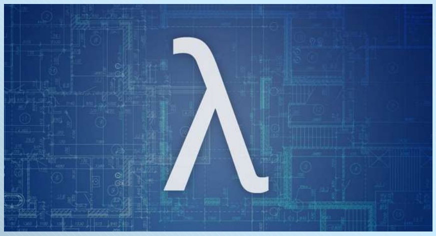
Lambda Expression & Functional Programming Part 1



Objectives

- At the end of this topic, you should be able to
  - Understand lambda expression and its use.
  - Understand Java functional interface.
  - Explain the syntax of lambda expression.
  - Understand functional programming with lambda.
  - Understand variable capture in lambda expression.

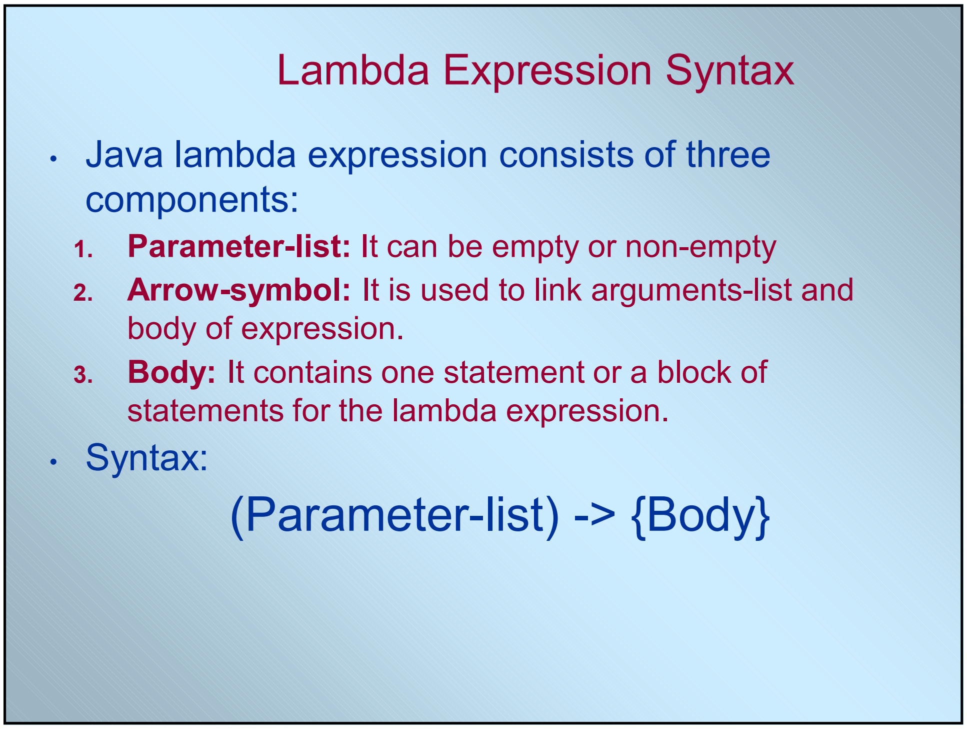
What is a Lambda Expression?

- Lambda expression is a new and important feature of Java which was included in Java SE 8.
- Provides a clear and concise way to implement a functional interface
- Also known as anonymous method.
- A step towards functional programming in Java (treats code/function as data)
- Java lambda expression is treated as a function (compiler does not create .class file)

Functional Interface

- A lambda interface implements a functional interface
- A functional interface is a Java interface
- It can only have one abstract method (aka Single Abstract Method (SAM) interface)
- Java has many predefine functional interfaces such as Comparator, ActionListener, Runnable and those in the java.util.function package
- The @FunctionalInterface annotation can be used when defining a functional interface

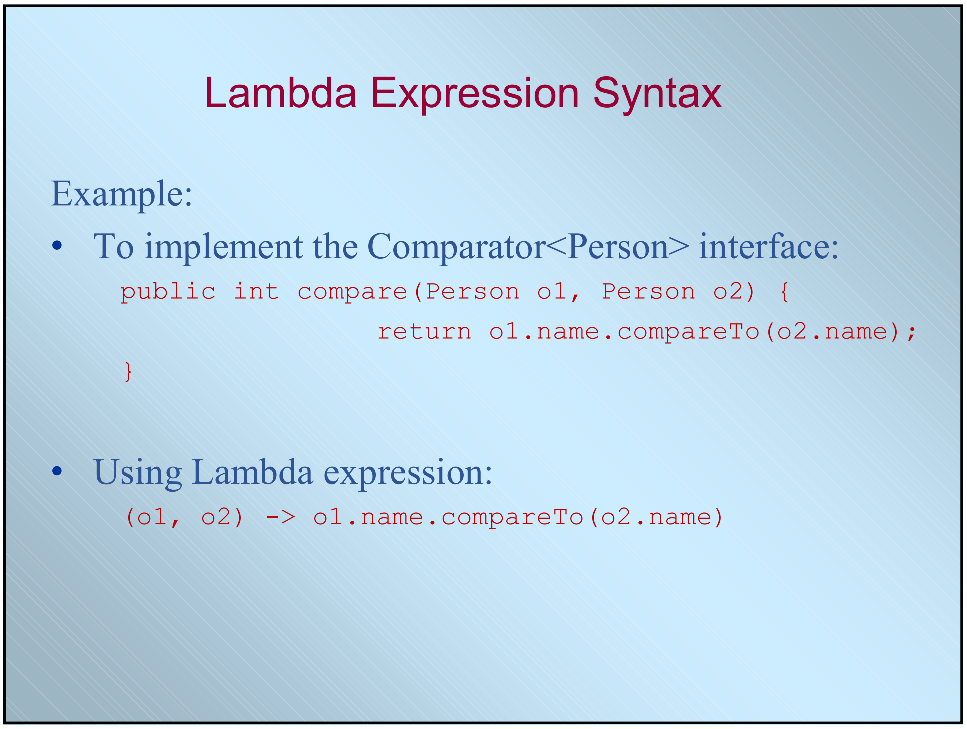
```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```



Lambda Expression Syntax

- Java lambda expression consists of three components:
  1. **Parameter-list:** It can be empty or non-empty
  2. **Arrow-symbol:** It is used to link arguments-list and body of expression.
  3. **Body:** It contains one statement or a block of statements for the lambda expression.
- Syntax:

(Parameter-list) -> {Body}



Lambda Expression Syntax

Example:

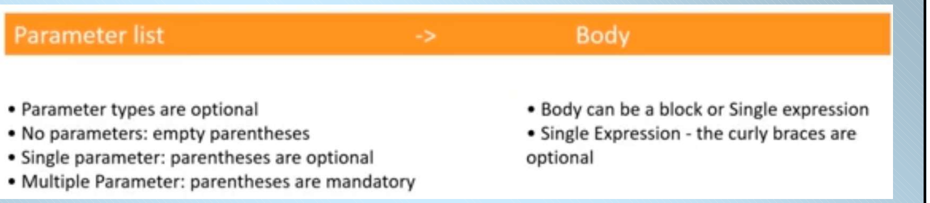
- To implement the Comparator<Person> interface:

```
public int compare(Person o1, Person o2) {
                return o1.name.compareTo(o2.name);
}
```

- Using Lambda expression:

```
(o1, o2) -> o1.name.compareTo(o2.name)
```

Lambda Expression Syntax



Parameter list -> Body

- Parameter types are optional
- No parameters: empty parentheses
- Single parameter: parentheses are optional
- Multiple Parameter: parentheses are mandatory

- Body can be a block or Single expression
- Single Expression - the curly braces are optional

Examples:

```
  () -> System.out.println(“Hello, World!)   //No parameter
   s -> System.out.println(s)                //One parameter
  (o1, o2) -> o1.name.compareTo(o2.name)     //Multiple parameter


(o1, o2) -> {return o1.name.compareTo(o2.name);} //Body with Single stmt
(o1, o2) -> o1.name.compareTo(o2.name)           //Body with Single stmt
(o1, o2) -> { System.out.println(“Hello”);   //Body with multiple stmts
              return o1.name.compareTo(o2.name);
            }
```

Without Lambda Expression

Two options:

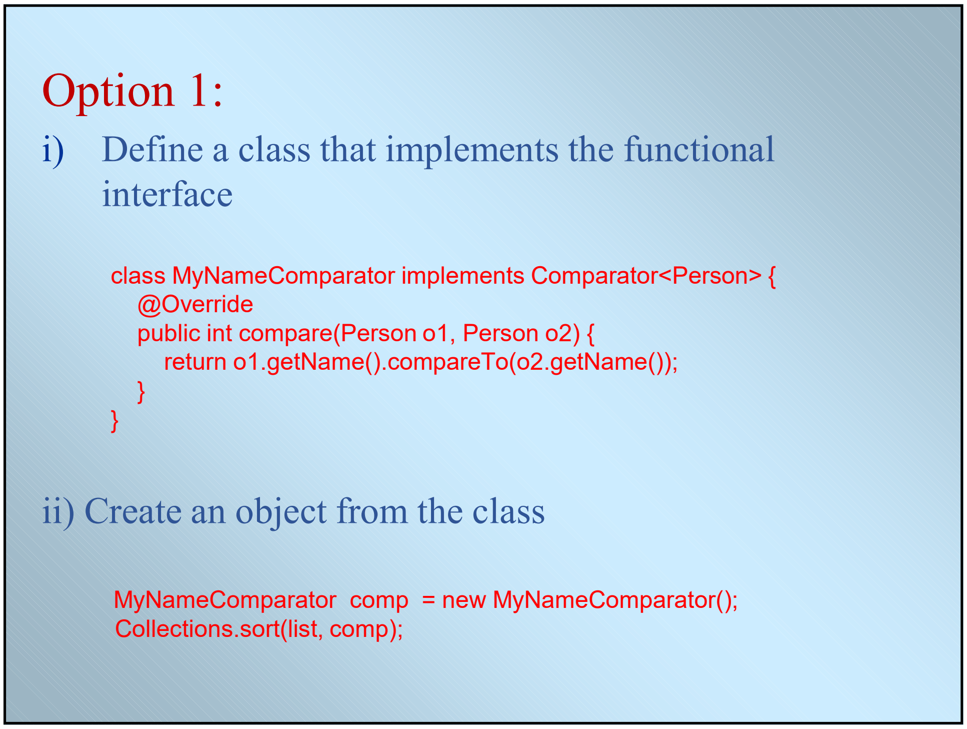
Option 1:

i) Define a class that implements the functional interface

ii) Create & use an object from the class

Option 2:

Create & use an object from an anonymous inner class that implements the functional interface



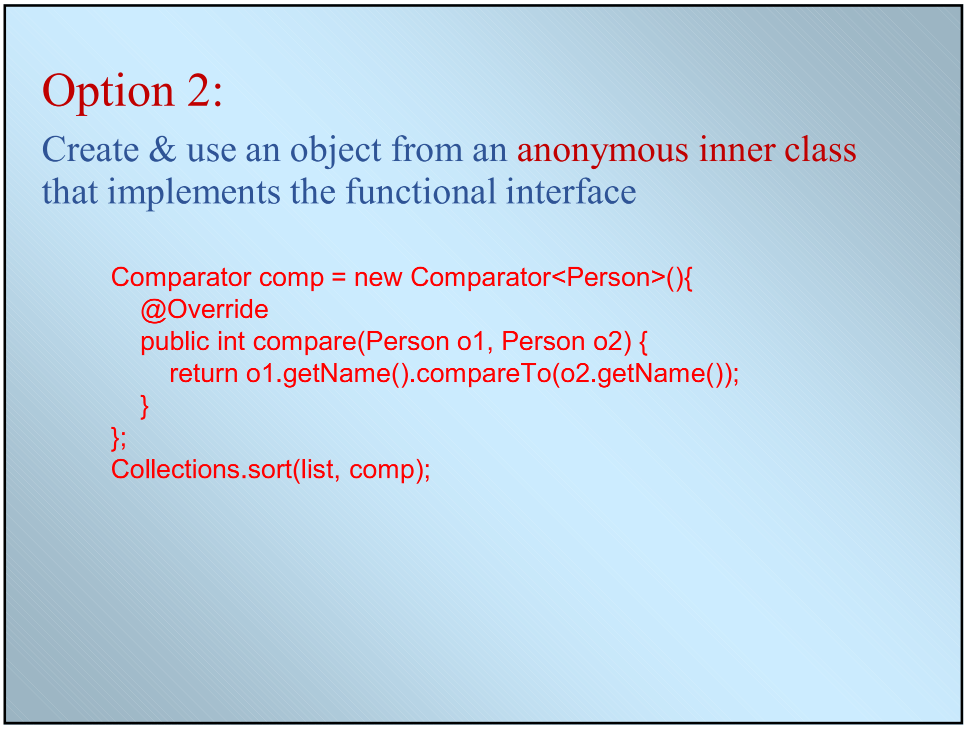
Option 1:

i) Define a class that implements the functional interface

```
class MyNameComparator implements Comparator<Person> {
   @Override
   public int compare(Person o1, Person o2) {
      return o1.getName().compareTo(o2.getName());
   }
}
```

ii) Create an object from the class

```
MyNameComparator  comp  = new MyNameComparator();
Collections.sort(list, comp);
```



Option 2:

Create & use an object from an anonymous inner class that implements the functional interface

```
Comparator comp = new Comparator<Person>(){
   @Override
   public int compare(Person o1, Person o2) {
      return o1.getName().compareTo(o2.getName());
   }
};
Collections.sort(list, comp);
```

With Lambda Expression:

```
Comparator<Person> comp = (o1, o2)  -> o1.getName().compareTo(o2.getName());

Collections.sort(list, comp);
```

Advantages of Lambda Expression:

- Provides a clear and concise way to implement a functional interface
- Allows functional programming style such as code/function can be assigned to variable, pass as method argument and function composition.

Note: Lambda can only implements functional interface while anonymous class can implement all types of interfaces.

Functional Programming Example

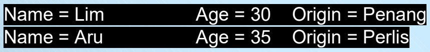
```
List<Person> people = new ArrayList<>();
people.add(new Person("Ali", 25, "Penang"));
people.add(new Person("Bob", 20, "Kedah"));
people.add(new Person("Lim", 30, "Penang"));
people.add(new Person("Aru", 35, "Perlis"));
people.add(new Person("Ken", 22, "Kedah"));

printByAge(people, 25);
```

Method:

```
public static void printByAge(List<Person> people, int age){
        for (Person p : people) {
            if (p.getAge() > age)  {
                System.out.println(p);
            }
        }
    }
```

Output:



```
Name = Lim              Age = 30    Origin = Penang
Name = Aru              Age = 35    Origin = Perlis
```

Functional Programming Example

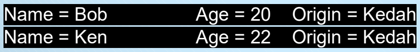
```
List<Person> people = new ArrayList<>();
people.add(new Person("Ali", 25, "Penang"));
people.add(new Person("Bob", 20, "Kedah"));
people.add(new Person("Lim", 30, "Penang"));
people.add(new Person("Aru", 35, "Perlis"));
people.add(new Person("Ken", 22, "Kedah"));

printByOrigin(people, “Kedah");
```

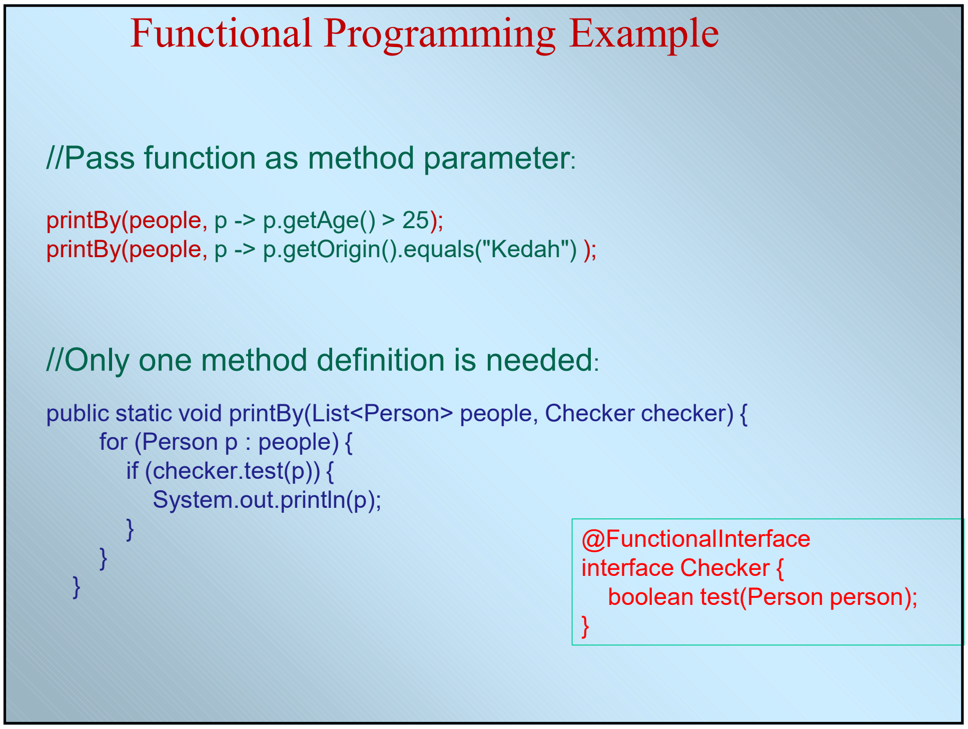
Method:

```
public static void printByOrigin(List<Person> people, String origin){
        for (Person p : people) {
            if (p.getOrigin().equals(origin))  {
                System.out.println(p);
            }
        }
    }
```

Output:



```
Name = Bob              Age = 20    Origin = Kedah
Name = Ken              Age = 22    Origin = Kedah
```



Functional Programming Example

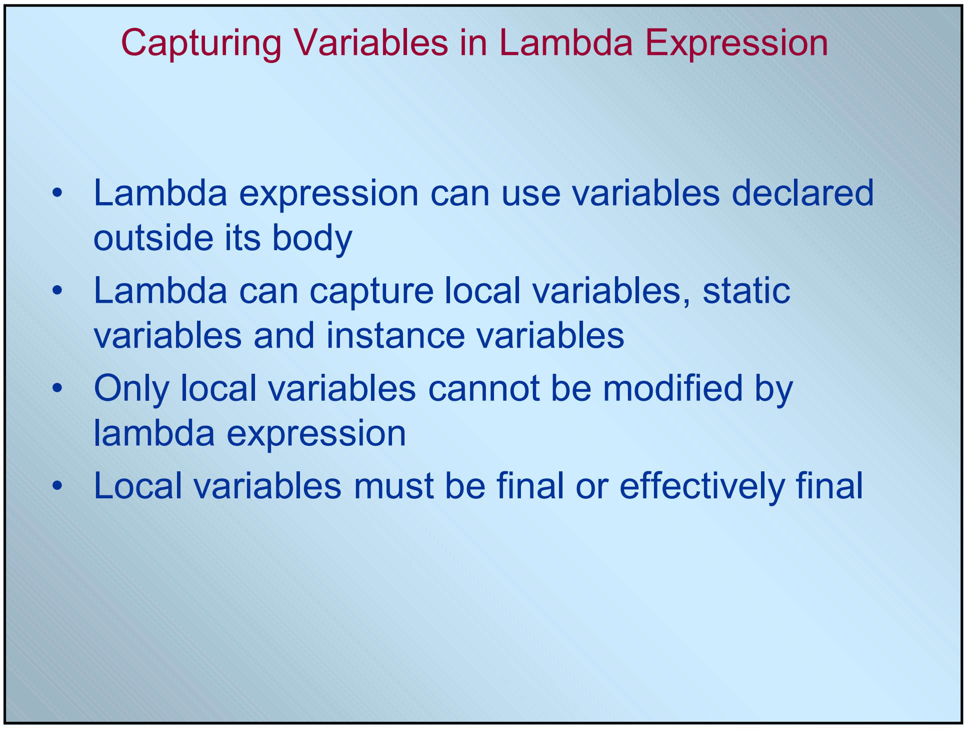
//Pass function as method parameter:

```
printBy(people, p -> p.getAge() > 25);
printBy(people, p -> p.getOrigin().equals("Kedah") );
```

//Only one method definition is needed:

```
public static void printBy(List<Person> people, Checker checker) {
        for (Person p : people) {
            if (checker.test(p)) {
                System.out.println(p);
            }
        }
    }
```

```
@FunctionalInterface
interface Checker {
    boolean test(Person person);
}
```



Capturing Variables in Lambda Expression

- Lambda expression can use variables declared outside its body
- Lambda can capture local variables, static variables and instance variables
- Only local variables cannot be modified by lambda expression
- Local variables must be final or effectively final

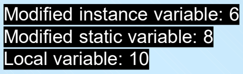
Capturing Variables in Lambda Expression

```
public class LambdaVariableAccess {
    static int instanceVariable = 5;    static int staticVariable = 7;

public void doSomething() {
        int localVar = 10;
        Runnable operation = () -> {
            instanceVariable++;
            staticVariable++;
            //localVar++;  ==>not allowed for local variable
            System.out.println("Modified instance variable: " + instanceVariable);
            System.out.println("Modified static variable: " + staticVariable);
            System.out.println("Local variable: " + localVar);
        };
        operation.run();
    }

    public static void main(String[] args) {
        LambdaVariableAccess test = new LambdaVariableAccess();
        test.doSomething();
    }
}
```

Output:



```
Modified instance variable: 6
Modified static variable: 8
Local variable: 10
```