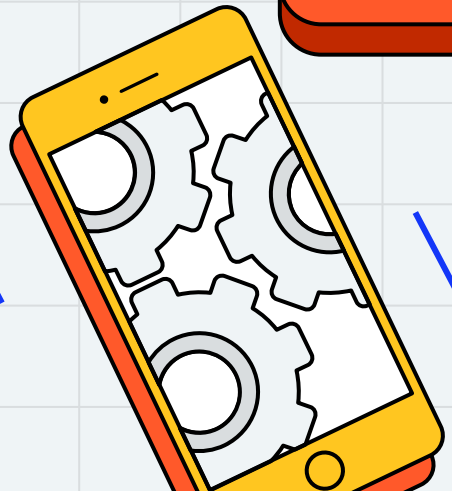
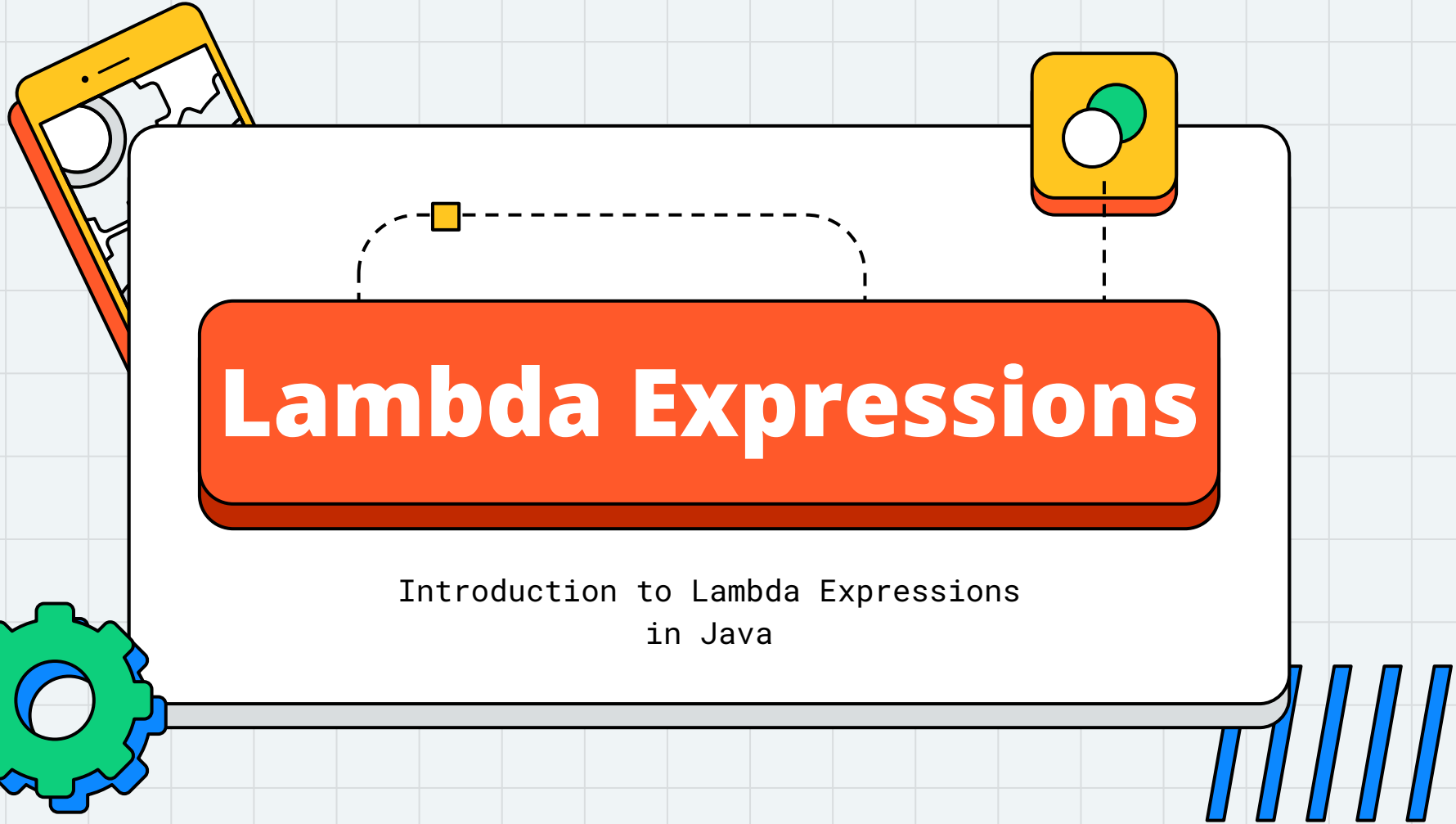
Application Programming



Hend Alkittawi


Lambda Expressions
Introduction to Lambda Expressions
in Java

INTRODUCTION

- To understand Java's implementation of lambda expressions we need to understand what **Functional Interfaces** and a **Lambda Expressions** are!
- A **Functional Interface** is an interface that contains **one and only one abstract method**
- A **Lambda Expression** is an anonymous (unnamed) **method**, used to implement a method defined by a functional interface
- A functional interface defines a **target type** of a lambda expression!

LAMBDA EXPRESSIONS FUNDAMENTALS

- Lambda expressions use the **lambda operator** (->)
  - left-side: specifies any parameters requires by the lambda expression
  - right-side: the lambda body which specifies the actions of the lambda expression

```
(parameters) -> (body)
```

- Lambda body can be
  - single expression
    - () -> (123.45)
    - () -> (Math.random() * 100)
    - (n) -> ((n%2) == 0)
  - block of code
    - (n) -> { for(int i = 0; i < n; i++)
      System.out.println(i); }

LAMBDA EXPRESSIONS FUNDAMENTALS

- A lambda expression is not executed on its own; it forms the ***implementation of the abstract method defined by the functional interface*** that specifies its target type!
- The lambda expression can be specified only in a context in which a target type is specified.
  - one of these contexts is created when a lambda expression is assigned to a functional interface reference.

```
FunctionalInterface var = (parameters) -> (body);
```

LAMBDA EXPRESSIONS - SINGLE EXPRESSION

- An instance of a class is automatically created that implements the functional interface, with the lambda expression defining the behavior of the abstract method declared by the functional interface.
- When that method is called through the target, the lambda expression is executed
- The lambda expression gives us a way to **transform a code segment into an object**!

```
public interface MyNumber {
	public double getValue();
}
```

```
public class MyFirstNumber implements MyNumber{
	@Override
	public double getValue() {
		return 100;}
}
```

```
public class MySecondNumber implements MyNumber{
	@Override
	public double getValue() {
		return Math.random() * 100; }
}
```

```
public class LambdaDemo {
    public static void main(String[] args) {

        MyFirstNumber first = new MyFirstNumber();
        double m = first.getValue();

        MySecondNumber second = new MySecondNumber();
        double n = second.getValue();

        MyNumber x = () -> 100;
        MyNumber y = () -> Math.random() * 100;

        System.out.println("m: " + m + " n: " + n +
                           " x: " + x.getValue() + " y: " + y.getValue());
        // MyNumber z = () -> "123.5";
}
```

LAMBDA EXPRESSIONS - SINGLE EXPRESSION

- The type of the parameter (n) is not specified; it is inferred from the context.
    - The parameter type of test()
- It is possible to explicitly specify the type of the parameter in a lambda expression
    - (int n) -> (n % 2) == 0

```
public interface NumericTest {
       public boolean test(int n);
}
```

```
public class LambdaDemo {
    public static void main(String[] args) {

       NumericTest isEven = (n) -> (n % 2) == 0;

       System.out.println(isEven.test(5));
       System.out.println(isEven.test(6));

       NumericTest isPositive = (n) -> (n > 0);

       System.out.println(isPositive.test(-1));
       System.out.println(isPositive.test(1));
    }
}
```

LAMBDA EXPRESSIONS - CODE BLOCK

- A block lambda encloses the body within braces { }
- The block body of a lambda is similar to a method body

```
public interface NumericFunction {
       public int func(int n);
}
```

```
public interface StringFunction {
       public String func(String n);
}
```

```
public class LambdaDemo {

    public static void main(String[] args) {

       NumericFunction factorial = (n) -> { int result = 1;
                                            for (int i = 1; i <= n; i++) {
                                                result = result * i;     }
                                            return result;
                                           };
       System.out.println(factorial.func(5));

       StringFunction reverse = (str) -> { String result = "";
                                           for (int i = str.length() - 1; i >= 0; i--) {
                                                result = result + str.charAt(i);       }
                                           return result;
                                          };
       System.out.println(reverse.func("Lambda"));
    }
}
```

LAMBDA EXPRESSIONS FUNDAMENTALS

- The functional interface associated with a lambda expression can be **generic**.

```
public interface NumericFunction {
       public int func(int n);
}
```

```
public interface StringFunction {
       public String func(String n);
}
```

```
public interface SomeFunction<T> {
       public T func(T t);
}
```

```
public class LambdaDemo {
    public static void main(String[] args) {

       SomeFunction<Integer> factorial = (n) -> { int result = 1;
                                                  for (int i = 1; i <= n; i++){
                                                      result = result * i;    }
                                                  return result;
                                                 };
       System.out.println(factorial.func(5));

       SomeFunction<String> reverse = (str) -> { String result = "";
                                                 for (int i = str.length() - 1; i >= 0; i--){
                                                      result = result + str.charAt(i);      }
                                                 return result;
                                                };
       System.out.println(reverse.func("Lambda"));
}
}
```

LAMBDA EXPRESSION FUNDAMENTALS

- A lambda expression can be passed as **an argument** to a method.
- A very powerful use of lambda expressions which gives you a way to pass executable code as an argument to a method
- The **type of the parameter** receiving the lambda expression must be of a **functional interface** compatible with the lambda

```
public interface StringFunction {
       public String func(String n);
}
```

```
public class LambdaDemo {

    public static void main(String[] args) {

       String inStr = "Lambdas Add Power To Java!";

       StringFunction capitalize = (str) -> str.toUpperCase();
       String outStr1 = stringOp(capitalize, inStr);

       String outStr2 = stringOp( (str) -> str.toLowerCase(), inStr);
       System.out.println("inStr: " + inStr
                          + " outStr1: " + outStr1
                          + " outStr2: " + outStr2);
    }

    public static String stringOp(StringFunction sf, String s) {
       return sf.func(s);
    }
}
```

PREDEFINED FUNCTIONAL INTERFACES

- The previous examples have defined their own functional interfaces.
- In many cases, there is no need to define your own functional interface; Java's java.util.function package provides several predefined functional interfaces!

Interface	Method	Parameter(s)	Returns
`Function<T,R>`	`apply`	`T`	`R`
`Predicate<T>`	`test`	`T`	`boolean`
`UnaryOperator<T>`	`apply`	`T`	`T`
`BinaryOperator<T>`	`apply`	`T, T`	`T`

FUNCTION<T, R> INTERFACE

- The **Function** interface requires a parameter type and a return type.

Interface	Method	Parameter(s)	Returns
Function<T,R>	apply	T	R

```
public interface Function<T, R> {
        public R apply(T t);
}
```

- A usage example …

```
Function<Integer,Double> getHalf = (x) -> (x / 2.0);
double result = getHalf.apply( 5 );
```

PREDEFINED FUNCTIONAL INTERFACES - EXAMPLE

```
public interface NumericFunction {
       public int func(int n);
}
```

```
public class LambdaDemo {

    public static void main(String[] args) {

       NumericFunction factorial = (n) -> { int result = 1;
                                            for (int i = 1; i <= n; i++)
                                               result = result * i;
                                            return result;
                                          };
       System.out.println(factorial.func(5));
    }
}
```

Interface	Method	Parameter(s)	Returns
Function<T,R>	apply	T	R

```
public interface Function<T, R> {
       public R apply(T t);
}
```

```
public class LambdaDemo {
    public static void main(String[] args) {
       Function<Integer, Integer> factorial = (n) -> { int result = 1;
                                                       for (int i = 1; i <= n; i++)
                                                            result = result * i;
                                                       return result;
                                                     };
              System.out.println(factorial.apply(5));
    }
}
```

PREDICATE<T> INTERFACE

- The **Predicate** interface requires a parameter type and returns a boolean.

Interface	Method	Parameter(s)	Returns
Predicate<T>	test	T	boolean

```
public interface Predicate<T> {
        public boolean test(T t);
}
```

- A usage example …

```
Predicate<Double> checkPassing =(grade)->(grade >= 60);
boolean isPassing = checkPassing.test( 68.5 );
```

UNARY OPERATOR

- The **UnaryOperator** interface requires a parameter type and returns a value of the same type

Interface	Method	Parameter(s)	Returns
UnaryOperator<T>	apply	T	T

```
public interface UnaryOperator<T> {
        public T apply(T t);
}
```

- An usage example …

```
UnaryOperator<Integer> uSquare = (i) -> (i*i);
int result = uSquare.apply( 3 );
```

BINARY OPERATOR

- The **BinaryOperator** interface requires a parameter type and returns a value of the same type.

Interface	Method	Parameter(s)	Returns
BinaryOperator<T>	apply	T, T	T

```
public interface BinaryOperator<T> {
        public T apply(T t1, T t2);
}
```

- A usage example

```
BinaryOperator<Integer> bMult = (a, b)-> (a * b);
int result = bMult.apply( 5, 2 );
```

LAMBDA EXPRESSIONS AND ANONYMOUS INNER CLASSES

- Inner classes are classes defined within another class.
- An anonymous inner class is a class without a name, for which only one object is created!

```
public static Comparator<Book> bookComparator = new Comparator<Book>() {
    public int compare(Book book1, Book book2) {
          return book1.getTitle().compareTo(book2.getTitle());
    }
};
```

- A lambda expression can be utilized instead of an anonymous inner class

```
public static Comparator<Book> bookComparator = (book1, book2) -> {
                                return book1.getTitle().compareTo(book2.getTitle());
                                                                    };
```

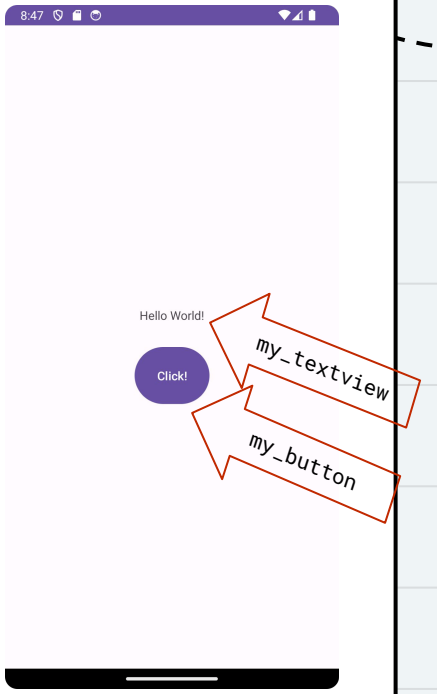
LAMBDA EXPRESSIONS AND ANONYMOUS INNER CLASSES

- In Android, recall that the **setOnClickListener()** method takes a listener as its argument. It takes an object that implements **View.OnClickListener**.
- The listener can be implemented as an anonymous inner class, which puts the implementation of the listeners' methods right where you want to see them.
- The syntax can be simplified by using lambda expressions!

LAMBDA EXPRESSIONS AND ANONYMOUS INNER CLASSES

- The listener can be implemented as an anonymous inner class

```
Button button = (Button) findViewById(R.id.my_button);
button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            TextView myTextView = (TextView) findViewById(R.id.my_textview);
            myTextView.setText("Salam!");
        }
});
```



- The syntax can be simplified by using lambda expressions!

```
Button button = (Button) findViewById(R.id.my_button);
button.setOnClickListener((view) ->  {
   TextView myTextView = (TextView) findViewById(R.id.my_textview);
   myTextView.setText("Salam!");
});
```

THANK YOU!

DO YOU HAVE ANY QUESTIONS?

hend.alkittawi@utsa.edu

By Appointment

Online