Medii de programare

Lecture 5
Outline

- Functional programming

Functional interfaces

```
@FunctionalInterface
public interface Sprint {
    public void sprint(Animal animal);
}
public class Tiger implements Sprint {
    public void sprint(Animal animal) {
        System.out.println("Animal is sprinting fast!
            "+animal.toString());
    }
}
```

Which of the following is a functional interface?

```
public interface Run extends Sprint {}
public interface SprintFaster extends Sprint {
    public void sprint(Animal animal);
}
public interface Skip extends Sprint {
    public default int getHopCount(Kangaroo kangaroo)
        {return 10;}
    public static void skip(int speed) {}
}
public interface Walk {}
public interface Dance extends Sprint {
    public void dance(Animal animal);
}
public interface Crawl {
    public void crawl();
    public int getCount();
}
```

Implementing functional interfaces with lambdas

e01

The lambda syntax

```
(Animal a) -> {a.canHop();}
```

Which of the following are valid lambdas?

```
1.()   -> new Duck()
2.d    -> {return d.quack();}
3.(Duck    d) -> d.quack()
4.(Animal  a, Duck d) -> d.quack()
5.Duck     d -> d.quack()
6.a,d -> d.quack()
7.Animal   a, Duck d -> d.quack()
8.a,   b -> a.startsWith("test")
9.c    -> return 10;
10. a -> { return a.startsWith("test") }
11. (int   y, z) -> {int x=1; return y+10; }
12. (a,    Animal b, c) -> a.getName()
13. (a,    b) -> { int a = 0; return 5;}
14. (a,    b) -> { int c = 0; return 5;}
```

The Predicate interface

`java.util.function`

e02

Method references

```
public class EmployeeHelper {
     public static int compareBySalary(Employee e1,
         Employee e2) {
         return e1.getSalary() - e2.getSalary();
     }
     public static int compareByName(Employee e1,
         Employee e2){
         return e1.getName().compareTo(e2.getName());
     }
}
```

Sort employees using lambdas:

1. `Comparator<Employee> bySalary = (e1, e2) -> EmployeeHelper.compareBySalary(e1, e2);`
2. `Collections.sort(employeeList, bySalary)`

Sort employees using method references:

1. `Comparator<Employee> bySalary = EmployeeHelper::compareBySalary;`
2. `Collections.sort(employeeList, bySalary)`

*****Static method reference:**

`(Consumer<T> --- java.util.function)`

- `Consumer<List<Integer>> methodRef1 = Collections::sort; //` which sort method is called? deferred execution
- `Consumer<List<Integer>> lambda1 = list -> Collections.sort(list);`

*****Method reference on a specific instance:**

- `String str = "abc";`
- `Predicate<String> methodRef2 = str::startsWith;`
- `Predicate<String> lambda2 = s -> str.startsWith(s);`

*****Method reference on an instance to be determined at runtime:**

- `Predicate<String> methodRef3 = String::isEmpty;`
- `Predicate<String> lambda3 = s -> s.isEmpty();`

*****Constructor reference:**

`(Supplier<T> --- java.util.function)`

- `Supplier<ArrayList> methodRef4 = ArrayList::new;`
- `Supplier<ArrayList> lambda4 = () -> new ArrayList();`

Functional style collection methods

`boolean removeIf(Predicate<? super E> filter)` //remove conditionally

```
1.List<String>     list= new
  ArrayList(Arrays.asList("abc", "xyz"));
2.list.removeIf(s  -> s.startsWith("a"));
```

? Replace line 2 with a method reference

`void replaceAll(UnaryOperator<E> o)` //apply lambda to all elements

```
1. List<Integer>    list = Arrays.asList(1, 2, 3);
2.list.replaceAll(x -> x*2);
```

Looping through a collection

```
v1:
List<String> list = Arrays.asList("hello", "world");
for(String s: list){
    System.out.println(s);
}

v2:
list.forEach(s -> System.out.println(s));

v3.
list.forEach(System.out::println);
```

Functional style map methods

- *putIfAbsent*
- computeIfAbsent
- computeIfPresent
- merge

~~e03~~

Lambda scopes

- lambda expressions can access: static variables, instance variables, *effectively final* method params, *effectively final* local variables.

```
1. class App{
2.     static void convertStuff(){
3.         int  inc=1;   //effectively final local var
4.         UnaryOperator<Integer> increment = value
   -> value + inc;
5.         Integer result=increment.apply(7);
6.         System.out.println(result);
7.         //inc=2;     //compile error
8.      }
9. }
```

Built-in functional interfaces

- **Supplier** - generate values without taking any input
- **Consumer and BiConsumer** - process an argument (or two arguments), but not return anything
- **Predicate and BiPredicate -** test a condition (filter)
- **Function and BiFunction -** turn a param to something else (may be a different type)
- **UnaryOperator and BinaryOperator -** special cases of *function;* all params are of the same type

~~e04~~

What functional interface would you use?

1. Returns a *String* without taking any parameters
2. Returns a *Boolean* and takes a *String*
3. Returns an *Integer* and takes two *Integer*S

What functional interface would you use to fill in the blanks?

1. _____<List> ex1 = x -> "".equals(x.get(0));
2. _____<Long> ex2 = (Long l) -> System.out.println(l);
3. _____ <String, String> ex3 = (s1, s2) -> false;

Do the following lines compile? Explain.

1. Function<List<String>> ex1 = x -> x.get(0);
2. UnaryOperator<Long> ex2 = (Long l) -> 3.14;
3. Predicate ex3 = String::isEmpty;

Optionals

`java.util.Optional`

- a container for a value which may be null or not
- prevents *NullPointerException*
- is *not* a functional interface

e05

Streams

`java.util.stream.Stream`

e06p1