GUJARAT TECHNOLOGICAL UNIVERSITY

MASTER OF COMPUTER APPLICATIONS

SEMESTER: III

Subject : **4639302**

Programming in JAVA

UNIT - 3

- Lambda Expressions,
- Inner classes

Lambda Expressions

- why Lambdas?
- syntax of lambda expression,
- functional interfaces,
- method reference,
- constructor reference,
- variable scope,
- processing lambda expression

Introduction

- Lambda expressions is the biggest feature of Java 8.
- Lambda expression facilitates functional programming, and simplifies the development a lot.

Why Lambdas?

- Lambda expression is a block of code that you can pass around so it can be executed later.
- Reduces tedious boilerplate for callbacks.
- When sorting strings by length, the compare method had to be called repeatedly to compute
  - first.length() - second.length()
- Lambda expression:
  - (String first, String second) -> first.length() - second.length()

Why Lambdas?

- **Lambda expression:**
  - (String first, String second) ->
    first.length() - second.length()

- **Much simpler than:**

```
class LengthComparator implements Comparator<String>
{
    public int compare(String first, String second)
    {
        return first.length() - second.length();
    }
}
```

The Syntax of
Lambda Expressions

important characteristics of a lambda expression/syntax.

- **Optional type declaration**
  - No need to declare the type of a parameter.
  - The compiler can inference the same from the value of the parameter.
- **Optional parenthesis around parameter**
  - No need to declare a single parameter in parenthesis.
  - For multiple parameters, parentheses are required.
- **Optional curly braces**
  - No need to use curly braces in expression body if the body contains a single statement.
- **Optional return keyword**
  - The compiler automatically returns the value if the body has a single expression to return the value.
  - Curly braces are required to indicate that expression returns a value.

The Syntax of
Lambda Expressions

- **Simplest form: (parameters) -> expression**
- **If the code doesn't fit in a single expression, use { } and a return statement:**

```
(String first, String second) ->
{
  if (first.length() < second.length()) return -1;
  else if (first.length() > second.length()) return 1;
  else return 0;
}
```

- **If there are no parameters, you still supply parentheses:**
  - () -> Toolkit.getDefaultToolkit().beep();
- **If parameter types can be inferred, omit them:**
  - Comparator<String> comp = (first, second) -> first.length() - second.length();
- **If there is exactly one parameter with inferred type, omit parentheses:**
  - ActionListener listener = event -> Toolkit.getDefaultToolkit().beep();

Functional Interfaces

- **Functional interface is an Interface with a single abstract method.**
  - Examples: ActionListener, Comparator
- **Lambda expression can be used whenever a functional interface value is expected:**

```
Arrays.sort(words,    (first, second) -> first.length() - second.length());

Timer t = new Timer(1000, event ->
  {
    System.out.println("At the tone, the time is " + new Date());
    Toolkit.getDefaultToolkit().beep();
  });
```

- **Conversion to a functional interface is the only thing that you can do with a lambda expression.**

Method References

- **Consider a lambda expression that calls a single method:**
  - Timer t = new Timer(1000, event -> System.out.println(event));
- **You can use a method reference instead:**
  - Timer t = new Timer(1000, System.out::println);
- **Another example:**
  - Arrays.sort(words, String::compareToIgnoreCase)
- **Three cases:**
  - object::instanceMethod
  - Class::staticMethod
  - Class::instanceMethod

Constructor References

- **Person::new is a reference to a Person constructor.**
  - Same as s -> new Person(s)
  - The compiler uses overloading resolution to pick the correct constructor.
- **Example—turn list of names into list of Person objects:**
  - ArrayList<String> names = . . .;
  - Stream<Person> stream = names.stream().map(Person::new);
  - List<Person> people = stream.collect(Collectors.toList());
- **The map method turns a stream of strings into a stream of Person objects.**
- **Constructor references also work for arrays:**
  - int[]::new is the same as the lambda expression n -> new int[n]
  - Useful to overcome limitation of Java generics: Illegal to call new T[n]
  - Turn stream to array of the correct type:
    - Person[] people = stream.toArray(Person[]::new);

Variable Scope

- **A lambda expression can access variables from the enclosing scope:**

```
public static void repeatMessage(String text, int delay){
  ActionListener listener = event ->  {
    System.out.println(text);
    Toolkit.getDefaultToolkit().beep();
  };
  new Timer(delay, listener).start();
}
```

- **Consider a call:**
  - repeatMessage("Hello", 1000); // Prints Hello every 1,000 milliseconds
- **The text variable is not defined in the lambda expression.**
- **It is gone when repeatMessage returns!**
- **A lambda expression is a closure, containing:**
  - A block of code
  - Parameters
  - Values for the free variables

Effectively Final Variables

- **A lambda variable can only capture a variable whose value is effectively final:**

```
public static void countDown(int start, int delay)  {
  ActionListener listener = event ->   {
    start--; // Error: Can't mutate captured variable
    System.out.println(start);
  };
  new Timer(delay, listener).start();
}
```

- **Also illegal if the variable changes outside the lambda expression:**

```
public static void repeat(String text, int count) {
  for (int i = 1; i <= count; i++)    {
    ActionListener listener = event ->
      System.out.println(i + ": " + text); // Error: Cannot refer to changing i
    new Timer(1000, listener).start();
  }
```

Processing Lambda Expressions

- **Repeat an action n times:**

```
repeat(10, () -> System.out.println("Hello, World!"));
```

- **Pick a functional interface for the second parameter:**

```
public static void repeat(int n, Runnable action) {
  for (int i = 0; i < n; i++) action.run();
}
```

- **To pass the count to the action, pick a functional interface from the java.util.function package:**

```
public static void repeat(int n, IntConsumer action) {
  for (int i = 0; i < n; i++) action.accept(i);
}
```

- **Called like this:**

```
repeat(10, i -> System.out.println("Countdown: " + (9 - i)));
```

Common Functional Interfaces

Functional Interface	Parameter Types	Return Type	Abstract Method Name	Description	Other Methods
`Runnable`	none	`void`	`run`	Runs an action without arguments or return value	
`Supplier<T>`	none	`T`	`get`	Supplies a value of type `T`	
`Consumer<T>`	`T`	`void`	`accept`	Consumes a value of type `T`	`andThen`
`BiConsumer<T, U>`	`T, U`	`void`	`accept`	Consumes values of types `T` and `U`	`andThen`
`Function<T, R>`	`T`	`R`	`apply`	A function with argument of type `T`	`compose`, `andThen`, `identity`

Common Functional Interfaces

BiFunction<T, U, R>	T, U	R	apply	A function with arguments of types T and U	andThen
UnaryOperator<T>	T	T	apply	A unary operator on the type T	compose, andThen, identity
BinaryOperator<T>	T, T	T	apply	A binary operator on the type T	andThen, maxBy, minBy
Predicate<T>	T	boolean	test	A boolean-valued function	and, or, negate, isEqual
BiPredicate<T, U>	T, U	boolean	test	A boolean-valued function with two arguments	and, or, negate

Functional Interfaces

for Primitive Types

Functional Interface	Parameter Types	Return Type	Abstract Method Name
BooleanSupplier	none	boolean	getAsBoolean
*P*Supplier	none	*p*	getAs*P*
*P*Consumer	*p*	void	accept
Obj*P*Consumer<T>	T, *p*	void	accept
*P*Function<T>	*p*	T	apply
*P*To*Q*Function	*p*	*q*	applyAs*Q*
To*P*Function<T>	T	*p*	applyAs*P*
To*P*BiFunction<T, U>	T, U	*p*	applyAs*P*
*P*UnaryOperator	*p*	*p*	applyAs*P*
*P*BinaryOperator	*p*, *p*	*p*	applyAs*P*
*P*Predicate	*p*	boolean	test

More about Comparators

- **Comparator interface has useful methods for creating and composing comparators.**
- **The static method comparing makes a comparator from a key extractor function:**

```
Arrays.sort(people, Comparator.comparing(Person::getName));
```

- **If the key is a primitive type, use comparingInt or comparingDouble to avoid boxing:**

```
Arrays.sort(words, Comparator.comparingInt(String::length));
```

- **The default method thenComparing chains comparators:**

```
Arrays.sort(people,  Comparator.comparing(Person::getLastName)
    .thenComparing(Person::getFirstName));
```

- **sort people by the length of their names using Lambda**

```
Arrays.sort(people, Comparator.comparing(Person::getName,
        (s, t) -> Integer.compare(s.length(), t.length())));
```

or

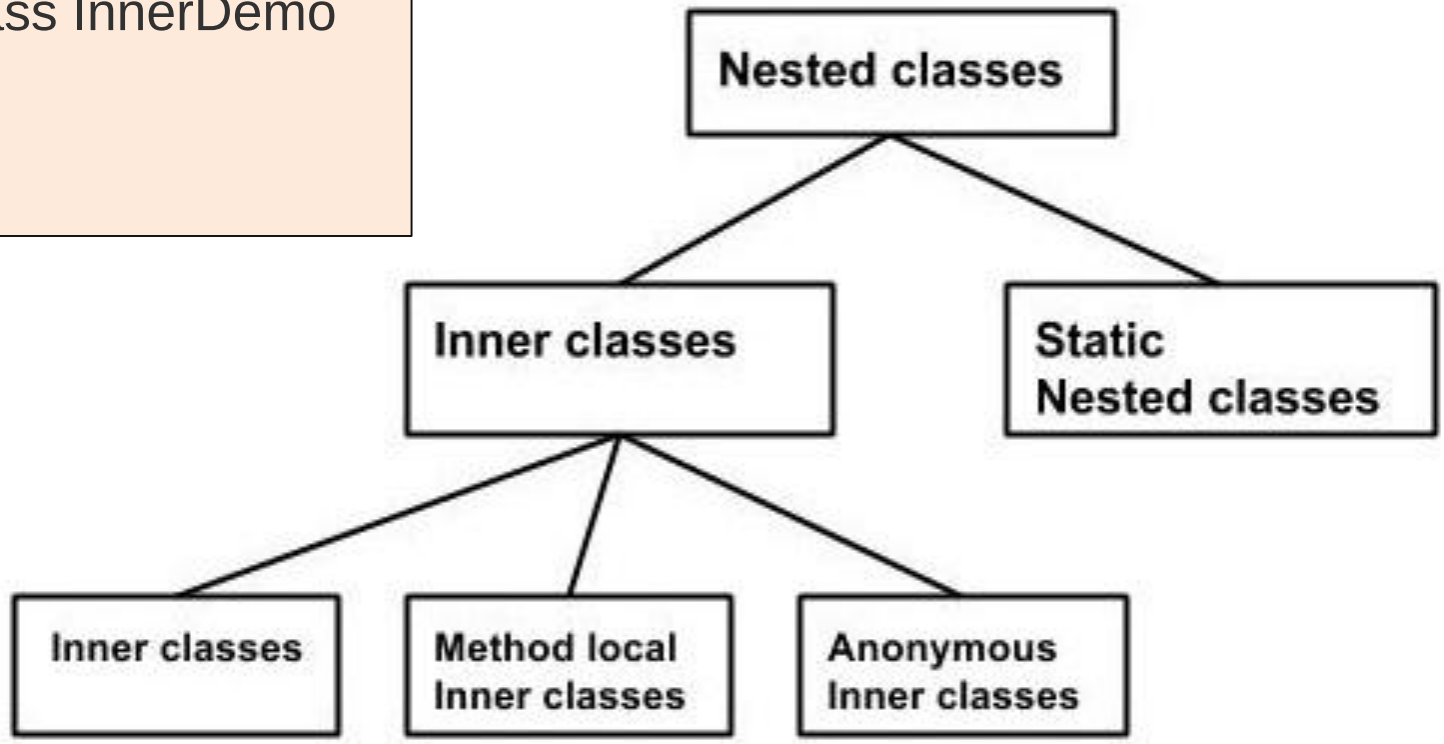
```
Arrays.sort(people, Comparator.comparingInt(p -> p.getName().length()));
```

Inner Classes

Inner class is a Class that is defined inside another class.

Inner Class / Nested class

```
class OuterDemo
{
      class InnerDemo
      {
      }
}
```



Type of Inner Class

- Inner classes are a security mechanism in Java.
- We know a class cannot be associated with the access modifier **private**, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.
- Type of Inner Class
  - Method-local Inner Class
  - Anonymous Inner Class
  - Static Inner Class

Inner Class

- Creating an inner class is quite simple.
- You just need to write a class within a class.
- Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

```
class OuterDemo
{
  int num;

  // inner class
  private class InnerDemo
  {
    public void print() {
      System.out.println("This is an inner class");
    }
  }
  // Accessing he inner class from the method within
  void displayInner()
  {
    InnerDemo inner = new InnerDemo();
    inner.print();
  }
}
public class MyClass  {
  public static void main(String args[])
  {
    // Instantiating the outer class
    OuterDemo outer = new OuterDemo();

    // Accessing the displayInner() method.
    outer.displayInner();
  }
}
```

Accessing the Private Members

```
class OuterDemo {
    // private variable of the outer class
    private int num = 1984;
    public class InnerDemo  {
        public int getNum()  {
                System.out.println("method of the inner class");
                return num; //accessing outer class private variable
        }
    }
}
public class PrivateMemberDemo  {
    public static void main(String args[])   {
            // Instantiating the outer class
            OuterDemo outer = new OuterDemo();
            // Instantiating the inner class
            OuterDemo.InnerDemo inner = outer.new InnerDemo();
            System.out.println(inner.getNum());
     }
}
```

Method-local Inner Class

```
public class LocalInnerClassDemo  {
  void myMethod() {
          int num = 14;
          class MethodInnerDemo  {     //local class
                public void print() {   System.out.println("num : "+num);
          }
    }
    // Accessing the inner class
    MethodInnerDemo inner = new MethodInnerDemo();
    inner.print();
  }
  public static void main(String args[])  {
     LocalInnerClassDemo outer = new LocalInnerClassDemo();
     outer.myMethod();
  }
}
```

Anonymous Inner Class

- An inner class declared without a class name is known as an **anonymous inner class**.
- In case of anonymous inner classes, we declare and instantiate them at the same time.
- Generally, they are used whenever you need to override the method of a class or an interface.

```
abstract class X {
   public abstract void myMsg();
}
public class AnonymousInnerDemo
{
   public static void main(String args[])
   {
      X obj = new X()
      {
            // anonymous inner class
            public void myMsg()
            {
                System.out.println("hi");
             }
       };
       obj.myMsg();
   }
}
```

Static Inner/Nested Class

- A static inner class is a nested class which is a static member of the outer class.
- It can be accessed without instantiating the outer class, using other static members.
- Just like static members, a static nested class does not have access to the instance variables and methods of the outer class.

Example

```
public class StaticInnerClassDemo {
        static class X    {
            public void msg()  {
                    System.out.println("This is nested class");
            }
        }
        public static void main(String args[])    {
                StaticInnerClassDemo.X obj = new StaticInnerClassDemo.X();
                obj.msg();
        }
}
```

Contact:

Bipin S. Rupadiya

Assistant Professor, JVIMS

Mo. : +91-9228582425

Email: info@bipinrupadiya.com

Blog : www.BipinRupadiya.com