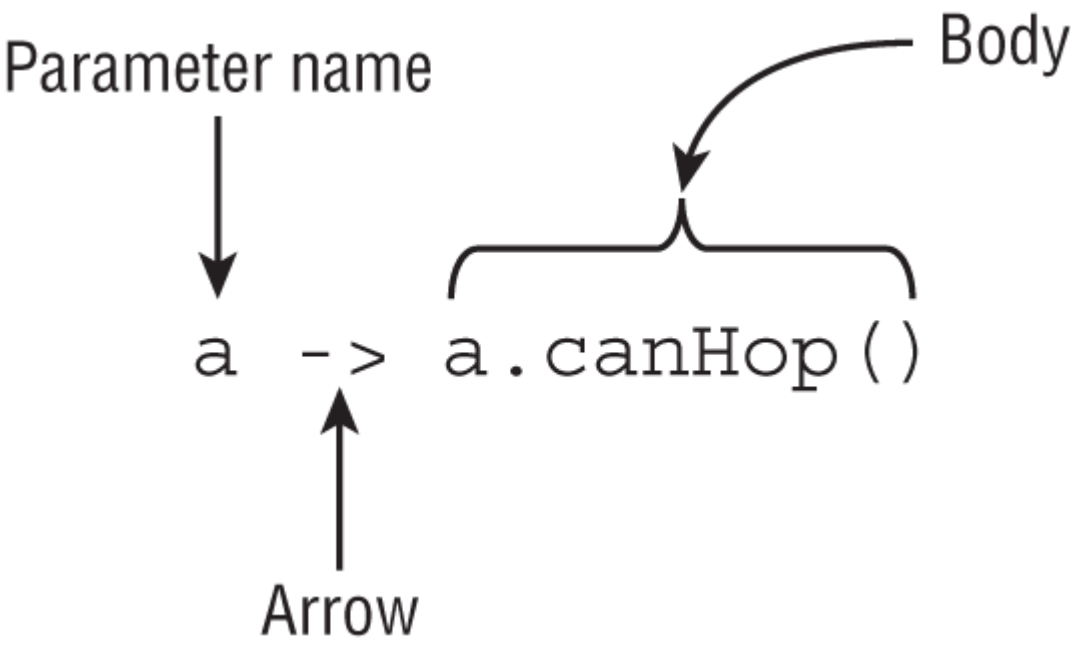
Capitolul 8 - Lambdas and Functional Interfaces

> **Functional programming**
>
> Is a way of writing code more declaratively, focusing more on expression than loops.

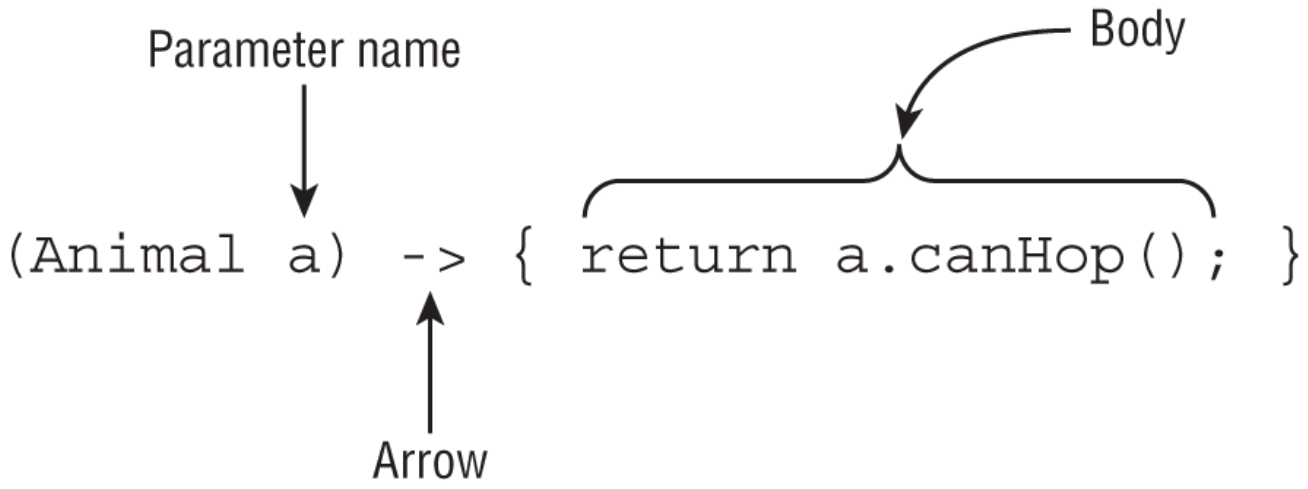
> **Lambda expression**
>
> Is a block of code that gets passed around.

- A single parameter specified with the name `a`
- The arrow operator (`->`) to separate the parameter and body
- A body that calls a single method and returns the result of that method



The second example shows the most verbose form of a lambda that returns a `boolean`.

- A single parameter specified with the name `a` and stating that the type is `Animal`
- The arrow operator (`->`) to separate the parameter and body
- A body that has one or more lines of code, including a semicolon and a `return` statement



Coding Functional Interfaces

Functional interfaces

Is an interface that contains a single abstract method.

```
@FunctionalInterface
public interface Sprint {
   public void sprint(int speed);
}

public class Tiger implements Sprint {
   public void sprint(int speed) {
      System.out.println("Animal is sprinting fast! " + speed);
   }
}

_____________________________________________________________
public interface Dash extends Sprint {} //is a functional interface

public interface Skip extends Sprint {
   void skip();
} //is not a functional interface

public interface Sleep {
   private void snore() {}
   default int getZzz() { return 1; }
} // niciuna nu este metoda abstracta

public interface Climb {
   void reach();
   default void fall() {}
   static int getBackUp() { return 100; }
   private static boolean checkHeight() { return true; }
}// is a functional interface, because exist only one abstract method
```

Method references

> 🔗 Method references
>
> Another way to make the code easier to read, such as simply mentioning the name of the method.

```
public interface LearnToSpeak {
   void speak(String sound);
}

public class DuckHelper {
   public static void teacher(String name, LearnToSpeak learner) {
      // Exercise patience (omitted)
      learner.speak(name);
   }
}

public class Duckling {
   public static void makeSound(String sound) {
      LearnToSpeak learner = s -> System.out.println(s); //redundant

      LearnToSpeak learner = System.out::println; //much better

      DuckHelper.teacher(sound, learner);
   }
```

A method reference and a lambda behave the same way at runtime. You can pretend the compiler turns your method references into lambdas for you.
There are four formats for method references.

- `static` methods

```
interface Converter {
   long round(double num);
}

14: Converter methodRef = Math::round; //using method references
15: Converter lambda = x -> Math.round(x); //using lambda
16:
17: System.out.println(methodRef.round(100.1));  // 100
```

- Instance methods on a particular object

```
interface StringStart {
   boolean beginningCheck(String prefix);
}

18: var str = "Zoo";
19: StringStart methodRef = str::startsWith;
20: StringStart lambda = s -> str.startsWith(s);
21:
22: System.out.println(methodRef.beginningCheck("A"));  // false
```

```
interface StringChecker{
    boolean check();
}

var str = "";
StringChecker methodRef = str::isEmpty;
StringChecker lambda = () -> str.isEmpty();
System.out.println(mehtodRef.check()); // return true
```

```
var str = "";

StringChecker lambda = () -> str.startsWith("Zoo");

StringChecker methodReference = str::startsWith;         // DOES NOT COMPILE

StringChecker methodReference = str::startsWith("Zoo");  // DOES NOT COMPILE
```

Neither of these works! While we can pass the `str` as part of the method reference, there's no way to pass the `"Zoo"` parameter with it. Therefore, it is not possible to write this lambda as a method reference.

- Instance methods on a parameter to be determined at runtime

```
interface StringParameterChecker {
   boolean check(String text);
}

23: StringParameterChecker methodRef = String::isEmpty;
24: StringParameterChecker lambda = s -> s.isEmpty();
25:
26: System.out.println(methodRef.check("Zoo"));  // false
```

Line 23 says the method that we want to call is declared in `String`. It looks like a `static` method, but it isn't. Instead, Java knows that `isEmpty()` is an instance method that does not take any parameters. Java uses the parameter supplied at runtime as the instance on which the method is called.

- Constructors -> is a special type of method reference that uses `new` instead of a method and instantiates an object.

```
interface EmptyStringCreator {
   String create();
}

30: EmptyStringCreator methodRef = String::new;
31: EmptyStringCreator lambda = () -> new String();
32:
33: var myString = methodRef.create();
34: System.out.println(myString.equals("Snake"));  // false
-----------------------------------------------------------------
interface StringCopier {
   String copy(String value);
}

32: StringCopier methodRef = String::new;
33: StringCopier lambda = x -> new String(x);
34:
35: var myString = methodRef.copy("Zebra");
36: System.out.println(myString.equals("Zebra"));  // true
```

Working with Built-in Functional Interfaces

Functional interface	Return type	Method name	# of parameters
Supplier<T>	T	get()	0
Consumer<T>	void	accept(T)	1 (T)
BiConsumer<T, U>	void	accept(T,U)	2 (T, U)
Predicate<T>	boolean	test(T)	1 (T)
BiPredicate<T, U>	boolean	test(T,U)	2 (T, U)
Function<T, R>	R	apply(T)	1 (T)
BiFunction<T, U, R>	R	apply(T,U)	2 (T, U)
UnaryOperator<T>	T	apply(T)	1 (T)
BinaryOperator<T>	T	apply(T,T)	2 (T, T)

Interface instance	Method return type	Method name	Method parameters
Consumer	Consumer	andThen()	Consumer
Function	Function	andThen()	Function
Function	Function	compose()	Function
Predicate	Predicate	and()	Predicate
Predicate	Predicate	negate()	–
Predicate	Predicate	or()	Predicate

Functional interfaces	Return type	Single abstract method	# of parameters

`DoubleSupplier`
`IntSupplier`
`LongSupplier`
	`double`
	`int`
	`long`
		`getAsDouble`
		`getAsInt`
		`getAsLong`
			0

`DoubleConsumer`
`IntConsumer`
`LongConsumer`
	`void`	`accept`	1 (`double`)
			1 (`int`)
			1 (`long`)

`DoublePredicate`
`IntPredicate`
`LongPredicate`
	`boolean`	`test`	1 (`double`)
			1 (`int`)
			1 (`long`)

`DoubleFunction<R>`
`IntFunction<R>`
`LongFunction<R>`
	`R`	`apply`	1 (`double`)
			1 (`int`)
			1 (`long`)

`DoubleUnaryOperator`
`IntUnaryOperator`
`LongUnaryOperator`
	`double`
	`int`
	`long`
		`applyAsDouble`
		`applyAsInt`
		`applyAsLong`
			1 (`double`)
			1 (`int`)
			1 (`long`)

`DoubleBinaryOperator`
`IntBinaryOperator`
`LongBinaryOperator`
	`double`
	`int`
	`long`
		`applyAsDouble`
		`applyAsInt`
		`applyAsLong`
			2 (`double, double`)
			2 (`int, int`)
			2 (`long, long`)

Functional interfaces	Return type	Single abstract method	# of parameters
ToDoubleFunction<T>	double	applyAsDouble	1 (T)
ToIntFunction<T>	int	applyAsInt	
ToLongFunction<T>	long	applyAsLong	
ToDoubleBiFunction<T, U>	double	applyAsDouble	2 (T, U)
ToIntBiFunction<T, U>	int	applyAsInt	
ToLongBiFunction<T, U>	long	applyAsLong	
DoubleToIntFunction	int	applyAsInt	1 (double)
DoubleToLongFunction	long	applyAsLong	1 (double)
IntToDoubleFunction	double	applyAsDouble	1 (int)
IntToLongFunction	long	applyAsLong	1 (int)
LongToDoubleFunction	double	applyAsDouble	1 (long)
LongToIntFunction	int	applyAsInt	1 (long)
ObjDoubleConsumer<T>	void	accept	2 (T, double)
ObjIntConsumer<T>			2 (T, int)
ObjLongConsumer<T>			2 (T, long)

Working with Variables in Lambdas

Listing parameters

PARAMETER LIST FORMATS

You have three formats for specifying parameter types within a lambda: without types, with types, and with `var`. The compiler requires all parameters in the lambda to use the same format. Can you see why the following are not valid?

```
5: (var x, y) -> "Hello"              // DOES NOT COMPILE
6: (var x, Integer y) -> true         // DOES NOT COMPILE
7: (String x, var y, Integer z) -> true  // DOES NOT COMPILE
8: (Integer x, y) -> "goodbye"        // DOES NOT COMPILE
```

Lines 5 needs to remove `var` from `x` or add it to `y`. Next, lines 6 and 7 need to use the type or `var` consistently. Finally, line 8 needs to remove `Integer` from `x` or add a type to `y`.

Local Variables inside a lambda body

```
11: public void variables(int a) {
12:   int b = 1;
13:   Predicate<Integer> p1 = a -> {
14:     int b = 0;
15:     int c = 0;
16:     return b == c; }
17: }
```

There are three syntax errors. The first is on line 13. The variable `a` was already used in this scope as a method parameter, so it cannot be reused. The next syntax error comes on line 14, where the code attempts to redeclare local variable `b`. The third syntax error is quite subtle and on line 16. See it? Look really closely.

The variable `p1` is missing a semicolon at the end. There is a semicolon before the `}`, but that is inside the block. While you don't normally have to look for missing semicolons, lambdas are tricky in this space, so beware!

Referencing Variables from Lambda body

Lambda bodies are allowed to reference some variables from the surrounding code. The following code is legal:

```
public class Crow {
  private String color;
  public void caw(String name) {
    String volume = "loudly";
    Consumer<String> consumer = s ->
        System.out.println(name + " says "
            + volume + " that she is " + color);
  }
}
```

The only thing lambdas cannot access are variables that are not `final` or effectively final.

Variable type	Rule
Instance variable	Allowed
Static variable	Allowed
Local variable	Allowed if `final` or effectively final
Method parameter	Allowed if `final` or effectively final
Lambda parameter	Allowed