SWEN20003
Object Oriented Software Development

Advanced Java and OOP

Shanika Karunasekera
karus@unimelb.edu.au

University of Melbourne

The Road So Far

- Java Foundations
  - A Quick Tour of Java
- Object Oriented Programming Foundations
  - Classes and Objects
  - Arrays and Strings
  - Input and Output
  - *Software Tools and Bagel*
  - Inheritance and Polymorphism
  - Interfaces and Polymorphism
- Advanced Object Oriented Programming and Software Design
  - Modelling Classes and Relationships
  - Generics
  - Collections and Maps
  - Design Patterns
  - Exceptions
  - Software Design and Testing
  - Event Driven Programming

Lecture Objectives

After this lecture you will be able to:

- Describe and use enumerated types
- Make use of functional interfaces and lambda expressions
- Use Java streams

Enumerated Types

Motivating Example

You have asked you to build a preliminary design, before telling you the rules of a card game.

How would you design this game, knowing only that you are implementing a card game.

Motivating Example

Problem: How do you represent a `Card` class?

A `Card` consists of a `Suit`, `Rank`, and `Colour`.

Okay... How do we represent those?

Defining a Card

```
public class Card {

    private Rank rank;
    private Suit suit;
    private Colour colour;

    public Card(Rank rank, Suit suit, Colour colour) {
        this.rank = rank;
        this.suit = suit;
        this.colour = colour;
    }

}
```

Enumerated Types

Keyword

enum: A class that consists of a **finite** list of constants.

- Used any time we need to represent a fixed set of values
- Must list *all* values
- Otherwise, just like any other class; they can have methods and attributes!

Let's define the `Card` class and the `Rank` enum.

Defining a Card

```
public enum Rank {
    ACE,
    TWO,
    THREE,
    FOUR,
    FIVE,
    SIX,
    SEVEN,
    EIGHT,
    NINE,
    TEN,
    JACK,
    QUEEN,
    KING
}
```

What does it do? How would you expect to **use** it?

Enum Variables

```
Rank rank = Rank.ACE;
Card card = new Card(Rank.FOUR, ..., ...);
```

The values of an enum are accessed *statically*, because they are constants.

Enum objects are treated just like any other object.

Let's make the other components...

Defining a Card

```
public enum Colour {
    RED, BLACK
}
```

```
public enum Suit {
    SPADES, CLUBS, DIAMONDS, HEARTS
}
```

Can anyone see a flaw in our `Card` design? Any assumptions we've made/not made?

Shouldn't the `Colour` and `Suit` be related in some way?

Defining a Card

```
public enum Suit {
    SPADES(Colour.BLACK),
    CLUBS(Colour.BLACK),
    DIAMONDS(Colour.RED),
    HEARTS(Colour.RED);

    private Colour colour;

    private Suit(Colour colour) {
        this.colour = colour;
    }
}
```

Now, every `Suit` is automatically tied to the appropriate `Colour`; this *may or may not* be useful behaviour.

Enum Variables

```
public static void main(String args[]) {
    ArrayList<Rank> ranks = new ArrayList<>();

    ranks.add(Rank.TEN);
    ranks.add(Rank.FOUR);
    ranks.add(Rank.EIGHT);
    ranks.add(Rank.THREE);
    ranks.add(Rank.ACE);

    System.out.println(ranks);
    Collections.sort(ranks);
    System.out.println(ranks);
}
```

→ sort by the order in Rank class

```
[TEN, FOUR, EIGHT, THREE, ACE]
[ACE, THREE, FOUR, EIGHT, TEN]
```

Enum Variables

Enums come pre-built with...

- Default constructor
- `toString()`
- `compareTo()`
- `ordinal()`

Enums are also *classes*, so we can add (or override) any method or attribute we like.

```
public boolean isFaceCard() {
    return this.ordinal() > Rank.TEN.ordinal();
}
```

比10大
就是Face Card

Assess Yourself

What is an enum?

What other applications can you think of for them?

Variadic Parameters

What?

```
1  import java.util.Arrays;
2  import java.util.List;
3
4  public class VariadicExample1 {
5      public static void main (String[] args) {
6          List<Integer> list1 = Arrays.asList(12, 5);
7          System.out.println(list1);
8
9          List<Integer> list2 = Arrays.asList(12, 5, 45, 18);
10         System.out.println(list2);
11
12         List<Integer> list3 = Arrays.asList(12, 5, 45, 18, 33);
13         System.out.println(list3);
14     }
15 }
```

Output:

```
[12, 5]
[12, 5, 45, 18]
[12, 5, 45, 18, 33]
```

Variadic Parameters

How does this method work? Is it overloaded for any number of arguments...?

Of course not, that's silly.

Keyword

Variadic Method: A method that takes an *unknown number* of arguments.

Variadic methods *implicitly* convert the input arguments into an array. Be careful!

Variadic Parameters

Let us write our own Variadic method!

```
1  public class variadicExample2 {
2      public static void main (String[] args) {
3          System.out.println(concatenate("Hello", "world!"));
4          System.out.println(concatenate("Programming", , "is", "fun!"));
5      }
6
7      public static String  concatenate(String... strings) {
8          String string = "";
9          for (String s : strings) {
10             string += " " + s;
11         }
12         return string;
13     }
14 }
```

automatically converted to arraylist of string

Output:

```
Hello world!
Programming is fun!
```

Assess Yourself

Write a variadic method that computes the average of an unknown number of integers.

```
1  public class ComputeAverage {
2      public static void main (String[] args) {
3          System.out.println("Average = " + average(1, 2));
4          System.out.println("Average = " + average(1, 3, 5));
5      }
6      public static double average(int... nums) {
7          int total = 0;
8          for (int i : nums) {
9              total += i;
10         }
11         return 1.0 * total / nums.length;
12     }
13 }
```

Output:

```
Average = 1.5
Average = 3.0
```

Functional Interfaces

Assess Yourself

Thinking of a game, how might we represent the fact that *some* `Sprite` objects can attack, but not all?

```
public interface Attackable {
    public void attack();
}
```

What is the purpose of the interface?

What are the limitations of the interface?

What if there was an easier way?

Functional Interfaces

Keyword

Functional Interface: An interface that contains only a single abstract method; also called a Single Abstract Method interface.

```
@FunctionalInterface
public interface Attackable {
    public void attack();
}
```

Functional interfaces can contain only one non-static method; adding more will raise an error.

Functional Interfaces

Cool story... But... Why?

Functional interfaces are a *tool* that we can use with other techniques...

But let's look at a few functional interfaces first.

Functional Interfaces

```
public interface Predicate<T>
```

The `Predicate` functional interface...

- Represents a *predicate*, a function that accepts one argument, and returns `true` or `false`
- Executes the `boolean test(T t)` method on a single object
- Can be combined with other predicates using the `and`, `or`, and `negate` methods

Functional Interfaces

```
public interface UnaryOperator<T>
```

The `UnaryOperator` functional interface...

- Represents a *unary* (single argument) function that accepts one argument, and returns an object of the same type
- Executes the `T apply(T t)` method on a single object

Assess Yourself

We've seen two functional interfaces: `Predicate` and `UnaryOperator`.

Sample exam question
Describe one application/use case for **each** of the following functional interfaces: `Predicate`, `UnaryOperator`.

Sample exam question
The functional interface `ToIntFunction<T>` represents a function that takes a single argument, and converts it to an integer. Give a **specific** example of how you might use this.

Functional Interfaces

"Oh my god, so many interfaces... Do we have to make a class for each one?!"

That brings us to...

Lambda Expressions

Lambda Expressions

Keyword

Lambda Expression: A technique that treats code as *data* that can be used as an "object"; for example, allows us to *instantiate* an interface without implementing it.

```
public interface Predicate<T>
```

```
Predicate<Integer> p = i -> i > 0;
```

Predicate, return a boolean
take an integer, return true if i > 0

The `Predicate` functional interface is now an *object* that implements the function to test if integers are greater than zero.

Predicate - Example

```
1  import java.util.function.Predicate;
2  public class PredicateDemo {
3      public static void main (String[] args) {
4          Predicate<Integer> p = i -> i > 0;
5          Boolean b1 = p.test(10);
6          System.out.println("b1: " + b1);
7          Boolean b2 = p.test(-10);
8          System.out.println("b2: " + b2);
9      }
10 }
```

Output:

```
b1: true
b2: false
```

UnaryOperator - Example

```
1  import java.util.function.UnaryOperator;
2  public class UnaryOperatorDemo {
3      public static void main (String[] args) {
4          UnaryOperator<Integer> u1 = i -> i + 1;
5          UnaryOperator<Integer> u2 = i -> i - 1;
6          Integer b = 10;
7          Integer b1 = u1.apply(b);
8          Integer b2 = u2.apply(b);
9          System.out.println("b1 =  " + b1);
10         System.out.println("b2 = " +  b2);
11     }
12 }
13
```

unaryoperator
return the same type

Output:

```
b1 = 11
b2 = 9
```

Lambda Expressions

```
(sourceVariable1, sourceVariable2, ...)
            -> <operation on source variables>
```

A lambda expression takes zero or more arguments (source variables) and applies an operation to them

Operations could be:

- Doubling an integer
- Comparing two objects
- Performing a boolean test on an object
- Copying an object
- ...

Assess Yourself

```
1  import java.util.function.Predicate;
2  import java.util.List;
3  import java.util.Arrays;
4  public class CombiningPredicates {
5      public static void main (String args[]) {
6          Predicate<Integer> p1 = i -> i > 0;
7          Predicate<Integer> p2 = i -> i%2 == 0;
8          Predicate<Integer> p3 = p1.and(p2);
9          List<Integer> nums = Arrays.asList(1, 2, 5, 6, -2, 7, 4, 5);
10         for (Integer i : nums) {
11             if (p3.test(i)) {
12                 System.out.println(i);
13             }
14         }
15     }
16 }
```

combine two predicate

```
2
6
4
```

Assess Yourself

Consider the following method in the `List<T>` class.

```
public abstract class List<T> {
    public void replaceAll(UnaryOperator<T> operator);
}
```

Can you write a program which uses this method to convert words stored in a `ArrayList`, to uppercase and then to lower case using the `replaceAll` method?

Assess Yourself

```
1  import java.util.List;
2  import java.util.Arrays;
3  import java.util.function.UnaryOperator;
4  public class ConvertStringCase {
5      public static void main(String[] args) {
6          List<String> names = Arrays.asList("Tony", "Thor", "Thanos");
7          UnaryOperator<String> toUpper = a -> a.toUpperCase();
8          names.replaceAll(toUpper);
9          System.out.println(names);
10         UnaryOperator<String> toLower = a -> a.toLowerCase();
11         names.replaceAll(toLower);
12         System.out.println(names);
13     }
14 }
```

why not directly use List ?

Output:

```
[TONY, THOR, THANOS]
[tony, thor, thanos]
```

Anonymous Classes vs. Lambdas

Lambda expressions can often be used *in place* of anonymous classes, but are not the same thing.

Anonymous Class

```
starWarsMovies.sort(new Comparator<Movie> {
    public int compare(Movie m1, Movie m2) {
        return m1.rating - m2.rating;
    }
});
```

Lambda Expression

```
starWarsMovies.sort((m1, m2) -> m1.rating - m2.rating);
```

Lambda Expressions

Lambda expressions are *instances* of *functional interfaces*, that allow us to treat the functionality of the interface as an *object*.

This makes our code **much** neater, and easier to read.

What next?

Method References

Rewind a Bit

```
List<String> names = Arrays.asList("Tony", "Thor", "Thanos");

names.replaceAll(name -> name.toUpperCase());

System.out.println(names);
```

What does this code do?

How would you describe the *effect* of the lambda expressions?

The lambda expression *applies one method* to every element of the list.
We can take this a step further...

Method References

```
names.replaceAll(String::toUpperCase);
```

Keyword

Method Reference: An object that stores a *method*; can take the place of a lambda expression **if** that lambda expression is only used to call a single method.

Method references can be *stored* in the same way a lambda expression can:

```
UnaryOperator<String> operator = s -> s.toLowerCase();
```

```
UnaryOperator<String> operator = String::toLowerCase;
```

Method Reference Examples

Static methods:

```
Class::staticMethod
Person::printWarning
```

Instance methods:

```
Class::instanceMethod || object::instanceMethod
String::startsWith || person::toString
```

Constructor:

```
Class::new
String::new
```

Method arguments are now *implied*, and given when the method is *called*.

Method Reference Examples

```
public class Number {
    public static boolean isOdd(int n) {
        return n % 2 != 0;
    }
}
```

Method Reference Examples

```
1  public class MethodReferenceDemo {
2      public static void main(String[] args) {
3          List<Integer> list = Arrays.asList(12, 5, 45, 18, 33, 24, 40);
4          Predicate<Integer> p = i -> Number.isOdd(i);
5          findNumbers(list, p);
6          System.out.println(findNumbers(list, p));
7          System.out.println(findNumbers(list, Number::isOdd));
8      }
9
10     public static List<Integer> findNumbers(List<Integer> list,
11                 Predicate<Integer> p) {
12         List<Integer> newList = new ArrayList<>();
13         for (Integer i : list) {
14             if (p.test(i)) {
15                 newList.add(i);
16             }
17         }
18         return newList;
19     }
20 }
```

Output:

```
[5, 45, 33]
[5, 45, 33]
```

Streams

Assess Yourself

Write a function that accepts a list of `String` objects, and returns a *new* list that contains only the `Strings` with at least five characters, starting with `"C"`. The elements in the new list should all be in *upper case*.

```
public List<String> findElements(List<String> strings) {
    List<String> newStrings = new ArrayList<>();

    for (String s : strings) {
        if (s.length() >= 5 && s.startsWith("C")) {
            newStrings.add(s.toUpperCase());
        }
    }
}
```

Motivation

Now that we have these fancy new tools, what can we do with them?

What if we wanted to apply *multiple* functions to the same data?

That's where streams come in!

Keyword

Stream: A series of elements given in *sequence*, that are *automatically* put through a *pipeline* of operations.

Using Streams

We can think of that example as applying a sequence of operations to our list:

- Iterating through the list...
- Selecting elements with length greater than five...
- *And* elements with first character "C"...
- Then, converting those elements to upper case...
- And adding them to a new list

```
list = list.stream()
            .filter(s -> s.length() > 5)
            .filter(s -> s.startsWith("C"))
            .map(String::toUpperCase)
            .collect(Collectors.toList());
```

predicate

→ operation on string

map from String to UpperCase string

Streams

Streams are a powerful Java technique that allow you to apply *sequential* operations to a collection of data. These operations include:

- map (convert input to output)
- filter (select elements with a condition)
- limit (perform a maximum number of iterations)
- collect (gather all elements and output in a list, array, String...)
- reduce (*aggregate* a stream into a single *value*)

Given this...

Assess Yourself

Implement a stream pipeline that takes a list of `People`, and generates a `String` consisting of a comma separated list.

The list should contain the names (in upper case) of all the people who are between the ages of 18 and 40.

```
.stream()
.filter(p -> p.getAge() >= 18)
.filter(p -> p.getAge() <= 40)
.map(Person::getName)
.map(String::toUpperCase)
.collect(Collectors.joining(", "));
```

Stream Example

```
1  import java.util.List;
2  import java.util.Arrays;
3  import java.util.stream.Collectors;
4  public class StreamDemo {
5      public static void main(String[] args) {
6          List<Person> people = Arrays.asList(
7              new Person("Peter Parker", 18),
8              new Person("Black Widow", 34),
9              new Person("Thor", 1500),
10             new Person("Nick Fury", 67),
11             new Person("Iron Man", 49)
12         );
13         String output = people.stream()
14             .filter(p -> p.getAge() >= 18)
15             .filter(p -> p.getAge() <= 40)
16             .map(Person::getName)
17             .map(String::toUpperCase)
18             .collect(Collectors.joining(", "));
19         System.out.println(output);
20     }
21 }
```

```
PETER PARKER, BLACK WIDOW
```

What you need to know

You should be able to conceptually describe all of the techniques presented in this lecture.

You should be able to *read* and *interpret* code using any of the techniques in this lecture.

You will **not** be expected to **write** code on anything from today.

Learning Objectives

After this lecture you will be able to:

- Describe and use enumerated types
- Make use of functional interfaces and lambda expressions
- Use Java streams