Functional programming with Java

Main functional features introduced with Java 8

- functional interfaces
- lambda expressions
- method (and constructor) references
- streams (no details in these slides)

Functional programming with Java

Example 1

```
IntStream intStream = IntStream.of(1, -2, -3, 4, -5, -6, -7, 16);
assert intStream.filter(i -> i >= 0).mapToDouble(Math::sqrt)
        .collect(StringBuilder::new, (x, y) -> x.append(y).append(" "),
                StringBuilder::append)
        .toString().equals("1.0 2.0 4.0 ");
```

- a lambda expression with one parameter: `i -> i >= 0`
- a reference to a static method: `Math::sqrt`
- a reference to a constructor: `StringBuilder::new`
- a lambda expression with two parameters: `(x, y)-> x.append(y).append("")`
- a reference to an instance method: `StringBuilder::append`

Functional programming with Java

Example 2

```
public record Person(String name, int birthYear) {}
...
Comparator<Person> byYearThenName = Comparator.comparing(
    Person::birthYear).thenComparing(Person::name);
SortedSet<Person> geniuses = new TreeSet<>(byYearThenName);
Person galileo = new Person("Galileo Galilei", 1564);
Person william = new Person("William Shakespeare", 1564);
Person alan = new Person("Alan Turing", 1912);
geniuses.add(galileo);
geniuses.add(william);
geniuses.add(alan);
assert geniuses.size() == 3 && geniuses.first() == galileo
       && geniuses.last() == alan;
```

- a static factory method with a parameter of type `Function`: `Comparator.comparing`
- a default factory method with a parameter of type `Function`: `Comparator.thenComparing`
- references to getter methods: `Person::name`, `Person::birthYear`

Functional interfaces

- interfaces with exactly one abstract method (although they may contain more static and default methods)
- instances of functional interfaces can be created with method reference expressions and lambda expressions
- functional interfaces may be annotated with `@FunctionalInterface`
- it is a compile-time error if an interface declaration is annotated with `@FunctionalInterface` but is not, in fact, a functional interface.

Examples in `java.util.functional`

- `BinaryOperator<T>`: `T apply(T t1, T t2)`
- `Consumer<T>`: `void accept(T t)`
- `Function<T,R>`: `R apply(T t)`
- `Predicate<T>`: `boolean test(T t)`
- `Supplier<T>`: `T get()`
- `UnaryOperator<T>`: `T apply(T t)`

Lambda expressions

- lambda expressions are always poly expressions: their types depend from the context
- lambda expressions can occur only in (non **var**) assignment, invocation, or casting contexts
- examples

```
// two declared-type parameters
(int x, int y) -> {return x + y;}
// two inferred-type parameters
// but declared and inferred types styles cannot be mixed
(x, y) -> {return x + y;}
// simplified version when a simple expression is returned
(x, y) -> x + y
// single inferred-type parameter
(x) -> x+1
// parentheses optional for single inferred-type parameter
x -> x+1
// no parameters, void block body
( ) -> { System.out.println("I am a lambda"); }
```

Types of lambda expressions

A lambda expression can only be compatible with functional interface types

```
@FunctionalInterface interface Lambda1<T> { T id(T x); }
@FunctionalInterface interface Lambda2 { Number wide(Integer x); }
@FunctionalInterface interface Lambda3 { int apply(int x); }

// compile-time error: the target type must be a functional interface
Object l = x -> x;
// compile-time error: lambda expression needs an explicit target-type
var ll = x -> x;
Lambda1<Integer> l1 = x -> x; // ok
assert l1.id(42).equals(42);
Lambda2 l2 = x -> x; // ok
assert l2.wide(42).equals(42);
l1 = (Integer x) -> x; // ok
l2 = (Integer x) -> x; // ok
// compile-time error: contravariance not supported
l2 = (Number x) -> x;
l2 = (Integer x) -> 42; // ok
// compile-time error: boxing for parameters not supported
l2 = (int x) -> x;
Lambda3 l3 = x -> x; // ok
assert l3.apply(42) == 42;
l3 = x -> Integer.valueOf(42); // ok
// compile-time error: unboxing for parameters not supported
l3 = (Integer x) -> x;
```

Local variables not declared in lambda bodies

Lambda expressions can use local variables, formal parameters, or exception parameters not declared in their bodies.
However the following rules apply:

- any local variable, formal parameter, or exception parameter used but not declared in a lambda expression must either be declared final or be effectively final
- any local variable used but not declared in a lambda body must be definitely assigned before the lambda body

Local variables not declared in lambda bodies

Example 1

```
void m1(int x) {
    int y = 1;
    // legal: x and y are both effectively final
    foo(() -> x+y);
}
void m2(int x) {
    int y;
    y = 1;
    // legal: x and y are both effectively final
    foo(() -> x+y);
}
```

Local variables not declared in lambda bodies

Example 2

```
void m3(int x) {
    int y;
    if (...) y = 1;
    // illegal: y is effectively final, but not definitely assigned
    foo(() -> x+y);
}
void m4(int x) {
    int y;
    if (...) y = 1; else y = 2;
    // legal: x and y are both effectively final
    foo(() ->; x+y);
}
void m5(int x) {
    int y;
    if (...) y = 1;
    y = 2;
    // illegal: y is not effectively final
    foo(() -> x+y);
}
void m6(int x) {
    // illegal: x is not effectively final
    foo(() -> x+1);
    x++;
}
```

Local variables not declared in lambda bodies

Example 3

```
void m7(int x) {
    // illegal: x is not effectively final
    foo(() -> x=1);
}
void m8() {
    int y;
    // illegal: y is not definitely assigned before the lambda
    foo(() -> y=1);
}
void m9(String[] arr) {
    for (String s : arr) {
        // legal: s is effectively final
        // (it is a new variable on each iteration)
        foo(() -> s);
    }
}
void m10(String[] arr) {
    for (int i = 0; i < arr.length; i++) {
        // illegal: i is not effectively final
        // (it is not final, and is incremented)
        foo(() -> arr[i]);
    }
}
```

Method references

Rules

- method reference expressions are always poly expressions
- a method reference can only be compatible with functional interface types
- in comparison with lambda expressions compatibility is more flexible

Method references

Example:

```
@FunctionalInterface interface I1 {C op(C c);}
@FunctionalInterface interface I2 {C op();}
@FunctionalInterface interface I3 {void op(int i);}
@FunctionalInterface interface I4 {void op(Integer i);}
class C {
    static C m1(C c){...}
    C m2(){...}
    static C m3(Object o){...}
    static void m4(int x){...}
    static void m5(Integer x){...}
    C(){...}
    C(C x){...}
}
...
public static void main(String[] args) {
    I1 mr = C::m1;
    mr = C::m2;
    mr = C::new; // C(C x)
    mr = C::m3;
    I2 mr2 = C::new; // C()
    I3 mr3 = C::m4;
    mr3 = C::m5; // unboxing allowed
    I4 mr4 = C::m5;
    mr4 = C::m4; // boxing allowed
}
```

Method references and binding of **`this`**

For method references to instance methods the binding of **`this`** can be defined once for all or can be deferred

```
@FunctionalInterface interface FI0 { String op(); }
@FunctionalInterface interface FI1 { String op(C c); }
public class C {
    private String str;
    public C(String str) { this.str = str; }
    public static String sm() { return "C.sm"; }
    public String im() { return "C.im:" + str; }
}
public static void main(String[] args) {
    FI0 f = C::sm; // neither C::im nor new C("one")::sm is valid
    assert f.op().equals("C.sm");
    f = new C("one")::im; // this bound to new C("one")
    // f = new C("one")::sm; // not valid
    assert f.op().equals("C.im:one");
    FI1 g = C::im; // binding of this is deferred
    assert g.op(new C("two")).equals("C.im:two");
}
```

Method references and binding of `super`

```
@FunctionalInterface interface I {String op();}
public class C1 { String m(){return "C1";} }
public class C2 extends C1 {
    String m(){
        I mr = super::m;
        return mr.op();
    }
}
...
public static void main(String[] args) {
    assert new C2().m().equals("C1");
}
```