Lambda Expressions

In Java 8

Java Features

Presented by Ramin Zare

بخش اول

1. Lambda Expression ها چه هستند؟
2. Lambda Expression ها از چه جنسی هستند؟
3. گرامر

Lambda Expression ها چه هستند؟

- λ-Lambda Expression ها یک بلاک کد یا به تعبیری فانکشنی بدون نام هستند که شامل لیستی از ورودی ها و یک بدنه میباشد.
- علامت فلش (<-) برای جدا کردن بخش تعریف پارامتر ها و بدنه استفاده میشود
- مثال:

```
(int x) -> x + 1
(int x , int y) -> x + y
(int x , int y) -> {int max = x > y ? x : y;
                    return max;}
() -> { }
```

Lambda Expression ها از چه جنسی هستند؟

وقتی میگوییم :

```
(String str) -> str.length()
```

این فانکشن از چه جنسی است ؟

جواب این است که نمی دانیم !!!

Lambda Expression ها از چه جنسی هستند؟

هر عبارت لامبدا باید به نوعی پیاده سازی از هر Functional Interface باشد. Interface که حداکثر یک متود abstract داشته باشد

```
interface StringToIntMapper {

    int map(String str);

}
```

```
StringToIntMapper mapper = new StringToIntMapper() {
    @Override
    public int map(String str) {
        return str.length();
    }
};
```

```
StringToIntMapper mapper2 = (String str) -> str.length();
```

گرامر

(<lambdaParameterList>) -> { <lambdaBody> }

برخلاف متود ها عبارات لامبدا این چهار بخش را ندارند :

1. این عبارات نام ندارند
2. Return Type ندارند. جنس خروجی یک لامبدا توسط کامپایلر از جایی که استفاده میشود تشخیص داده میشود
3. گزارش Exception ندارند
4. نمیتوانند Type Parameter تعریف کنند ؛ بنابراین lambda expression ها نمیتوانند generic باشند

گرامر

`(x , y) -> {`
`  return x + y;`
`}`
 	
`(final int x ,final int y) ->`
`  x + y;`
	`int sum(int x , int y){`
	`  return x + y;`
	`}`

`str -> str.length()`	`int map(String str){`
	`  return str.length();`
	`}`

`() -> {`
`  out.print(LocalDate.now());`
`}`
	`void printCurrentDate(){`
	`  out.print(LocalDate.now());`
	`}`

`() -> {`
`}`
	`void doNothing(){`
	`  // No Code goes here`
	`}`

بخش دوم

1. Target Typing
2. رفع ابهام Target Typing
3. استفاده از Lamda Expression در کجاها مجاز است
4. Functional Interfaces

Target Typing

- هر lambda expression بیانگر یک instance از یک interface است
- عبارات لامبدا را از نوع poly expression هستند
- کامپایلر نوع تایپ lambda را حدس میزند ، فضایی که این عبارات انتظار یک تایپ خاص را دارد target type میگویند

Target Typing

این شبه کد را در نظر بگیرید

```
T t = <Lambda Expression>
```

کامپایلر از قوانین زیر برای تشخیص اینکه عبارت با T سازگار است استفاده میکند :

1. T باید Functional Interface باشد
2. آیا تعداد پارامتر های عبارت با تک متود abstract تعریف شده در T همخوانی دارد ؟
3. آیا نوع و مقدار خروجی عبارت با تک متود abstract تعریف شده در T همخوانی دارد ؟
4. اگر بدنه یک Exception از نوع checked پرتاب کند باید متود abstract آن را گزارش داده باشد.

Target Typing رفع ابهام

```
@FunctionalInterface
public interface Appender{
   String append(String str1 , String str2);
}
@FunctionalInterface
public interface Adder{
   double add(double num1 , double num2);
}
```

Target Typing رفع ابهام

```
public class Util {

    public void append(Appender appender) {

        appender.append("Hello", " How Are you ?");

    }

    public void append(Adder adder) {

        adder.add(12L, 14L);

    }

}
```

Target Typing رفع ابهام

```
Util util = new Util();

util.append((x, y) -> x + y); //Compile Time Error
```

```
Error:(21, 13) java: reference to append is ambiguous

  both method append(com.javaland.lambda.Test.Appender) in
com.javaland.lambda.Test.Util and method
append(com.javaland.lambda.Test.Adder) in com.javaland.lambda.Test.Util
match
```

رفع ابهام Target Typing

برای رفع ابهام از روش های زیر میتوانید استفاده کنید:

1. از پارامتر های explicit استفاده کنید

```
util.append((double x,double y) -> x + y);
```

2. از cast استفاده کنید

```
util.append((Appender) ((x,y) -> x + y));
```

3. مستقیما از lambda expression استفاده نکنید ابتدا آن را به یک تایپ assign کنید

```
Appender appender =(x,y) -> x + y;
util.append(appender);
```

استفاده از Lamda Expression در کجاها مجاز است

Assignment Context	`ReferenceType variable1 = <LambdaExpression>;`
Method Invocation Context	`util.append(<LambdaExpression>);`
Return Context	`return <LambdaExpression>;`
Cast Context	`(Adder) <LambdaExpression>;`

Functional Interfaces

یک functional interface یک interface است که تنها یک متود abstract دارد

وجود این عناصر در functional interface ها ایرادی ندارد :

- Default methods
- Static methods
- Public methods inherited from Object class

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

استفاده از @FunctionalInterface

وجود این annotation موجب میشود که کامپایلر تضمین کند که حتما یک متود abstract در interface وجود داشته باشد

```
@FunctionalInterface
public interface Appender {
   String append(String str1, String str2);
}
```

بخش سوم

1. Generic Functional Interface
2. Commonly Used Functional Interfaces
3. Method Reference ها
4. Variable Capture
5. Intersection Type
6. Invoke Dynamic

Generic Functional Interface

این امکان وجود دارد که Functional interface ها را generic تعریف کنید

```
@FunctionalInterface
public interface Appender<T> {
   T append(T str1, T str2);
}

public void append(Appender<String> appender) {
   appender.append("Hello", " How Are you ?");
}

Appender<String> appender = (x, y) -> x + y;

util.append((String x, String y) -> x + y);
```

Commonly Used Functional Interfaces

Function<T,R>	R **apply**(T t)
BiFunction<T,U,R>	R **apply**(T t, U u)
Predicate<T>	boolean **test**(T t)
BiPredicate<T,U>	boolean **test**(T t, U u)
Consumer<T>	void **accept**(T t)
BiConsumer<T,U>	void **accept**(T t, U u)
Supplier<T>	T **get**()
UnaryOperator<T>	T **apply**(T t)
BinaryOperator<T>	T **apply**(T t1, T t2)

Function<T,R> Interface

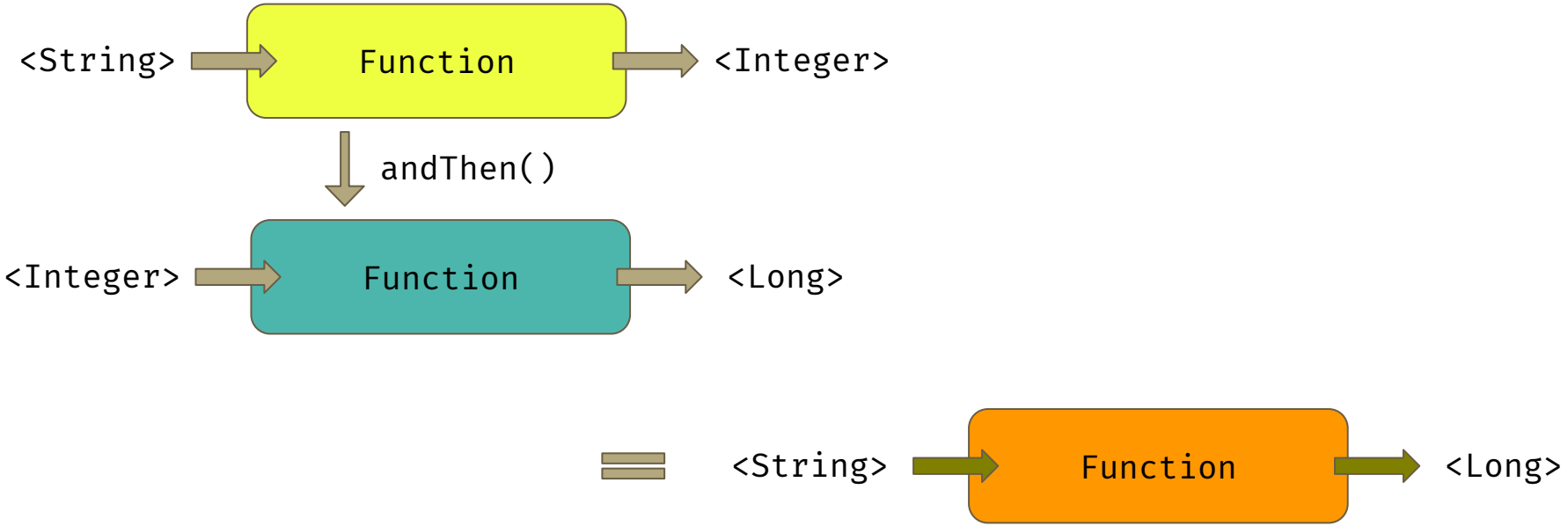
- IntFunction<R>
- LongFunction<R>
- DoubleFunction<R>
- ToIntFunction<T>
- ToLongFunction<T>
- ToDoubleFunction<T>

Function<T,R> Interface

- `default <V> Function<T,V> andThen`
- `default <V> Function<T,V> compose`
- `static <T> Function<T,T> identify`

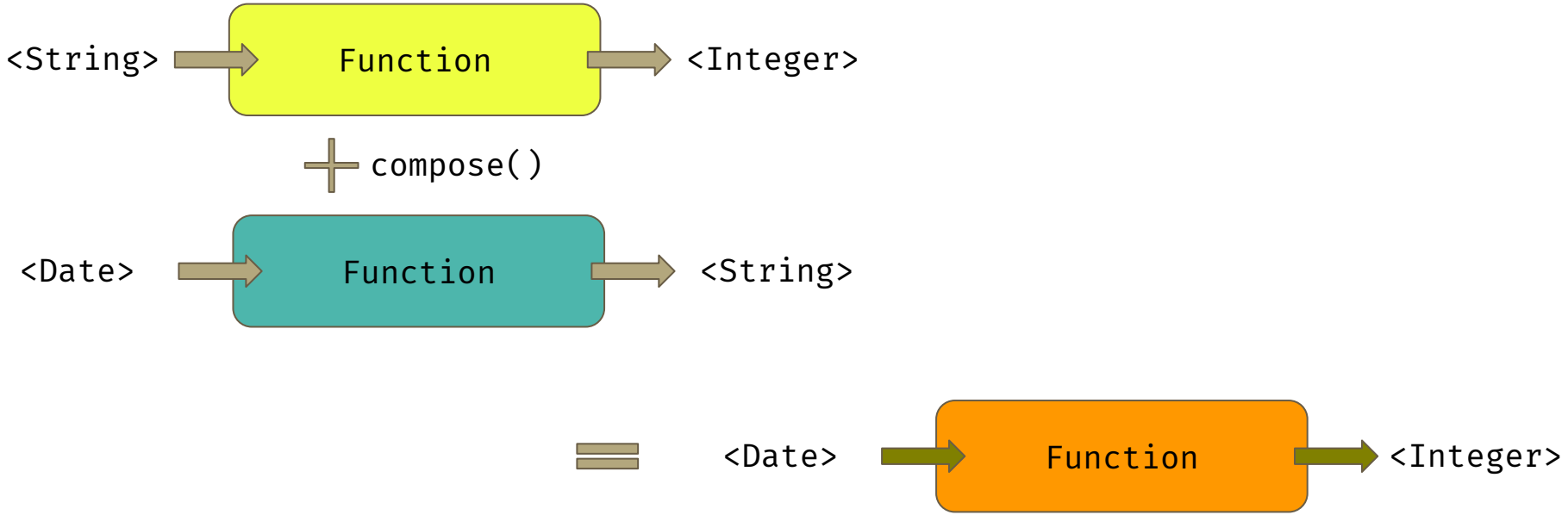
Function<T,R> Interface

```
default <V> Function<T,V> andThen
```



Function<T,R> Interface

```
default <V> Function<T,V> compose
```



Predicate<T>

- default Predicate<T> negate
- default Predicate<T> and
- default Predicate<T> or
- static <T> Predicate<T> isEqual(Object targetRef)

Using Functional Interface

Functional Interface ها معمولاً توسط دوگروه استفاده میشوند:

1. طراحان کتابخانه ها و API نویس ها
2. توسط استفاده کننده های کتابخانه ها

Method Reference ها

متود Reference ها صرفا روشی برای ساده نویسی در lambda ها محسوب میشوند

گرامر آن به شکل زیر است

```
<Qualifier>::<MethodName>
```

```
ToIntFunction<String> lengthFunction = str -> str.length();
ToIntFunction<String> lengthFunction2 = String::length;
```

```
Function<String[],List<String>> asList= Arrays::<String>asList;
```

Method Reference

`TypeName::staticMethod`	
`objectRef::instanceMethod`	**Bound Receiver**
`ClassName::instanceMethod`	**Unbound Receiver**
`TypeName.super::instanceMethod`	
`ClassName::new`	
`ArrayTypeName::new`	

Variable Capture

شبیه به یک inner و anonymous inner class یک lambda هم به متغیر های محلی effectively final دسترسی دارند با این دو شرط:

- متغیر به صورت final معرفی شود
- final تعریف نشود ولی فقط یک بار مقدار دهی شود

Intersection Type

- جاوا ۸ یک تایپ جدید به نام intersection type ها اضافه کرده است
- میتوان از این خصوصیت در cast کردن lambda ها به یک target type استفاده کرد
- علامت & برای معرفی تایپ جدید در cast مورد استفاده قرار میگیرد

```
Sensitive sen = (Sensitive & Adder) (x,y) -> x +y;
```

Invoke dynamic

- Invoke dynamic روشی است که JVM برای تولید بایت کد در زمان اجرا استفاده میکند
- در این روش به جای تولید بایت کد های مستقل در زمان کامپایل به ازای هر عبارت لامبدا , در زمان اجرا آنها را میسازد (و یا از cache واکشی میکند) و صدا میزند
- متود `java.lang.invoke.LambdaMetafactory.metafactory` این استراتژی را پیاده سازی کرده است

Thanks!

Presented by :

Ramin Zare