Functional Interface as a Type

```java
static void q1() {
    Fish o1 = new Fish();
    Duck o2 = new Duck();
    o1.swim();
    o2.swim();
}

static void q2() {
    CanSwimIntf o3 = new CanSwimIntf() {
        public void swim() {
            System.out.println(x: "o3 is a fish");
        }
    };
    CanSwimIntf o4 = new CanSwimIntf() {
        public void swim() {
            System.out.println(x: "o4 is a Duck");
        }
    };
    o3.swim();
    o4.swim();
}

static void q3() {
    CanSwimIntf o5;
    o5 = () -> System.out.println(x: "o5 is a fish");
    o5.swim();
    CanSwimIntf o6 = () -> System.out.println(x: "o5 is a duck");
    o6.swim();
}
```

```java
public interface InterfaceCanSwim {
    public void swim();
}
```

Outline

- Anonymous Inner Class
- Enhanced Interface & Functional Interface
- Lambda Expression

```java
class Fish implements CanSwimIntf {
  public void swim() {
    System.out.println("flexing
          my tail back and forth");
  }
}
class Duck implements CanSwimIntf {
  public void swim() {
    System.out.println("waddling");
  }
}
```

Anonymous Inner Class

https://www3.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI_3.html

nested class

- There are 4 types of nested classes:

- non-static (instance) inner class (as a outer class member).
- static nested class (as a outer class member),
- local inner class (defined inside a method),
- anonymous local inner class (defined inside a method),

non-static (instance) inner class

- a non-static nested class belongs to an instance of the outer class, just like any instance variable or method. It can be referenced via outerClassInstanceName.innerClassInstanceName. A non-static nested class is formally called an inner class.
  - Remark : line 24 MyOuterClassWithInnerClass ‘s attribute

```
1  public class MyOuterClassWithInnerClass {
2     // Private member variable of the outer class
3     private String msgOuter = "Hello from outer class";
4
5     // Define an inner class as a member of the outer class
6     // This is merely an definition.
7     // Not instantiation takes place when an instance of outer class is constructed
8     public class MyInnerClass {
9        // Private variable of the inner class
10       private String msgInner;
11       // Constructor of the inner class
12       public MyInnerClass(String msgInner) {
13          this.msgInner = msgInner;
14          System.out.println("Constructing an inner class instance: " + msgOuter);
15                // can access private member variable of outer class
16       }
17       // A method of inner class
18       public void printMessage() {
19          System.out.println(msgInner);
20       }
21    }
22
23    // Declare and construct an instance of the inner class, inside the outer class
24    MyInnerClass anInner = new MyInnerClass("Hi from inner class");
25 }
```

Two class files are produced: `MyOuterClassWithInnerClass.class` and `MyOuterClassWithInnerClass$MyInnerClass.class`.

non-static (instance) inner class

```java
public class MyOuterClassWithInnerClass {
   // Private member variable of the outer class
   private String msgOuter = "Hello from outer class";

   // Define an inner class as a member of the outer class
   // This is merely an definition.
   // Not instantiation takes place when an instance of outer class is constructed
   public class MyInnerClass {
      // Private variable of the inner class
      private String msgInner;
      // Constructor of the inner class
      public MyInnerClass(String msgInner) {
         this.msgInner = msgInner;
         System.out.println("Constructing an inner class instance: " + msgOuter);
               // can access private member variable of outer class
      }
      // A method of inner class
      public void printMessage() {
         System.out.println(msgInner);
      }
   }

   // Declare and construct an instance of the inner class, inside the outer class
   MyInnerClass anInner = new MyInnerClass("Hi from inner class");
}
```

Two class files are produced: `MyOuterClassWithInnerClass.class` and `MyOuterClassWithInnerClass$MyInnerClass.class`.

```java
public static void main(String[] args) {
    // Construct an instance of outer class, which create anInner
    MyOuterClassWithInnerClass anOuter = new MyOuterClassWithInnerClass();
    System.out.println(x: "Invoke inner class's method from this outer class instance");
    anOuter.anInner.printMessage();

    System.out.println(x: "Explicitly construct another instance of inner class");
    MyOuterClassWithInnerClass.MyInnerClass inner2 = anOuter.new MyInnerClass(msgInner: "Inner class 2");
    inner2.printMessage();

    System.out.println(x: "Explicitly construct an instance of inner class");
    MyOuterClassWithInnerClass.MyInnerClass inner3 = new MyOuterClassWithInnerClass().new MyInnerClass(
            msgInner: "Inner class 3");
    inner3.printMessage();
}
```

```
Constructing an inner class instance: Hello from outer class
Invoke inner class's method from this outer class instance
Hi from inner class
Explicitly construct another instance of inner class
Constructing an inner class instance: Hello from outer class
Inner class 2
Explicitly construct an instance of inner class
Constructing an inner class instance: Hello from outer class
Constructing an inner class instance: Hello from outer class
Inner class 3
```

Local Inner Class Defined Inside a Method

```java
public class MyOuterClassWithLocalInnerClass {
   // Private member variable of the outer class
   private String msgOuter = "Hello from outer class";

   // A member method of the outer class
   public void doSomething() {

      // A local variable of the method
      final String msgMethod = "Hello from method";

      // Define a local inner class inside the method
      class MyInnerClass {
         // Private variable of the inner class
         private String msgInner;
         // Constructor of the inner class
         public MyInnerClass(String msgInner) {
            this.msgInner = msgInner;
            System.out.println("Constructing an inner class instance: " + msgOuter);
               // can access private member variable of outer class
            System.out.println("Accessing final variable of the method: " + msgMethod);
               // can access final variable of the method
         }
         // A method of inner class
         public void printMessage() {
            System.out.println(msgInner);
         }
      }

      // Create an instance of inner class and invoke its method
      MyInnerClass anInner = new MyInnerClass("Hi, from inner class");
      anInner.printMessage();
   }

   // Test main() method
   public static void main(String[] args) {
       // Create an instance of the outer class and invoke the method.
       new MyOuterClassWithLocalInnerClass().doSomething();
   }
}
```

- Java allows you to define an inner class inside a method, just like defining a method's local variable. Like local variable, a local inner class does not exist until the method is invoked, and goes out of scope when the method exits.
- A local inner class has these properties:
  - A local inner class cannot have access modifier (such as private or public). It also cannot be declared static.
  - A local inner class can access all the variables/methods of the enclosing outer class.
  - A local inner class can have access to the local variables of the enclosing method only if they are declared final (to prevent undesirable side-effects).

An Anonymous Inner Class

- An anonymous inner class is a local inner class (of a method) without assigning an explicit classname. It must either "extends" an existing superclass or "implements" an interface. It is declared and instantiated in one statement via the new keyword.
  - notice line 21 is a one-time used object.

```
3  public class Anony {
4      private String msgOuter = "Hello from outer class";
5
6      // A member method of the outer class
7      public void doSomething() {
8
9          // A local variable of the method
10         final String msgMethod = "Hello from method";
11         A obj = new A() {
12             void aMethod() {
13                 System.out.println("msgMethod = " + msgMethod);
14             }
15         };
16         obj.aMethod();
17     }
18
       Run | Debug
19     public static void main(String[] args) {
20         // Create an instance of the outer class and invoke the method.
21         new Anony().doSomething();
22     }
23 }
24
25 abstract class A {
26     abstract void aMethod();
27 }
```

Enhanced Interface, Lambda Expressions, Streams and Functional Programming (JDK 8, 9, 10, 11)

https://www3.ntu.edu.sg/home/ehchua/programming/java/JDK8_Lambda.html

introduction

- JDK 8 is a MAJOR upgrade, which introduces many new language features to support Functional Programming:
  - Re-design the interface to support public default and public static methods. JDK 9 further supports private and private static methods.
  - Introduce lambda expressions as a shorthand for creating an instance of an anonymous inner class implementing a single-abstract-method interface.
  - Retrofit the Collection framework, by adding new default and static methods to the existing interfaces.
  - Introduce the Stream API to efficiently handle filter-map-reduce operations in functional programming.

JDK 8/9's interface

(Recap) There are three kinds of methods in JDK 8 interfaces: abstract (instance), default (instance) and static (class). All methods are public.

1. (Pre-JDK 8) public static (class) final fields or constants.
2. (Pre-JDK 8) public abstract (instance) methods WITHOUT implementation - MUST be overridden by the implementation subclasses.
3. (JDK 8) public default (instance) method with implementation - inherited by the implementation subclasses; MAY be overridden but NOT necessarily.
4. (JDK 8) public static (class) method with implementation - NOT inherited by its sub-types (unlike superclass' static methods).
5. (JDK 9) private (instance) method with implementation - NOT inherited by its sub-types; CANNOT be invoked by other static (class) methods within the interface.
6. (JDK 9) private static (class) method with implementation - NOT inherited by its sub-types; CAN be invoked by other static (class) methods within the interface.

(recap) interface

- Prior to JDK 8, a Java interface is a pure abstract superclass, containing public abstract (instance) methods without implementation.
- JDK 8 introduces public default (instance) and public static (class) methods.
- JDK 9 introduces private (instance) and private static (class) methods.

- JDK 8/9 blurs the distinction between interface and abstract superclass.

- Variables: Interface can contain only class variables (public static final). Abstract superclass can contain instance variables, but interface cannot.
- Method Access Control: All methods (abstract, static and default) in interface are public. JDK 9 supports private (instance) and private static (class) methods.
- A Java class can extend one superclass, but can implement multiple interfaces.

interface public default (instance) Methods (JDK 8)

MyImplClass1

```
public interface MyJ8InterfaceWithDefault {
   void foo();   // abstract public (instance) (pre-JDK 8)

   // Default methods are marked by keyword "default"
   default void bar() {    // public (instance) (JDK 8)
      System.out.println("MyJ8InterfaceWithDefault runs default bar()");
   }

   //default void bar1();
   //compilation error: missing method body, or declare abstract
}
```

```
public class MyImplClass1 implements MyJ8InterfaceWithDefault {
   // Need to override ALL the abstract methods,
   //   but not necessarily for the default methods.
   @Override
   public void foo() {
      System.out.println("MyImplClass1 runs foo()");
   }

   // Test Driver
   public static void main(String[] args) {
      MyImplClass1 c = new MyImplClass1();
      c.foo();  //MyImplClass1 runs foo()
      c.bar();  //MyJ8InterfaceWithDefault runs default bar()
   }
}
```

interface public default (instance) Methods (JDK 8)

MyImplClass2

- JDK 8 requires the implementation classes to override the default methods if more than one versions are inherited.

```
public interface MyJ8InterfaceWithDefault {
   void foo();   // abstract public (instance) (pre-JDK 8)

   // Default methods are marked by keyword "default"
   default void bar() {    // public (instance) (JDK 8)
      System.out.println("MyJ8InterfaceWithDefault runs default bar()");
   }

   //default void bar1();
   //compilation error: missing method body, or declare abstract
}
```

```
public interface MyJ8InterfaceWithDefault1 {
   // This default method has same signature but different implementation as MyJ8InterfaceWithDefault
   default void bar() {   // public (instance) (JDK 8)
      System.out.println("MyJ8InterfaceWithDefault1 runs default bar() too!");
   }
}
```

```
public class MyImplClass2 implements MyJ8InterfaceWithDefault, MyJ8InterfaceWithDefault1 {
   // Need to override ALL abstract methods
   @Override
   public void foo() {
      System.out.println("MyImplClass2 runs foo()");
   }

   @Override
   public void bar() {
      System.out.println("MyImplClass2 runs overridden bar()");
   }
   // bar() exists in both interfaces.
   // You MUST override, otherwise
   //compilation error: class MyImplClass2 inherits unrelated defaults for bar()
   //        from types MyJ8InterfaceWithDefault and MyJ8InterfaceWithDefault1

   public static void main(String[] args) {
      MyImplClass2 c = new MyImplClass2();
      c.foo();   //MyImplClass2 runs foo()
      c.bar();   //MyImplClass2 runs overridden bar()
   }
}
```

interface public static (class) Methods (JDK 8)

MyImplClass3

```
public interface MyJ8InterfaceWithStatic {
   void foo();   // abstract public (instance) (pre-JDK 8)

   static void bar() {  // public (class) (JDK 8)
      System.out.println("MyJ8InterfaceWithStatic runs static bar()");
   }

   //static void bar1();
   //compilation error: missing method body, or declare abstract
}
```

```
public class MyImplClass3 implements MyJ8InterfaceWithStatic {
   // Need to override ALL abstract method
   @Override
   public void foo() {
      System.out.println("MyImplClass3 run foo()");
   }

   // Test Driver
   public static void main(String[] args) {
      MyImplClass3 c = new MyImplClass3();
      c.foo();  //MyImplClass3 run foo()
      // Invoke static (class) method via ClassName.staticMethodName()
      MyJ8InterfaceWithStatic.bar();  //MyJ8InterfaceWithStatic runs static bar()

      // Interface's static methods are NOT inherited (Unlike Superclass)!!!
      //MyImplClass3.bar();
            //compilation error: cannot find symbol bar()
      //c.bar();
            //compilation error: cannot find symbol bar()
      //MyJ8InterfaceWithStatic c1 = new MyImplClass3();
      //c1.bar();
            //compilation error: illegal static interface method call
   }
}
```

functional interface

- An interface containing only ONE abstract method is called a single-abstract-method interface or functional interface.
- JDK has many functional interfaces. The most commonly-used are:
  - Interface java.awt.event.ActionListener with single abstract method actionPerformed(): used as ActionEvent handler.
  - Interface java.lang.Runnable with single abstract method run(): for starting a new thread.
  - Interface java.util.Comparator with single abstract method compare(): used in Collections.sort() or Arrays.sort().
  - These interfaces are commonly implemented in an anonymous inner class.
- The @FunctionalInterface annotation can be used to mark and inform the compiler that an interface contains only one abstract method.
  - This is useful to prevent accidental addition of extra abstract methods into a functional interface.

Lambda expression with functional interface

```java
// A Single-Abstract-Method Interface called Functional Interface
@FunctionalInterface   // ask compiler to check this interface contains only one abstract method
public interface HelloFunctionalInterface {
   void sayHello(String name);  // public abstract
}
```

- JDK 8's Functional Interface and Lambda Expression allow us to construct a "Function Object" in a one-liner (or a fewer lines of codes).
  - Java is an Object-oriented Programming language. Everything in Java are objects (except primitives). Functions are not objects in Java (but part of an object), and hence, they cannot exist by themselves.

```java
public class HelloFunctionalInterfaceInnerClassTest {
   public static void main(String[] args) {
      // Define an anonymous inner class implementing the interface.
      // Construct an instance and invoke the method.
      HelloFunctionalInterface h = new HelloFunctionalInterface() {
         public void sayHello(String name) {
            System.out.println("Hello, " + name);
         }
      };
      h.sayHello("Paul");
   }
}
```

```java
public class HelloFunctionalInterfaceLambdaTest {
   public static void main(String[] args) {
      // Define an anonymous inner class implementing the interface.
      // Construct an instance and invoke the method.
      HelloFunctionalInterface h = name -> System.out.println("Hello, " + name);
            // Using lambda expression as a shorthand for the above
      h.sayHello("Peter");
   }
}
```

Lambda Expression Syntax

- Lambda Expression defines the "sole" method of a Functional Interface. It consists of 2 parts: parameters and method body, separated by ->.
- The parameters are separated by commas and enclosed by parentheses.
  - The parentheses can be omitted if there is only one parameter.
  - The parameters' type and the return type are also optional, as they can be inferred from the method signature.
- The method body could be a statement or a block.
  - The method name is omitted, as it can be inferred from the sole abstract method of the Functional Interface.

The syntax is:

```
(arguments) -> { body }
```

For examples:

```
() -> statement    // No argument and one-statement method body

arg -> statement   // One argument (parentheses can be omitted) and method body

(arg1, arg2, ...) -> {
   body-block
}   // Arguments separated by commas and the block body

(Type1 arg1, Type2 arg2, ...) -> {
   method-body-block;
   return return-value;
}   // With arguments and block body
```

Lambda Expression Java 8

https://javatechonline.com/lambda-expression-java-8/

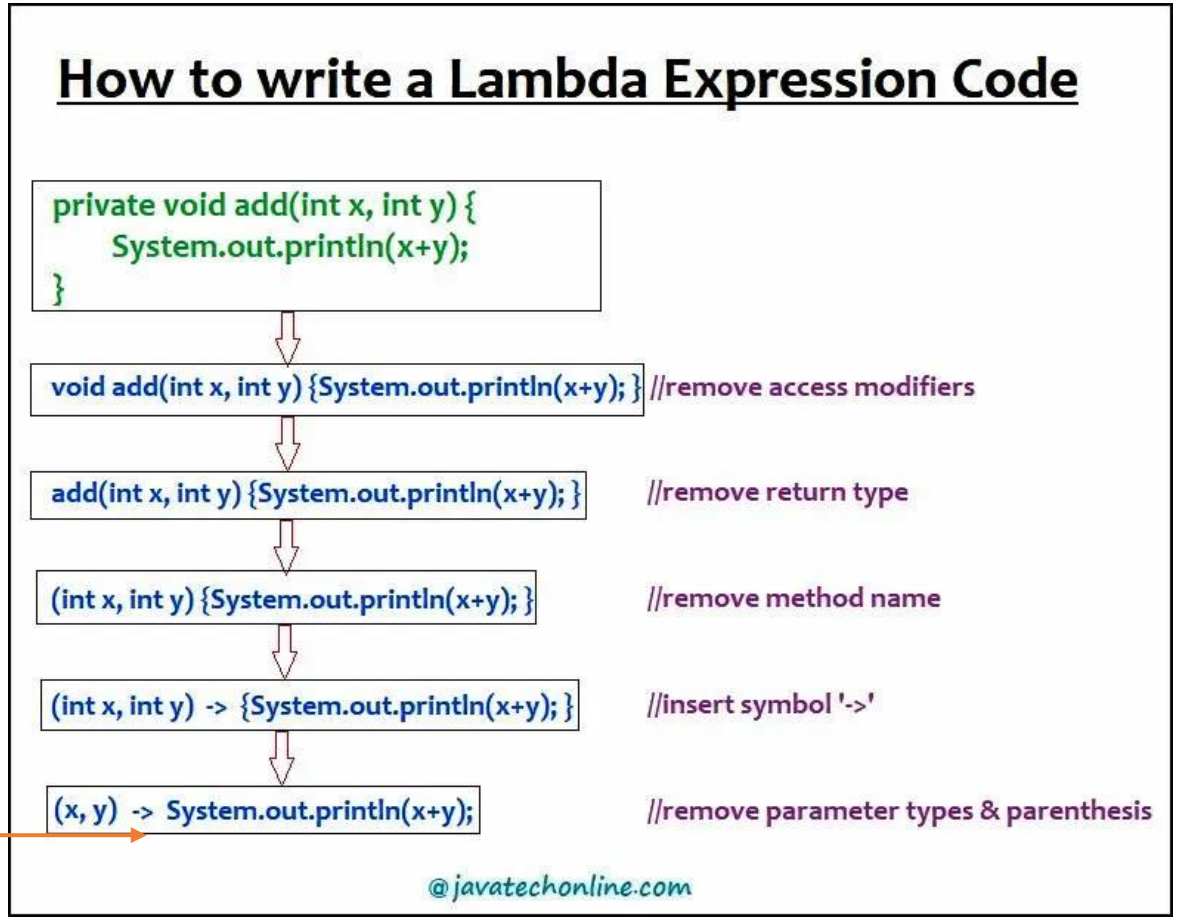
by devs5003 - April 19, 2021

What is a Lambda Expression

- Lambda (λ) Expression is an anonymous (nameless) function. In other words, the function which doesn't have the name, return type and access modifiers. Lambda Expression is also known as anonymous functions or closures.
- We use lambda expressions when we provide implementation of Functional Interfaces.
- It provides a clear and concise way to represent one method interface using an expression only. Lambda expressions can also be used in a method as an argument.
- Also, it is very useful in collection & Streams API.

Lambda Expression vs a Method in Java

- Method :
  A method has various parts such as :
  - 1) Method name 2) Parameter list
    3) Return type 4) Access/Non-access Modifiers 5) method body
    6) exception handlers with throws keyword and also try, catch in a normal scenario.
- Lambda Expression :
  A Lambda expression has only two parts :
  - 1) Parameter List & 2) Body
  - -> (arrow key!!!)



Note on Lambda Expressions

1. If only one method parameter is available and compiler can understand the type based on the context, then we can remove the type & parenthesis both.
   - Suppose (String s) -> {System.out.println(s)};
     can be simplified to s -> {System.out.println(s)};
2. If one statement present then curly braces are optional.
   - As in example at point # 1 can again be simplified as s -> System.out.println(s);
3. If no parameters are available, then we can use empty parenthesis like :
   - ( ) -> System.out.println("No parameter test");
4. Equally important, if you are learning Lambda expressions for the first time, Keep below syntax in your mind :
   - Interface iObj = (params) -> { method body };
   - we can use Lambda Expressions in place of the Anonymous inner class – if the class implements Functional Interface in that particular case.
5. variables declared inside Lambda expressions will be treated as a local variable.

Keypoints

- Benefits Of Using Lambda Expressions
  - We can write more readable, maintainable & concise code using Lambda expressions.
  - Also, we can incorporate functional programming capabilities in java language with Lambda Expressions.
  - We can use Lambda expressions in place of Inner classes to reduce the complexity of code accordingly.
  - Even we can pass a lambda expression as an argument to a method.
  - Additionally, we can provide lambda expression in place of an object.

Lambda expression example

- No Parameter
  - Notice ()

```
interface Sayable{
    public String say();
}
public class LambdaExpressionExample3{
public static void main(String[] args) {
    Sayable s=()->{
        return "I have nothing to say.";
    };
    System.out.println(s.say());
}
}
```

```
I have nothing to say.
```

https://www.javatpoint.com/java-lambda-expressions

Lambda expression example

- Single Parameter

```
Hello, Sonoo
Hello, Sonoo
```

```java
public class LambdaExpressionExample4{
    public static void main(String[] args) {

        // Lambda expression with single parameter.
        Sayable s1=(name)->{
            return "Hello, "+name;
        };
        System.out.println(s1.say("Sonoo"));

        // You can omit function parentheses
        Sayable s2= name ->{
            return "Hello, "+name;
        };
        System.out.println(s2.say("Sonoo"));
    }
}
```

https://www.javatpoint.com/java-lambda-expressions

Lambda expression example

- Multiple Parameters

```
30
300
```

```java
interface Addable{
    int add(int a,int b);
}

public class LambdaExpressionExample5{
    public static void main(String[] args) {

        // Multiple parameters in lambda expression
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression
        Addable ad2=(int a,int b)->(a+b);
        System.out.println(ad2.add(100,200));
    }
}
```

https://www.javatpoint.com/java-lambda-expressions

Lambda expression example

- Multiple Statements
  - In Java lambda expression, if there is only one statement, you may or may not use return keyword. You must use return keyword when lambda expression contains multiple statements.

https://www.javatpoint.com/java-lambda-expressions

```
@FunctionalInterface
interface Sayable{
    String say(String message);
}

public class LambdaExpressionExample8{
    public static void main(String[] args) {

        // You can pass multiple statements in lambda expression
        Sayable person = (message)-> {
            String str1 = "I would like to say, ";
            String str2 = str1 + message;
            return str2;
        };
            System.out.println(person.say("time is precious."));
    }
}
```

Comparator Interface

```
public class MovieByYear implements
Comparator<CSMovie> {
  @Override
  public int compare(CSMovie o1, CSMovie o2) {
        // TODO Auto-generated method stub
        return o1.getYear() - o2.getYear();
  }
}
```

```
public class CSMovie {
  private String title;
  private int year;
  private int revenue;

  public CSMovie(String t,
                 int yr, int rev) {
        title = t;
        year = yr;
        revenue = rev;
    }

  @Override
  public String toString() {
      return "CSMovie [title=" + title
            + ", year=" + year
            + ", revenue="
            + revenue + "]";
  }
  …
}
```

Comparator Interface

```
public class MovieByYear implements
Comparator<CSMovie> {
  @Override
  public int compare(CSMovie o1, CSMovie o2) {
        // TODO Auto-generated method stub
        return o1.getYear() - o2.getYear();
  }
}
```

q1
CSMovie [title=The Imitation Game, year=2014, revenue=320]
CSMovie [title=Matrix, year=1999, revenue=208]
CSMovie [title=Transcendence, year=2014, revenue=150]
CSMovie [title=PK, year=2014, revenue=240]
q2
CSMovie [title=Matrix, year=1999, revenue=208]
CSMovie [title=The Imitation Game, year=2014, revenue=320]
CSMovie [title=Transcendence, year=2014, revenue=150]
CSMovie [title=PK, year=2014, revenue=240]

```
static void q1() { // initialized list
    aList = new ArrayList<>();
    aList.add(new CSMovie("The Imitation Game",
                          2014, 320));
    aList.add(new CSMovie("Matrix",
                          1999, 208));
    aList.add(new CSMovie("Transcendence",
                          2014, 150));
    aList.add(new CSMovie("PK",
                          2014, 240));
    System.out.println("q1");
    printMovie();
    // Comparator c = new Comparator();
    // will never compile
}
static void q2() { // sort by year
        Collections.sort(aList,
                         new MovieByYear());
        System.out.println("q2");
        printMovie();
}
```

Comparator Interface

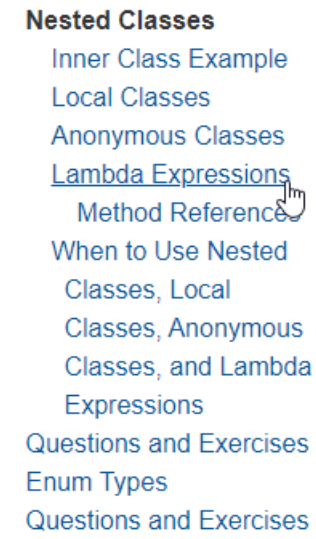
```
public class MovieByYear implements
Comparator<CSMovie> {
  @Override
  public int compare(CSMovie o1, CSMovie o2) {
        // TODO Auto-generated method stub
        return o1.getYear() - o2.getYear();
  }
} // Collections.sort(aList, new MovieByYear());
```

```
static void q3() { // sort by revenue
    Collections.sort(aList,
          new Comparator<CSMovie>() {
    public int compare(CSMovie m1, CSMovie m2) {
              int rev1 = m1.getRevenue();
               int rev2 = m2.getRevenue();
       return Integer.compare(rev1, rev2);
    }
         });
    System.out.println("q3");
    printMovie();
  }
static void q4() { //by year then revenue
   Collections.sort(aList, (m1, m2) -> {
     int byYear =
     Integer.compare(m1.getYear(), m2.getYear());
     if (byYear != 0)
         return byYear;
     int rev1 = m1.getRevenue();
     int rev2 = m2.getRevenue();
     return Integer.compare(rev1, rev2);
     });
   System.out.println("q4");
   printMovie();
}
```

q3
CSMovie [title=Transcendence, year=2014, revenue=150]
CSMovie [title=Matrix, year=1999, revenue=208]
CSMovie [title=PK, year=2014, revenue=240]
CSMovie [title=The Imitation Game, year=2014, revenue=320]
q4
CSMovie [title=Matrix, year=1999, revenue=208]
CSMovie [title=Transcendence, year=2014, revenue=150]
CSMovie [title=PK, year=2014, revenue=240]
CSMovie [title=The Imitation Game, year=2014, revenue=320]

Further Readings

- https://www.baeldung.com/java-8-lambda-expressions-tips
- https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html



.equals()
and .hashCode()

java.lang

Class Object

java.lang.Object

public class **Object**

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the me

Since:

JDK1.0

See Also:

`Class`

Constructor Summary

Constructors

Constructor and Description
`Object()`

Method Summary

Methods

Modifier and Type	Method and Description
`protected Object`	`clone()` Creates and returns a copy of this object.
`boolean`	`equals(Object obj)` Indicates whether some other object is "equal to" this one.
`protected void`	`finalize()` Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`Class<?>`	`getClass()` Returns the runtime class of this `Object`.
`int`	`hashCode()` Returns a hash code value for the object.
`void`	`notify()` Wakes up a single thread that is waiting on this object's monitor.
`void`	`notifyAll()` Wakes up all threads that are waiting on this object's monitor.
`String`	`toString()` Returns a string representation of the object.
`void`	`wait()` Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.
`void`	`wait(long timeout)` Causes the current thread to wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed.
`void`	`wait(long timeout, int nanos)` Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

.equals() and .hashCode()

It is also possible that an object is equal to another given object, then the equals() method follow the **equivalence relation** to compare the objects.

- **Reflexive:** If **x** is a non-null reference, the calling of **x.equals(x)** must return true.
- **Symmetric:** If the two non-null references are **x** and **y, x.equals(y)** will return **true** if and only if **y.equals(x)** return **true**.
- **Transitive:** If the three non-null references are **x, y**, and **z, x.equals(z)** will also return **true** if **x.equals(y)** and **y.equals(z)** both returns **true**.
- **Consistent:** If the two non-null references are **x** and **y**, the multiple calling of **x.equals(y)** constantly returns either true or false. It does not provide any information used in the comparison.
- For any non-null reference **x, x.equals(null)** returns false.

https://www.javatpoint.com/how-to-compare-two-objects-in-java

Object class defined equals() method like this:

```
public boolean equals(Object obj) {
        return (this == obj);
}
```

```
@Override
public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + id;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
}

@Override
public boolean equals(Object obj) {
        if (this == obj)
                return true;
        if (obj == null)
                return false;
        if (getClass() != obj.getClass())
                return false;
        DataKey other = (DataKey) obj;
        if (id != other.id)
                return false;
        if (name == null) {
                if (other.name != null)
                        return false;
        } else if (!name.equals(other.name))
                return false;
        return true;
}
```

https://www.digitalocean.com/community/tutorials/java-equals-hashcode

.equals() and .hashCode()

HashCodeExample.java

```
public class HashcodeExample
{
public static void main(String[] args)
{
//creating two instances of the Employee class
Employee emp1 = new Employee(918, "Maria");
Employee emp2 = new Employee(918, "Maria");
//invoking hashCode() method
int a=emp1.hashCode();
int b=emp2.hashCode();
System.out.println("hashcode of emp1 = " + a);
System.out.println("hashcode of emp2 = " + b);
System.out.println("Comparing objects emp1 and emp2 = " + emp1.equals(emp2));
}
}
```

Output:

```
hashcode of emp1 = 2398801145
hashcode of emp2 = 1852349007
Comparing objects emp1 and emp2 = false
```

```
//overriding equals() method
@Override
public boolean equals(Object obj)
{
if (obj == null)
return false;
if (obj == this)
return true;
return this.getRegno() == ((Employee) obj). getRegno();
}
```

Output:

```
hashcode of emp1 = 2032578917
hashcode of emp2 = 1531485190
Comparing objects emp1 and emp2 = true
```

https://www.javatpoint.com/how-to-compare-two-objects-in-java