Advanced Java

Lambda Expressions

Java local-variable type inference

- Added in Java 10
- No need to specify type of local variables in some cases
- Type of variable is specified using *var* keyword
- Compiler automatically infers the type of local variables
- Restricted to
    - Local variables, including
        - index variables of for-each loops
        - resource variables of the try-with-resources statement

Examples

- var list = new ArrayList<Integer>(); // infers ArrayList<Integer>
- for(var v : list) { …}   // infers Integer
- var myNum = new Integer(123);   // infers Integer
- var myClassObj = new MyClass(); // infers MyClass

Additional Restriction

- Is this possible?

```
var v;

v = new Object();
```

- Java is still a statically typed language
- When using *var*
  - There should be enough information to infer the type of local variable.
  - If not, the compiler will throw an error.
  - The type of the variable is inferred from the type of the initializer.

- What happens with “var v = null;”

```
error: cannot infer type for local variable v
 var v = null;
 ^ (variable initializer is 'null')
```

Lambda Expressions - Introduction

- Prior to Java SE 8, Java supported three programming paradigms:
  - Procedural programming
  - Object-oriented programming
  - Generic programming
- Java SE 8 adds Functional programming.

Introduction - 2

- Without functional programming
  - First, you typically determines what you want to accomplish
  - Then, specify the precise steps to accomplish that task.
  - Usually applies external iteration
    - Using a loop to iterate over a collection of elements.
    - Requires accessing the elements sequentially
- With functional programming
  - Specify what you want to accomplish in a task (a.k.a. as function), but not how to accomplish it
  - Internal iteration
    - Let the data structure determine how to iterate over a collection of elements
    - Internal iteration is easier to parallelize

Motivation example

- Limitations:
  - Besides adding, also needs to specify how the iteration will take place (external iteration)
  - There is a lot of code to do even a simple task
  - The program is sequential in nature. To parallel it is not trivial

```
private static int sumIterator(List<Integer> list) {
 Iterator<Integer> it = list.iterator();
 int sum = 0;
 while (it.hasNext()) {
  int num = it.next();
  if (num > 10) {
   sum += num;
  }
 }

 return sum;
}
```

Motivation Example

- Consider the following entity
- And users of the application are stored in a List<Person> attribute
- Want to be able to filter the members and only print those that satisfy a given criteria

```
public enum Sex {
    MALE, FEMALE
}

public class Person {
    private String name;
    private LocalDate birthday;
    private Sex gender;
    private String emailAddress;

    public int getAge() {
        // ...
    }

    public void print() {
        // ...
    }
}
```

Filter Members - 1

- Create a filtering method for criteria

```
public static void printPersonsOlderThan(List<Person> members, int age) {
    for (Person p : members) {
        if (p.getAge() >= age) {
            p.print();
        }
    }
}
```

- Additional criteria -> implement another *printPersonWithCriteria* method
- How to improve?
  - Separate the iteration code on Person from the filtering criteria code

Specify Filtering Criteria Code

Define abstraction

```
public interface CheckPerson {
  boolean test(Person p);
}
```

Define generic print

```
public static void printPersons (
    List<Person> members, CheckPerson tester) {
    for (Person p : members)
        if (tester.test(p))
            p.print();
}
```

Advantages:

- Refactor repeated code
- Search criteria can be reused

Specify Filtering Criteria Code with a Class

Define abstraction

```
public interface CheckPerson {
  boolean test(Person p);
}
```

Consider only young male adults:

```
class CheckMaleAdults implements CheckPerson {
    public boolean test(Person p) {
        return p.gender == Sex.MALE &&
            p.getAge() >= 18 && p.getAge() <= 25;
    }
}
```

Define generic print

```
public static void printPersons (
    List<Person> members, CheckPerson tester) {
    for (Person p : members)
        if (tester.test(p))
            p.print();
}
```

```
printPersons(members, new CheckMaleAdults());
```

Disadvantages:

- Additional interface
- Define class for each criteria

Specify Filtering Criteria Code with an Anonymous Class

Define abstraction

```
public interface CheckPerson {
  boolean test(Person p);
}
```

Define generic print

```
public static void printPersons (
    List<Person> members, CheckPerson tester) {
    for (Person p : members)
        if (tester.test(p))
            p.print();
}
```

```
printPersons(members,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Sex.MALE
                && p.getAge() >= 18 && p.getAge() <= 25;
        }
    });
```

- Reduces amount of code
- But syntax of anonymous class

Syntax of Java 8 Lambdas

- Java 8 SE supports functions as first-class citizens
  - Lambda expression
- A lambda is basically a method in Java without a declaration usually written as

  (parameterList) -> body
- A lambda can have zero or more parameters separated by commas and their type can be explicitly declared or inferred from the context
  - (int x, int y) -> { return x + y; }
  - (x, y) -> { return x + y; }
- ( ) is used to denote zero parameters
  - () -> { System.out.println("Hello World!"); }
- Parenthesis are not needed around a single parameter
  - x -> { return x * x; }

Syntax of Java 8 Lambdas – 2

- The body consists of a single expression or a statement block
- If you specify a single expression, then the Java runtime evaluates the expression and then returns its value (if needed)
- Braces are not needed around a single-statement body
  - (x) -> System.out.println(x);
  - (x) -> x + x;
- However, return statement is not an expression
  - In a lambda expression, you must enclose a return statement always in braces ({})
  - (x) -> { return x + x; }
- You can consider lambda expressions as anonymous methods

Implementation of Java 8 Lambdas

- The Java 8 compiler first converts a lambda expression into a function
- It then calls the generated function
- For example, `x -> System.out.println(x)` could be converted into a generated static function

```
public static void genName(Integer x) {
    System.out.println(x);
}
```

- But what type should be generated for this function?
- How should it be called?
- What class should it go in?

Solution: Functional Interfaces

- Design decision: Java 8 lambdas are assigned to functional interfaces
- A functional interface is a Java interface with ***exactly one* abstract method** that does not override a method in java.lang.Object

```
@FunctionalInterface
public interface Consumer<T> {
   void accept(T t);
}
```

- The package `java.util.function` defines many new useful functional interfaces.

Four Categories of Functional Interfaces

Supplier

```
interface Supplier<T> {
  T get();
}
```

Predicate

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

```
interface Predicate<T, U> {
  boolean test(T t, U u);
}
```

Consumer

```
interface Consumer<T> {
  void accept(T t);
}
```

```
interface Consumer<T, U> {
  void accept(T t, U u);
}
```

Function

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

```
interface BiFunction<T, U, R> {
  R apply(T t, U u);
}
```

Properties of the Generated Method

```
public interface Consumer<T> {
  void accept(T t);
}
```

```
x -> System.out.println(x);
```

- The method generated from a Java 8 lambda expression has the same signature as the method in the functional interface
  - void accept(T t)
- The type is the same as that of the functional interface to which the lambda expression is assigned
  - Consumer<T>
- The lambda expression becomes the body of the method in the interface

Assigning a Lambda to a Local Variable

How to print all elements in a List with Lambdas?

```
public class ArrayList<T> … {
  …
  void forEach(Consumer<T> action {
    for (T i:items) {
      action.accept(t);
  }
}
```

```
public interface Consumer<T> {
  void accept(T t);
}
```

Solution

```
class Main {
  public static void main(String[] args) {
    List<Integer> intSeq = Arrays.asList(1,2,3);

    Consumer<Integer> cnsmr = x -> System.out.println(x);
    intSeq.forEach(cnsmr):
  }
}
```

Assigning a Lambda to a method parameter

How to print all elements in a List with Lambdas?

```
public class ArrayList<T> … {
  …
  void forEach(Consumer<T> action {
    for (T i:items) {
      action.accept(t);
  }
}
```

```
public interface Consumer<T> {
  void accept(T t);
}
```

Solution

```
public class Main {
  public static void main(String[] args) {
    List<Integer> intSeq = Arrays.asList(1,2,3);

    intSeq.forEach(x -> System.out.println(x));
  }
}
```

Specify Filtering Criteria Code with a Lambda Expression

Define abstraction

```
public interface CheckPerson {
  boolean test(Person p);
}
```

Define generic print

```
public static void printPersons (
    List<Person> members, CheckPerson tester) {
    for (Person p : members)
        if (tester.test(p))
            p.print();
}
```

```
printPersons(members,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Sex.MALE
                && p.getAge() >= 18 && p.getAge() <= 25;
        }
    });
```

```
printPersons(members,
            p -> p.getGender() == Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25);
```

Specify filtering criteria code with a lambda expression-2

Reuse abstraction

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

```
printPersons(members,
            p -> p.getGender() == Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25);
```

Define generic print

```
public static void printPersons (
    List<Person> members, Predicate<Person> tester) {
    for (Person p : members)
        if (tester.test(p))
            p.print();
}
```

Specify filtering criteria code with a lambda expression-3

Reuse abstraction

```
public interface Consumer<T> {
  void consumer(T t);
}
```

```
printPersons(members,
    p -> {if (p.getGender() == Sex.MALE && p.getAge() >= 18  && p.getAge() <= 25)
               p.print(); } );
```

Define a more generic print

```
public static void printPersons (
   List<Person> members, Consumer<Person> c) {
   members.fotEach(c);
}
```

Accessing Local Variables of the Enclosing Scope

- Like anonymous classes, lambda expressions can access to:
  - to local variables (final or effectively final) of the enclosing scope
  - and fields and methods of the enclosing scope (static and non-static)
- Lambda expressions are lexically scoped:
  - Do not inherit any names from a supertype
  - Do not introduce a new level of scoping
  - Cannot define attributes
  - Use of **this** inside a lambda expression refers to the enclosing object and not to the lambda object
  - Declarations in a lambda expression are interpreted just as they are in the enclosing environment

Accessing variables - Example

```
public class A {
  private int x = 5;

  public void doSomething(int y) {
    B b = () -> System.out.println("this.toString() = " + this.toString() +
                                                    "\ntoString = " +  toString() +
                                                    "\nx = " + x + " y = " + y);
    System.out.println("A.toString() = " + this.toString());
    System.out.println("Lambda.toString()" + b.toString());
    x = 10;
    b.cc();
     x = 20;
     b.cc();
  }

  public static void main(String[] args) {
    new A().doSomething(3);
  }
}
```

```
interface B {
  void cc();
}
```

Result:

```
A.toString() = A@36baf30c
Lambda.toString()A$$Lambda$1/746292446@7a81197d
this.toString() = A@36baf30c
toString = A@36baf30c
x = 10 y = 3
this.toString() = A@36baf30c
toString = A@36baf30c
x = 20 y = 3
```

Lambdas as Objects

- A Java lambda expression is essentially an object

```
public class Person {
  private int _age;

  Person(int a) {
    _age = a;
  }


  public final int getAge() {
    return _age;
  }
}
```

```
import java.util.Comparator;
public class A {
  private int x = 5;

  public static void main(String[] args) {
    Comparator<Person> compareByAge =
          (p1, p2) -> { return p1.getAge() - p2.getAge(); };

    Person p = new Person(2);
    Person pp = new Person(5);

    int result = compareByAge.compare(p, pp);
  }
}
```

Method References as Lambdas

- A concise way to write lambda expression when:
  - Just call another method
  - With parameters given to the lambda

```
public interface MyPrinter{
  void print(String s);
}

MyPrinter printer = s -> System.out.println(s);
```

```
MyPrinter printer = System.out::println;
```

  - Double colons :: signal to the Java compiler that this is a method reference
    - Format Class or instance :: method
- Four kinds of method references

Method Reference Types

- *objectName*::*instanceMethodName* (**Instance Method Reference)**
  - Creates a lambda that:
    - invokes *instanceMethodName* on *objectName*
    - passes the lambda's arguments to the instance method
    - and returns the method's result
    - The argument types of *instanceMehodName* and lambda method must match
- *ClassName*::*staticMethodName (***Static Method Reference)**
  - Creates a lambda that
    - invokes *staticMethodName* on *ClassName*
    - passes the lambda's arguments to the static method
    - and returns the method's result
    - The argument types of *staticMethodName* and lambda method must match

Method Reference Types - 2

- *ClassName*`::`*instanceMethodName* (Parameter Method Reference)
  - Creates a lambda that
    - invokes the *instanceMethodName* on the first lambda's argument
    - passes the remaining parameters to the instance method
    - and returns the method's result
- *ClassName*`::new` **(Constructor Reference)**
  - Creates a lambda that
  - invokes one of the constructors of *ClassName*
  - passes the lambda's parameters to the constructor
  - The argument types of one of the constructors of *ClassName* and lambda method must match

```
public interface Factory {
  String create(char[] val);
}

Factory factory = String::new;
Factory factory = chars -> new String(chars);
```