Session: 15

Functional Programming in Java

Objectives

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

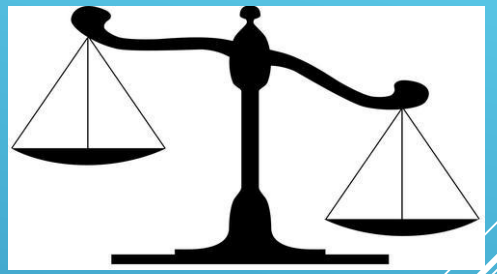
Introduction 1/2

Lambda expressions are newly introduced in Java 8 to facilitate functional programming.

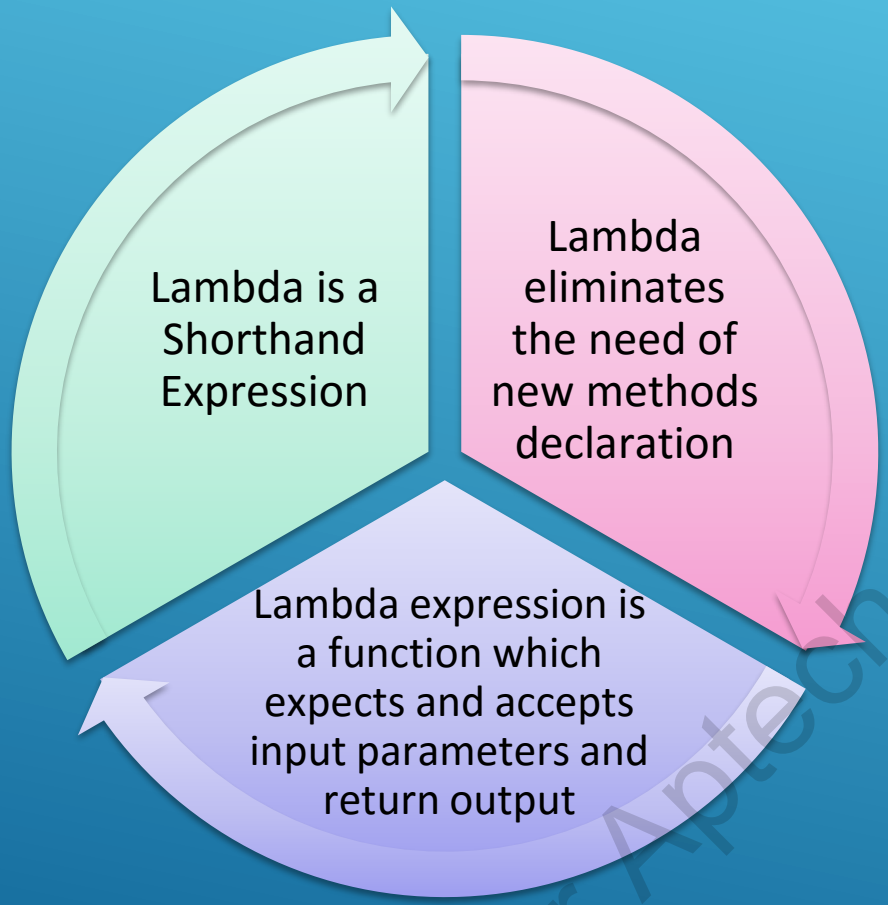
Introduction 2/2

Advantages of Functional Programming:

- Easy to Test Programs
- Thread Safe
- Modular



Lambda Expressions



Syntax of Lambda Expressions

Syntax:

```
parameters → body
```

where,

`parameters` are variables

→ is the lambda operator

`body` is parameter values

Rules for Lambda Expressions

Rules

- Type declarations are optional
- Parentheses around parameters can be omitted
- Curly braces may or may not be present
- Return keyword may or may not be present
- Varied number of statements

Single Method Interface and Lambdas 1/3

- Functional programming creates event listeners.
- Event listeners are defined as Java interfaces.

A single method interface Code Snippet where `State` is already declared will be:

```
interface StateChangeListener {
public void onStateChange(State previousState,
State presentState);
}
```

Single Method Interface and Lambdas 2/3

Code Snippet shows adding an event listener using an anonymous method implementation.

```
public class StateTest {
public static void main(String args[]) {
StateTest objStateTest = new StateTest();
objStateTest.addStateListener(new StateChangeListener() {
public void onStateChange(State previousState, State presentState) {
// action statements.
System.out.println("State change event occurred");
}
});
}
}
```

Single Method Interface and Lambdas 3/3

Code Snippet shows adding an event listener using a lambda expression.

```
public class StateTest {
public static void main(String args[]) {
StateTest objStateTest = new StateTest();
objStateTest.addStateListener((previousState,
presentState) →
System.out.println("State change event occurred"));
}
}
```

Lambda Parameters 1/3

Types of Lambda Parameters	Description
★ Zero Parameters	Parentheses with no comments
★ One Parameter	Parentheses contains a Value

Lambda Parameters 2/3

Types of Lambda Parameters	Description
★ Multiple Parameters	Added within parentheses
★ Parameters Types	Compiler is inconclusive about the parameter type

Lambda Parameters 3/3

Following Code Snippets show different types of parameters for lambdas:

Zero Parameter

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

One Parameter

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

Multiple Parameters

```
(pA, pB) → System.out.println("Multiple parameters defined: " + pA + ", " + pB);
```

Parameter Types

```
(Phone smartphone) → System.out.println("The smartphone is: " + smartphone.getName());
```

Returning a Value

For returning values from lambda expression, a `return` statement can be added for specific calculations.

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

Lambdas as Objects

Java lambda is an object and it can be assigned as a regular object to a variable.

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

Implementation of lambda where the lambda object is assigned to a variable and passed as an object.

```
SampleComparator sampleComparator = (iA, iB) →
return iA>iB;
boolean result = sampleComparator.compare(3, 6);
```

Code Snippet shows a lambda used to sort strings by length.

```
Arrays.sort(sampleStrArr,
(String strA, String strB) → strB.length() -
strA.length());
```

Advantages and Uses of Lambda Expressions 1/3

More readable code

Rapid fast coding specifically in Collections

Much easier parallel processing

Advantages and Uses of Lambda Expressions 2/3

Following Code Snippet shows a complete program utilizing lambda:

```
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);
}
}
```

Advantages and Uses of Lambda Expressions 3/3

Following is the output for complete program utilizing lambda:

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
```

Here, the code performs basic math operations using lambda expressions.

Scope for Lambda Expressions

Code Snippet uses lambda expressions with `Runnable` interface:

```
import static java.lang.System.out;
/**Aptech Java8
*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
}
}
```

Both the `dA` and `dB` lambdas call the `toString()` method of the `MyWishes` class.
This shows the scope available to the lambda.

Method References 1/4

Refers to constructors or methods.

Following are six types of method references:

`TypeName::static`	Refers to a static method of a class, an `enum`, or an interface
`TypeName.super::instance`	Refers to an instance method from the supertype of an object
`ObjectRef::instance`	Refers to an instance method
`ClassName::instance`	Refers to an instance method of a class
`ClassName::new`	Refers to the constructor from a class
`ArrayTypeName::new`	Refers to the constructor of the specified array type

Method References 2/4

Methods to determine a file type

Frequent filter of a list of files based on file types can be made by determining a file type.

```
public class FileFilters {// to filter files
public static boolean fileIsJpeg(File file)
{/*sample code*/}
public static boolean fileIsTiff(File file)
{/*sample code*/}
public static boolean fileIsPng(File file)
{/*sample code*/}
}
```

Method References 3/4

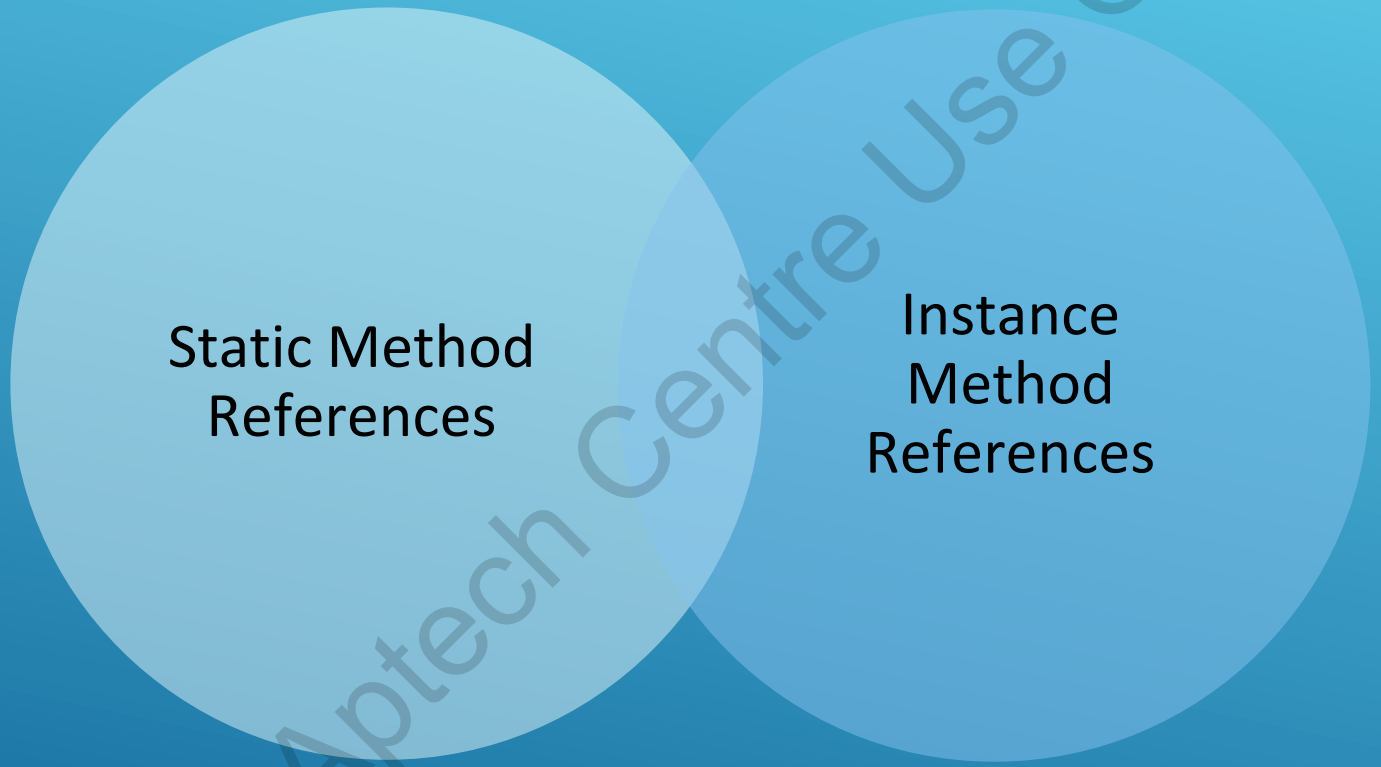
File Filtering Case

Method reference can be useful in file filtering cases.
In the following Code Snippet, a method is predefined as `getFiles()` that returns a Stream:

```
Stream<File> Jpegs =
getFiles().filter(FileFilters::fileIsJpeg);
Stream<File> Tiffs =
getFiles().filter(FileFilters::fileIsTiff);
Stream<File> Pngs =
getFiles().filter(FileFilters::fileIsPng);
```

Method References 4/4

Types of Method Reference



Static Method References

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
```

The first lambda expression funcA is created by defining an input value x and providing a lambda expression body. This is the normal way of creating a lambda expression.
The second lambda expression funcB is created by referencing a static method from Integer class.

Instance Method References

```
import java.util.function.Supplier;
public class MainTest{
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.

New functional interfaces included in the Java 8 package, `java.util.function`, are:

- **`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 2/2

The second line defines a function that represents '@' symbol to a String.

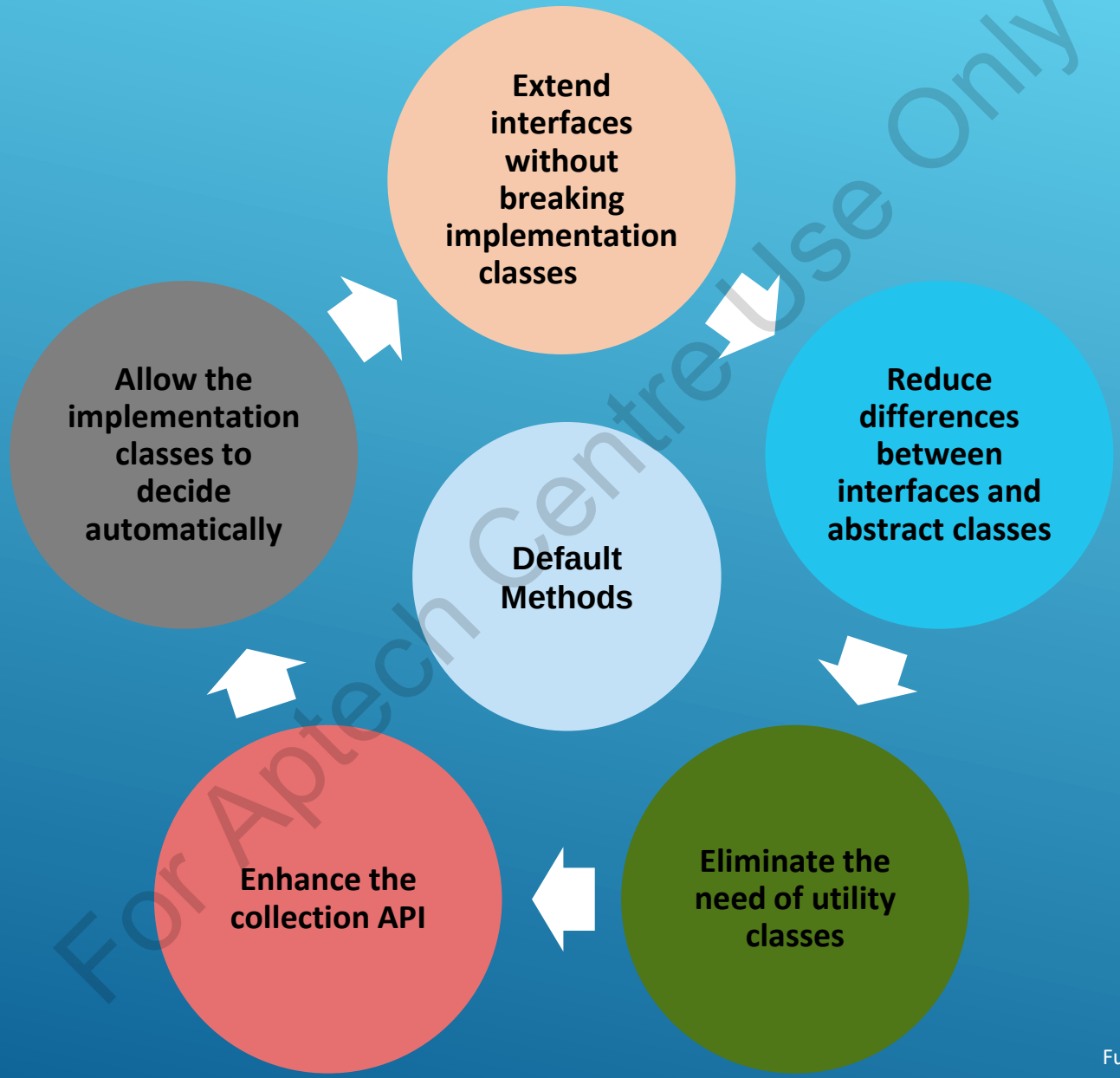
```
//Functional Interface sample use case
Function<String, Integer>sampleLengthA = (name) ->
name.length();//as int
Function<String, String>atr = (name) -> {return "@" +
name;};//as string
Function<String, Integer>samplLengthB = String::length;
//as int
```

Default Methods

Default methods are Virtual Extensions that contains default implementations of `forEach()` method.

```
class Test{
public static void main(String args[]){
Book book = new Novel();
book.print();
}
}
interface Book {
default void print(){
System.out.println("This is a book");
}
static void turnPages(){
System.out.println("Turning pages.");
}
}
interface Journal {
default void print(){
System.out.println("This is a journal");
}
}
class Novel implements Book, Journal {
public void print(){
Book.super.print();
Journal.super.print();
Book.turnPages();
System.out.println("This is a novel");
} }
```

Features of Default Methods



Default Method and Regular Method

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

```
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 1/3

Multiple Defaults are multiple interfaces contained within Java class. Java throws a compilation error if two or more interfaces defining the same default method.

```
public interface Green {
default void defaultMethod(){
System.out.println("Green default method");
}
}
public interface Red {
default void defaultMethod(){
System.out.println("Red default method");
}
}
public class Impl implements Green, Red{
}
```

Multiple Defaults 2/3

Compiling the previous Code Snippet will result in an error. In order to fix this, provide explicit default method implementation.

```
public class Impl implements Green, Red{
      public void defaultMethod() {
      …
       }
}
```

Multiple Defaults 3/3

Further, to invoke default implementation provided by any of super interfaces, the code can be as follows:

```
public class Impl implements Green, Red {
public void defaultMethod(){
// remaining code...
Green.super.defaultMethod();
}
}
```

Static Methods on Interfaces 1/2

Static Methods on Interfaces	Description
★ Easy to organize and access helper method in libraries	For example, the new Stream interface contains many static methods
★ Eliminates the need of a separate class	Parentheses contains a value

Static Methods on Interfaces 2/2

All method declarations in an interface, including static methods, are implicitly `public`.

```
public interface ProductInfo {
...
static ProductId getProductId (String
ProductString) {
...
}
...
}
```

Summary

- Functional programming emphasizes that 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.