Assess Yourself

Sample worded exam questions:

1. Describe how sequential programming differs from asynchronous programming.
2. Describe the event-driven programming paradigm.
3. How does the observer pattern demonstrate event-driven programming?
4. Explain (with examples) some of the downsides of using object-oriented programming for game development.
5. Describe the Entity-Component approach to game development.

SWEN20003
Object Oriented Software Development

Advanced Java and OOP

Semester 1, 2019

The Road So Far

- Java Foundations
- Classes and Objects
- Abstraction
- Advanced Java
    - Generic Classes
    - Generic Programming
    - Exception Handling
    - Software Testing and Design
    - Design Patterns
    - Games and Events
- Software Development Tools

Lecture Objectives

After this lecture you will be able to:

- Describe and use enumerated types
- Make use of functional interfaces and lambda expressions
- Do cool s**t in Java

Enumerated Types

Assess Yourself

You've been hired by gambling company *Soulless* to design and implement their newest card game called *Horses**t*.

Soulless have asked you to build a preliminary design before telling you the rules of *Horses**t*.

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

Assess Yourself

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

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

Okay... How do we represent those?

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, like any other class; can have methods and attributes!

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

Defining a Card

```java
public class Card {

    public Rank rank;
    public Suit suit;
    public Colour colour;

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

}
```

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

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

```java
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);
}
```

```
[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.

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

Assess Yourself

What is an enum?

What other applications can you think of for them?

Variadic Parameters

What?

```java
List<Integer> list = Arrays.asList(12, 5);

List<Integer> list = Arrays.asList(12, 5, 45, 18);

List<Integer> list = Arrays.asList(12, 5, 45, 18, 33);
```

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

Of course not, that's silly.

Variadic Parameters

Keyword

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

```java
public String concatenate(String... strings) {
    String string = "";

    for (String s : strings) {
        string += s;
    }

    return string;
}
```

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

Assess Yourself

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

```
public double average(int... nums) {
    int total = 0;

    for (int i : nums) {
        total += i;
    }

    return 1.0 * total / nums.length;
}
```

```
System.out.println(average(1));
System.out.println(average(1, 2, 3));
System.out.println(average(10, 20, 30, 40, 50));
```

```
1.0
2.0
30.0
```

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();
}
```

Seems like a pretty useless 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 "new" 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;
```

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

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

What does this code do?

```
Predicate<Integer> p1 = i -> i > 0;
Predicate<Integer> p2 = i -> i%2 == 0;
Predicate<Integer> p3 = p1.and(p2);

List<Integer> nums = Arrays.asList(1, 2, 5, 6, 7, 4, 5);

for (Integer i : nums) {
    if (p3.test(i)) {
        System.out.println(i);
    }
}
```

```
2
6
4
```

Assess Yourself

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

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

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

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

```
["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

```java
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:

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

```java
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

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

```java
public static List<Integer> findNumbers(
    List<Integer> list, Predicate<Integer> p) {
    List<Integer> newList = new ArrayList<>();

    for(Integer i : list) {
        if(p.test(i)) {
            newList.add(i);
        }
    }

    return newList;
}
```

Method Reference Examples

```java
List<Integer> list = Arrays.asList(12, 5, 45, 18, 33, 24, 40);

// Using an anonymous class
findNumbers(list, new Predicate<Integer>() {
    public boolean test(Integer i) {
        return Numbers.isOdd(i);
    }
});

// Using a lambda expression
findNumbers(list, i -> Numbers.isOdd(i));

// Using a method reference
findNumbers(list, Numbers::isOdd);
```

Streams

Assess Yourself

Write a function that accepts a list of `String` objects, and returns a *new* list that contains only the `String`s 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());
```

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 `Person` objects, 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.

```
List<Person> people = Arrays.asList(
                            new Person("Peter Parker", 18),
                            new Person("Black Widow", 34),
                            new Person("Thor", 1500),
                            new Person("Nick Fury", 67),
                            new Person("Iron Man", 49)
                        );
```

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.

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

```
"PETER PARKER, BLACK WIDOW"
```

Metrics

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.