Learning Java - A Foundational Journey

Session: 12

Functional Programming in Java

Objectives

- Explain lambda expressions
- Describe method references
- Explain functional interfaces
- Explain default methods

Introduction

- Functional programming is a type of programming approach.
- It emphasizes utilization of functions and writing code that does not change state.
- Using functional programming, you can pass functions as parameters to other functions and return them as values.
- Functional programming provides many benefits to programmers - it makes programs easier to test, it is thread-safe, and is more modular.

Lambda Expression

Facilitates functional programming and make application development easier

Is similar to the concept of an anonymous function

Is a function that expects and accepts input parameters and may return output

Creates possibility of passing functions in a code, which is similar to passing parameters and data in general

- The syntax of a lambda expression is as follows:

```
parameters -> body
where,
            parameters are variables
            -> is the lambda operator
            body is parameter values
```

Rules for Lambda Expressions

Type declarations are optional: The parameter type declaration is optional.

Parentheses around parameters can be omitted: Parentheses usage is optional with single parameter cases.

Curly braces may or may not be present: Similarly, curly braces are optional in single parameter cases.

The return keyword may or may not be present: The return keyword usage is optional in single parameter cases.

Statement body may contain varied number of statements: The body of the lambda expressions can contain zero, one, or more statements.

Single Method Interface and Lambdas

Functional programming can be used to create event listeners.

In Java, event listeners are frequently defined as Java interfaces with a single method.

Lambda Parameters

As Java lambda expressions are similar to anonymous methods, they can also handle parameters.

Zero Parameters:

Parentheses with no comments indicates that the lambda takes no parameters.

The method does not take any parameters.

Example:

```
() ->System.out.println("Zero parameter lambda");
```

One Parameter

If the method takes one parameter, then the lambda expression will be as follows:

```
(param) ->System.out.println("One parameter: "+param);//with parentheses
```

Here, the parentheses contains a value, which means the lambda receives one parameter.

Lambda expression with a single parameter requires no parentheses

Example:

```
param->System.out.println("One parameter: "+param);//without parentheses
```

Multiple Parameters

If the method returns with multiple parameters match, then parameters must be added within the parentheses

Example:

```
(pA, pB) ->System.out.println("Multipleparameters defined: "+pA+", "+pB);
```

Parameter Types

Defining parameter types for a lambda expression is necessary in some cases, where the compiler is inconclusive about the parameter type match between functional interface method and lambda

Example:

```
(previousState, presentState) -> {//bracket usage for multiple lines
System.out.println("The result as Previous state: "+previousState);
System.out.println("The result as Present state: "+presentState);
}
```

Returning a Value

The procedure for returning values from Java lambda expressions is similar to that in a regular Java method

```
 (pA) -> {
System.out.println("The output will be: " + pA);
return "result value"; // return statement
 }
```

A return statement for specific calculations can be added in a shortened form.

Example:

```
(iA, jB) -> {return iA>jB; }
```

```
(iA, jB) ->iA>jB;
```

Lambdas as Objects

Java lambda expression is a sort of object too

Can be assigned as a regular object to a variable and can be passed

```
public interface SampleComparator {
public boolean compare(int iA, int iB);
}
```

Advantages and Uses of Lambda Expressions 1-2

Lambda expressions provide several unique advantages

Some of the advantages of lambda expressions are as follows:

- More readable code
- Rapid fast coding specifically in Collections
- Much easier parallel processing

Advantages and Uses of Lambda Expressions 2-2

```
public class SampleLambda {
   public static void main(String args[]) {
       SampleLambda perform = new SampleLambda();
       //to receive results with type declaration
       MathOperation add = (int ab, int xy) -> ab + xy;
       // to receive results without type declaration
       MathOperation subtr = (ab, xy) -> ab - xy;
       // to receive results with return statement along with curly braces
          MathOperation multi = (int ab, int xy) ->{ return ab * xy; };
           // to receive results without return statement and curly braces
          MathOperation div = (int ab, int xy) -> ab / xy;
          System.out.println("Addition operation with Type declaration : 20 + 5 = "
          + perform.operate(20, 10, add));
          System.out.println("Subtraction operation without Type declaration:
          20 - 5 = " + perform.operate(20, 10, subtr));
          System.out.println("Multiplication with return statement: 20 x 5 = " +
          perform.operate(20, 10, multi));
          System.out.println("Division operation without return statement: 20 / 5
          = " + perform.operate(20, 10, div));
     }
     interface MathOperation {
        int operation(int ab, int xy);
     }
     private int operate(int ab, int xy, MathOperation mathOperation) {
        return mathOperation.operation(ab, xy);
     }
}
```

Output:

```
Addition operation with Type declaration: 20 + 5 = 30
Subtraction operation without Type declaration: 20 - 5 = 10
Multiplication with return statement: 20 x 5 = 200
Division operation without return statement: 20 / 5 = 2
```

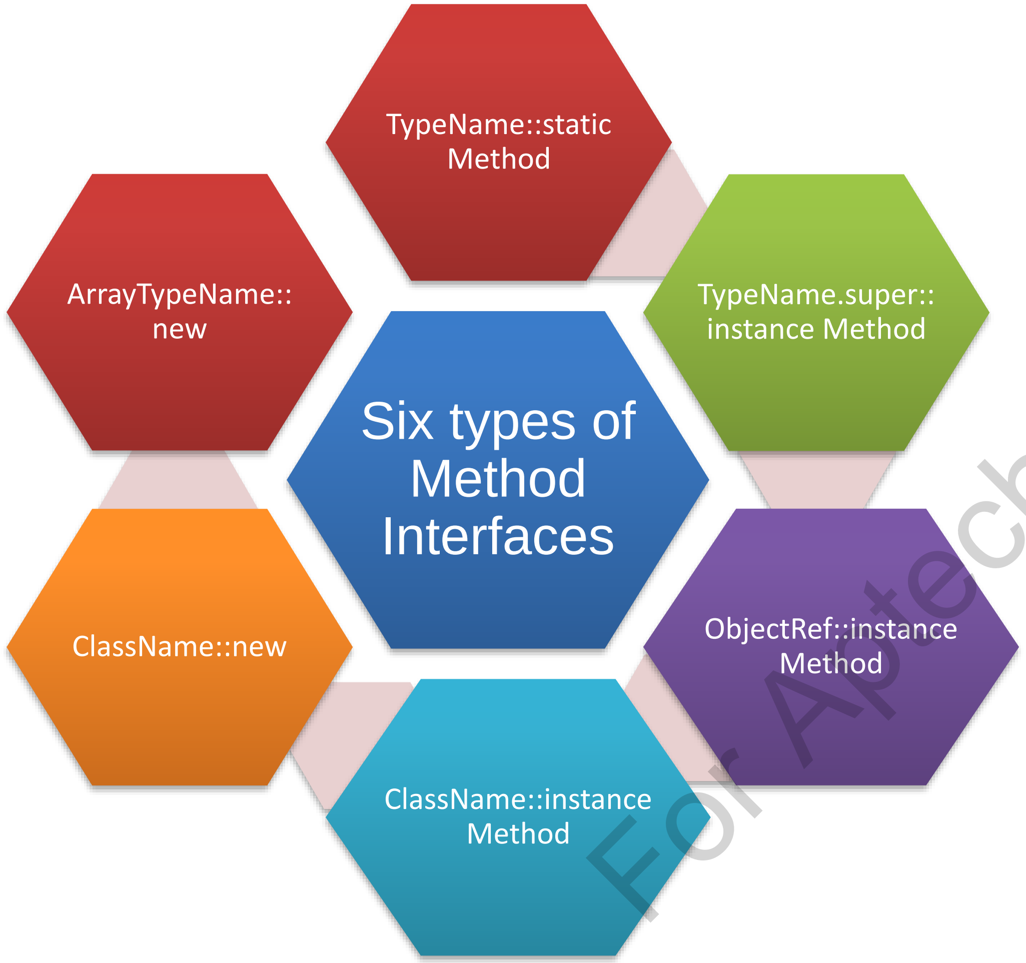
Scope for Lambda Expressions

```
import static java.lang.System.out;
/**
*Scope of Lambda example*/
public class MyWishes {
     Runnable dA = () ->out.println(this);
     Runnable dB = () ->out.println(toString());
     public String toString() { return "Happy New Year!"; }
     public static void main(String args[]) {
          new MyWishes().dA.run(); //Happy New Year
          new MyWishes().dB.run(); //Happy New Year
     }
}
```

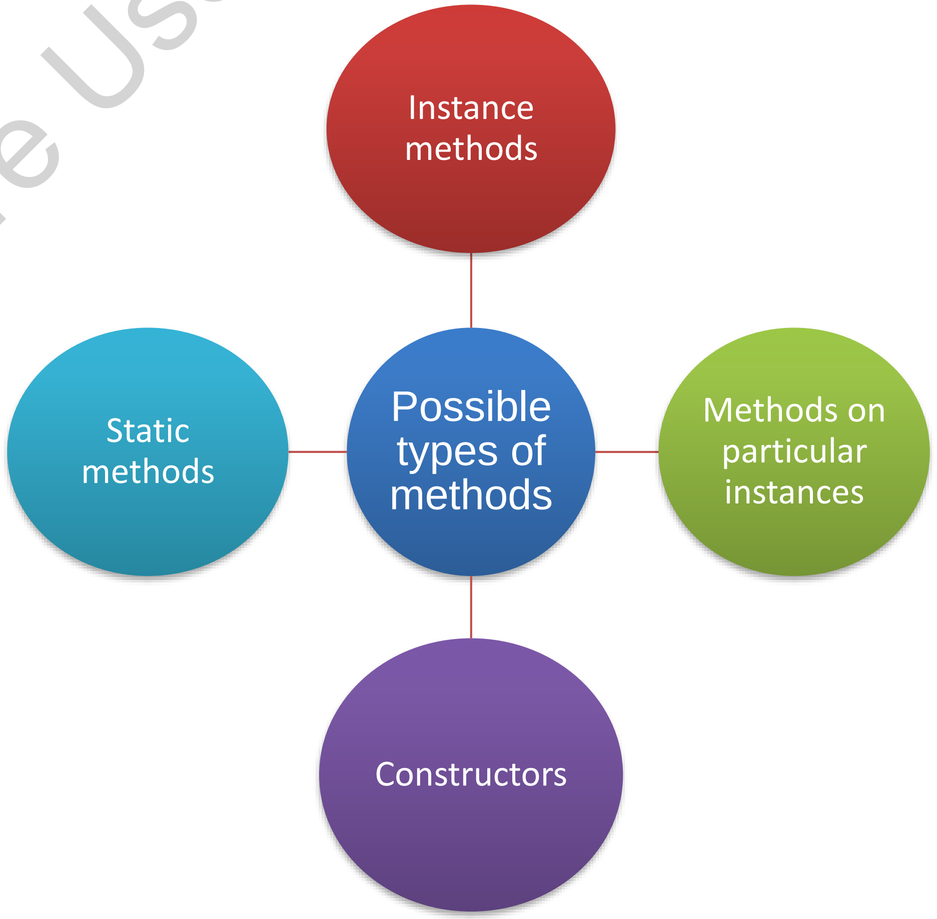
Uses lambda expressions with `Runnable` interface.

Method References

Method references represent easy-to-read lambda expressions for methods already having a name



Six types of Method Interfaces
TypeName::static Method
TypeName.super:: instance Method
ObjectRef::instance Method
ClassName::instance Method
ClassName::new
ArrayTypeName:: new



Static Method References

- A static method reference facilitates use of a static method as a lambda expression.
- Static methods can be defined in an enum, a class, or an interface.

```
import java.util.function.Function;
public class MainTest {
public static void main(String[] argv) {
// To retrieve result with a lambda expression
Function<Integer, String>funcA = x ->Integer.toBinaryString(x);
System.out.println(funcA.apply(11));
// To retrieve result with a method reference
Function<Integer, String>funcB = Integer::toBinaryString;
System.out.println(funcB.apply(11));
}
}
```

Output:

```
1011
1011
```

Instance Method References

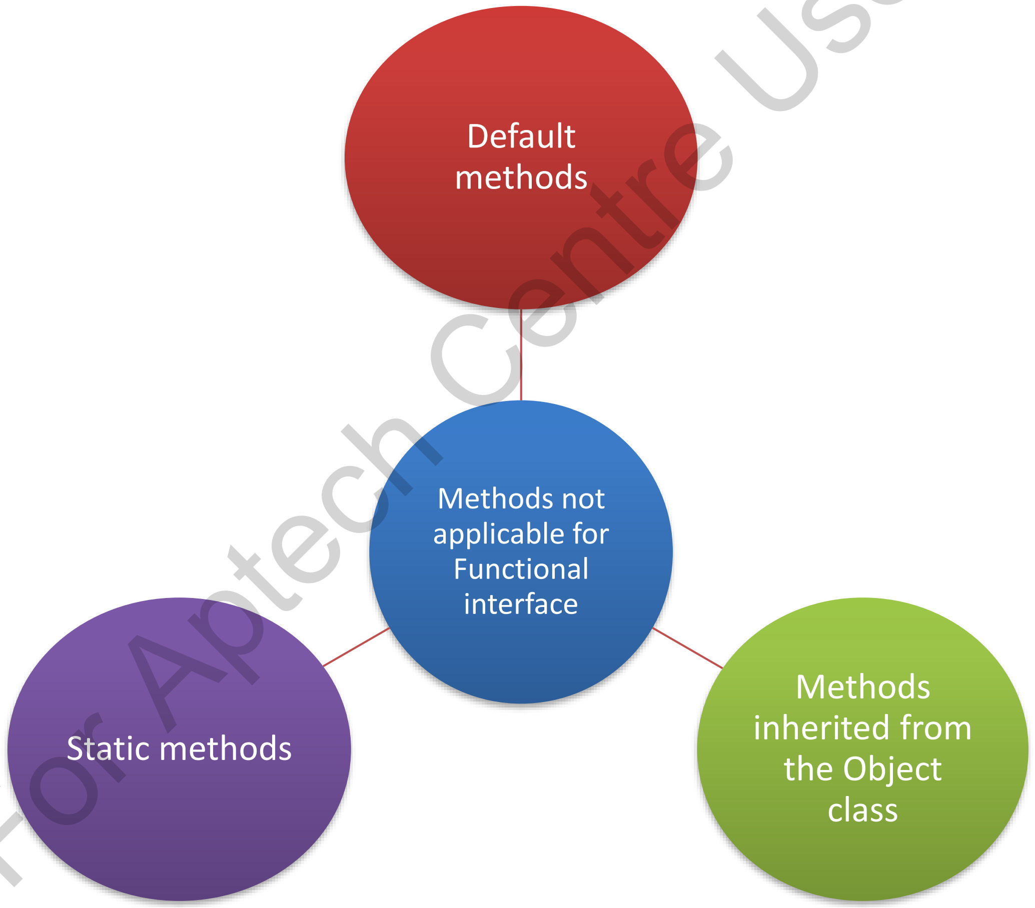
```
import java.util.function.Supplier;
public class MainTest2{
     public static void main(String[] argv){
        Supplier<Integer> sampleSupA = () -> "Aptech".length();
        System.out.println(sampleSupA.get()); // display result
        Supplier<Integer> sampleSupB = "Aptech"::length;
        System.out.println(sampleSupB.get()); // display result
}
}
```

Output:

```
6
6
```

Functional Interface 1-2

A functional interface is an interface with one method and is used as the type of a lambda expression.


Default methods
Methods not applicable for Functional interface
Static methods
Methods inherited from the Object class

Functional Interface 2-2

New functional interfaces included in package `java.util.function` from Java 8 onwards are as follows:

Predicate<T> - returns a Boolean value based on input of type T.
Supplier<T>- returns an object of type T.
Consumer<T> - performs an action with given object of type T.
Function<T, R> - gets an object of type T and returns R.
BiFunction – similar to Function, but with two parameters.
BiConsumer - similar to Consumer, but with two parameters.

Functional interface also contains several corresponding interfaces for primitive types, such as:

`IntSupplier`

`IntFunction<R>`

`IntPredicate`

`IntConsumer`

Default Methods

- Default methods reduce the difference among interfaces and abstract classes
- These methods eliminate requirement of utility classes
- Enhance the collection API and makes them lambda expression friendly
- Default methods can assist in the base implementation class removal
- Default methods become useless if any class in the hierarchy contains a method having same signature
- A default method cannot override a method from `java.lang.Object`
- These methods extend interfaces without breaking implementation classes

Default Method and Regular Method 1-2

Default method contains default modifier - that is the main difference between a regular method and default method.

Methods in classes can access and modify method arguments and also fields of their class.

A default method can only access its arguments, interfaces do not have any state.

Enable adding new functionality to existing interfaces without impacting the current implementation of these interfaces.

Default Method and Regular Method 2-2

```
public class Java8Tester {
     public static void main(String args[]) {
          Gadget gadget = new SmartGadget();
          gadget.print();
      }
}
interface Gadget {
     default void print() {
        System.out.println("This is a Gadget!");
        }
       static void call() {
            System.out.println("With Calling feature!");
      }
}
interface TextMessage {
       default void print() {
           System.out.println("With Text Messaging feature!");
         }
}
class SmartGadget implements Gadget, TextMessage {
     public void print() {
          Gadget.super.print();
          TextMessage.super.print();
          Gadget.call();
          System.out.println("It is a Smartphone!");
     }
}
```

Output:

```
This is a Gadget!
With calling feature!
With Text Messaging feature!
It is a Smartphone!
```

Multiple Defaults

- A Java class can implement one or more interfaces and each interface can state default method through same method signature.
- Eventually, the inherited methods conflict with one another and cause errors.
- Thus, Java throws a compilation error, if it is not sure whether the class implements two or more interfaces defining the same default method.

Static Methods on Interfaces

- Easy to organize and access helper methods in libraries
- Eliminate the requirement of a separate class

```
public interface ProductInfo {

...

  static ProductId getProductId (String ProductString) {

    ...

    }

...

}
```

Local-Variable Syntax for Lambda Parameters

- The Local-Variable Syntax for Lambda Parameters was enhanced where `var` can now be used similar to local variables when declaring formal parameters of implicitly typed lambda expressions.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
class Book{
int id;
String title;
float price;
public Book(int id, String title, float price) {
super();
this.id = id;
this.title = title;
this.price = price;
}
}
public class VarDemo{
public static void main(String[] args) {
List<Book> list=new ArrayList<Book>();
//Adding Products
list.add(new Book(1,"Harry Potter and the Chamber of 
Secrets",250f));
list.add(new Book(3,"Keyboard Ninjas",300f));
list.add(new Book(2,"The Three Investigators Club",150f));
System.out.println("Sorting on the basis of title...");
```

```
// implementing lambda expression
Collections.sort(list,(var p1, var p2)->{
return p1.title.compareTo(p2.title);
});
for(Book p:list){
System.out.println(p.id+" "+p.title+" "+p.price);
}
}
}
```

Output:

```
Sorting on the basis of title...
1 Harry Potter and the Chamber of Secrets 250.0
3 Keyboard Ninjas 300.0
2 The Three Investigators Club 150.0
```

Summary

- Functional programming is a type of programming approach that emphasizes utilization of functions and writing code that does not change state.
- Using functional programming, you can pass functions as parameters to other functions and return them as values.
- A lambda expression is a compact expression that does not require a separate class/function definition. It facilitates functional programming.
- Depending on the parameters being passed to the lambda expression, you will use/omit parentheses.
- Default method is a new feature in Java 8 that allows default implementation for methods in an interface.
- In addition to default methods, static methods can be defined in interfaces that makes it easy to organize and access helper methods in libraries.
- From Java 11 onwards, local-variable syntax is supported for lambda parameters.