softserve experience matters

Java Core

Java 8. Features

Agenda

- Anonymous classes
- Default methods in interfaces
- Lambda expressions
- Functional Interfaces
- Method and Constructor References
- Lambda Scopes
  - Accessing local variables
  - Accessing fields and static variables
  - Accessing Default Interface Methods
  - Built-in Functional Interfaces
  - Optional interface

Anonymous Inner Classes

- **Anonymous inner** class – class that has no name and is used if you need to create a single instance of the class.
- Any parameters needed to create an anonymous object class, are given in parentheses following name **supertype**:

```
new Supertype(list_of_parameters) {
    // body
};
```

Anonymous Inner Class - Example

```
people.sort( new Comparator<Person>() {

    @Override
    public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
    }

} );
```

- **new** creates an object
- **Comparator(...)** begins definition of anonymous class
- Similar to

```
public class NameComparator implements Comparator<Person>()
```

- Brace ({) begins class definition

Default Methods for Interfaces

- Java 8 enables us to add non-abstract method implementations to interfaces by utilizing the default keyword. This feature is also known as ***Extension Methods***.

For example:

```
public interface Formula {
    double calculate(int a);

    default double sqrt(int a) {
        return Math.sqrt(a);
    }

}
```

Default Methods for Interfaces

- Besides the abstract method calculate the interface **`Formula`** also defines the default method **`sqrt`**.
- Concrete classes only have to implement the abstract method **`calculate`**.
- The default method **`sqrt`** can be used out of the box.

```
Formula formula = new Formula() {
    @Override
    public double calculate(int a) {
        return sqrt(a * 100);
    }
};

formula.calculate(100);   // 100.0
formula.sqrt(16);         // 4.0
```

Default Methods for Interfaces

- The **`formula`** is implemented as an ***anonymous object***.

```
formula.calculate(100);     // 100.0
formula.sqrt(16);           // 4.0
```

- As we'll see in the next section, there's a much nicer way of implementing ***single method*** objects in Java 8.

Private methods for Interfaces

- From Java SE 9 on-wards, we can write **`private`** and **`private static`** methods too in an interface using **`private`** keyword.

```
public interface Formula {
    private int pow(int a, int b) {
        return (int)Math.pow(a, b);
    }

    private static double getPI() {
        return Math.PI;
    }

    default double circleArea(int radius) {
        return Formula.getPI() * pow(radius, 2);
    }
}
```

Lambda expressions

- Sort a list of strings in ***prior*** versions of Java:

```
List<String> names = Arrays
        .asList("Ivan", "Olexandra", "Anton", "Polina");

Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
});
```

Lambda expressions

- The static utility method **`Collections.sort`** accepts a ***list*** and a ***comparator*** in order to sort the elements of the given list.
- You often find yourself creating anonymous comparators and pass them to the sort method.

Lambda expressions

- In Java 8 comes with a much shorter syntax, ***lambda expressions***

```
Collections.sort(names, (String a, String b)
        -> { return b.compareTo(a); }
);
```

Lambda expressions

- As you can see the code is much shorter and easier to read. But it gets even shorter:

```
Collections.sort(names, (String a, String b)
        -> b.compareTo(a));
```

- For one line method bodies you can skip both the braces **{ }** and the **return** keyword. But it gets even more shorter:

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

- The java compiler is aware of the parameter types so you can skip them as well.

Functional Interfaces

- How does lambda expressions fit into Javas type system? Each lambda corresponds to a given type, specified by an interface. A so called ***functional interface*** must contain ***exactly one abstract method*** declaration.
- Each lambda expression of that type will be ***matched*** to this ***abstract method***.
- To ensure that your interface meet the requirements, you should add the **`@FunctionalInterface`** annotation. The compiler is aware of this annotation and throws a compiler error as soon as you try to add a second abstract method declaration to the interface.

Functional Interfaces

For example,

```
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}

Converter<String, Integer> converter =
        (from) -> Integer.valueOf(from);

Integer converted = converter.convert("123");

System.out.println(converted);      // 123
```

- Keep in mind that the code is also valid if the **`@FunctionalInterface`** annotation would be omitted.

Method and Constructor References

- The above example code can be further simplified by utilizing static method references:

```
Converter<String, Integer> converter = Integer::valueOf;
Integer converted = converter.convert("123");
System.out.println(converted);         // 123
```

- Java 8 enables you to pass references to methods or constructors via the `::` ***expression***. The above example shows how to reference a ***static method***.

Method and Constructor References

- We can also reference instance methods:

```
class StringUtil {
    char startsWith(String s) {
        return Character.valueOf(s.charAt(0));
    }
}

StringUtil strUtil = new StringUtil();

Converter<String, Character> converter = strUtil::startsWith;

char converted = converter.convert("Java");

System.out.println(converted);      // "J"
```

Method and Constructor References

- Let's see how the **::** ***expression*** works for constructors.

```
class Person {

    String firstName;
    String lastName;

    Person() { }

    Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

Method and Constructor References

- Next we specify a person factory interface to be used for creating new persons:

```
interface PersonFactory<P extends Person> {
    P create(String firstName, String lastName);
}
```

- Instead of implementing the factory manually, we glue everything together via constructor references:

```
PersonFactory<Person> personFactory = Person::new;
Person person = personFactory.create("Peter", "Parker");
```

- We create a reference to the Person constructor via **`Person::new`**. The compiler automatically chooses the right constructor by matching the method signature **`create`**.

Lambda Scopes

- Accessing outer scope variables from lambda expressions is very similar to anonymous objects. You can now access *"**effectively final**"* variables from outer scope as well as ***instance*** and ***static fields***.
- Lets consider
  - Accessing local variables
  - Accessing fields and static variables
  - Accessing Default Interface Methods

Accessing local variables

- We can read **final** local variables from outer scope of lambda expressions:

```
final int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);
stringConverter.convert(2);      //3
```

- As well as in anonymous objects the variable **num** is not required to be **final**. This code is also valid:

```
int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);
stringConverter.convert(2);      //3
```

- However **num** must be ***effectively final*** for the code to compile. The following code does *not* compile:

```
int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);
num = 3;
```

Accessing fields and static variables

- We also have both read and write access to instance ***fields*** and ***static variables*** from within lambda expressions.

```
class Test {
    static int outerStaticNum;
    int outerNum;

    void testScopes() {
        Converter<Integer, String> stringConverter1 = (from) -> {
            outerNum = 23;
            return String.valueOf(from);
        };
        Converter<Integer, String> stringConverter2 = (from) -> {
            outerStaticNum = 72;
            return String.valueOf(from);
        };
    }
}
```

Accessing Default Interface Methods

- Interface **Formula** defines a default method **sqrt** which can be accessed from each formula instance including anonymous objects.

```
public interface Formula {
    double calculate(int a);

    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

- But, ***default methods*** cannot be accessed from within lambda expressions. The following code does not compile:

```
Formula formula = (a) -> sqrt(a * 100);
```

Built-in Functional Interfaces

- The JDK 1.8 API contains many built-in functional interfaces. Some of them are well known from older versions of Java like **`Comparator`** or **`Runnable`**.
- Those existing interfaces are extended to enable Lambda support via the **`@FunctionalInterface`** annotation.
- But the Java 8 API is also full of new functional interfaces to make your life easier which contains in package **`java.util.function`**

softserve experience matters

Predicates

- **Predicates** are boolean-valued functions of one argument. The interface contains various default methods for composing predicates to complex logical terms (**`and`**, **`or`**, **`negate`**)

```
public interface Predicate<T> {

    boolean test(T t);
}
```

Functions

- **Functions** accept one argument and produce result. Default methods can be used to chain multiple functions together(**`compose`**, **`andThen`**).

```
public interface Function<T, R> {

    R apply(T t);
}
```

Suppliers

- **Suppliers** produce a result of a given generic type. Unlike Functions, Suppliers don't accept arguments.

```
public interface Supplier<T> {

    T get();
}
```

Consumers

- **Consumers** represents operations to be performed on a single input argument.

```
public interface Consumer<T> {

    void accept(T t);
}
```

Optionals

- **Optionals** are not functional interfaces, instead it's a nifty utility to prevent **`NullPointerException`**.
- Optional is a simple container for a value which may be **`null`** or non-**`null`**.
- Think of a method which may return a non-**`null`** result but sometimes return nothing. Instead of returning **`null`** you return an **`Optional`** in Java 8.

```
Optional<String> optional = Optional.of("Java");

optional.isPresent();           // true
optional.get();                 // "Java"
optional.orElse("fallback");    // "Java"

optional.ifPresent((s) ->
        System.out.println(s.charAt(0)));  // "J"
```

softserve experience matters

The end

USA HQ
Toll Free: 866-687-3588
Tel: +1-512-516-8880

Ukraine HQ
Tel: +380-32-240-9090

Bulgaria
Tel: +359-2-902-3760