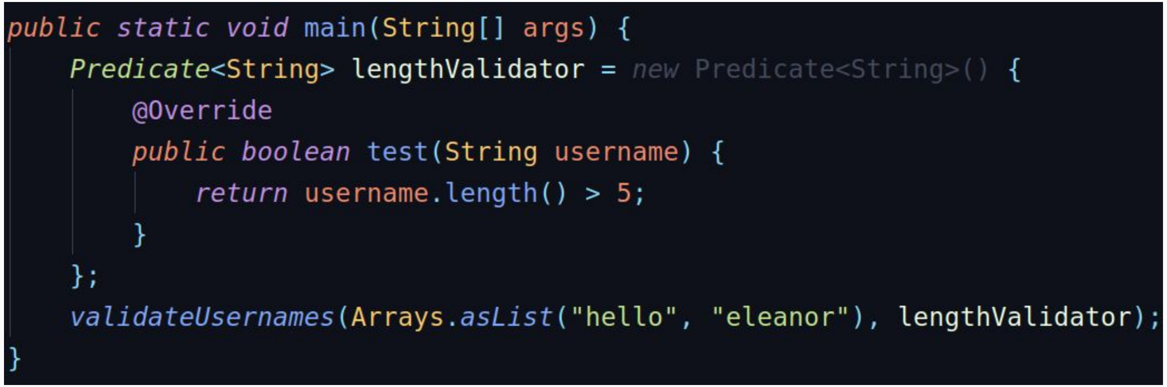


```java
class LengthValidator implements Predicate<String> {
    @Override
    public boolean test(String username) {
        return username.length() > 5;
    }
}

public class Program {
    public static void validateUsernames(List<String> usernames, Predicate<String> validator) {
        for (String username : usernames) {
            if (validator.test(username)) {
                System.out.println("valid username: " + username);
            }
        }
    }
}
```

Anonymous classes

- A class that is only used once adds unnecessary complexity.

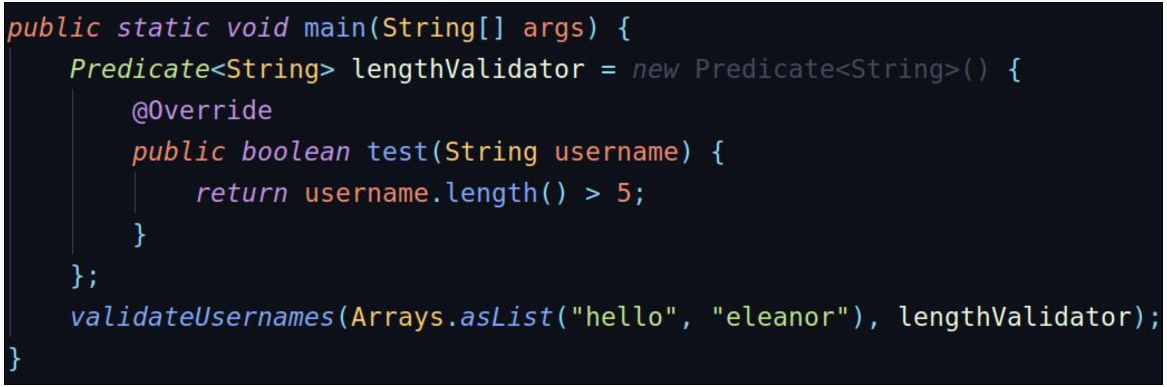


```java
public static void main(String[] args) {
    Predicate<String> lengthValidator = new Predicate<String>() {
        @Override
        public boolean test(String username) {
            return username.length() > 5;
        }
    };
    validateUsernames(Arrays.asList("hello", "eleanor"), lengthValidator);
}
```

- lengthValidator is an instance of an **anonymous class**.

A further simplification

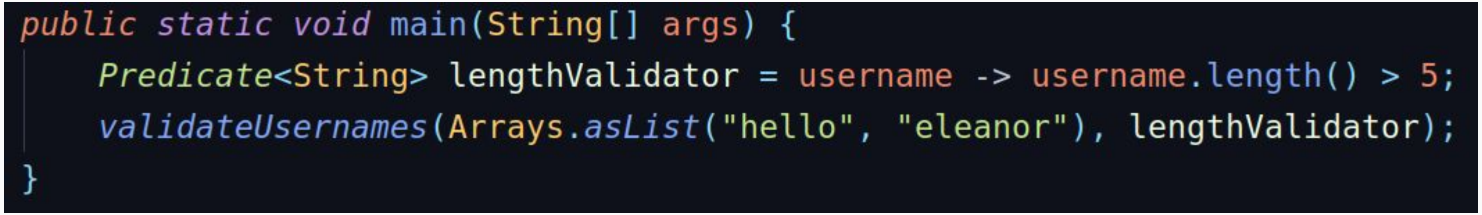
- Notice that we use lengthValidator *as though it were a function*!



```java
public static void main(String[] args) {
    Predicate<String> lengthValidator = new Predicate<String>() {
        @Override
        public boolean test(String username) {
            return username.length() > 5;
        }
    };
    validateUsernames(Arrays.asList("hello", "eleanor"), lengthValidator);
}
```

A further simplification: lambda functions

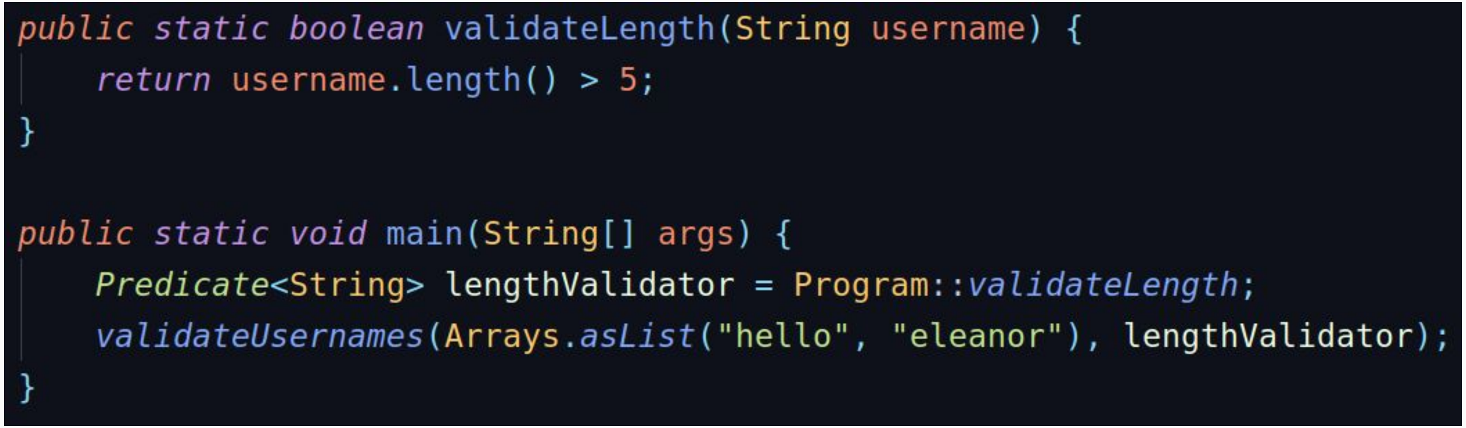
- Notice that we use lengthValidator *as though it were a function*!
- Introduce **lambda functions:** anonymous classes that act like functions.



```
public static void main(String[] args) {
    Predicate<String> lengthValidator = username -> username.length() > 5;
    validateUsernames(Arrays.asList("hello", "eleanor"), lengthValidator);
}
```

Method references

- We can also use existing methods as though they were anonymous classes (similar to **function pointers**).



```
public static boolean validateLength(String username) {
    return username.length() > 5;
}

public static void main(String[] args) {
    Predicate<String> lengthValidator = Program::validateLength;
    validateUsernames(Arrays.asList("hello", "eleanor"), lengthValidator);
}
```

Demonstration of Java streams using lambda functions