СУ-9

Функционални интерфейси и ламбда

```java
public interface Fly {
        public int getWingSpan() throws Exception;
        public static final int MAX_SPEED = 100;
        public default void land() {
                System.out.println("Animal is landing");
        }
        public static double calculateSpeed(float distance, double time) {
                return distance/time;

        }
}

public class Eagle implements Fly {
        public int getWingSpan() {
                return 15;
        }

        public void land() {
                System.out.println("Eagle is diving fast");
        }
}
```

```java
public interface Walk {
        boolean isQuadruped();
        abstract double getMaxSpeed();
}


public interface Run extends Walk {
        public abstract boolean canHuntWhileRunning();
        abstract double getMaxSpeed();
}


public class Lion implements Run {
        public boolean isQuadruped() {
        return true;
}

public boolean canHuntWhileRunning() {
        return true;
}

public double getMaxSpeed() {
        return 100;
}
}
```

Ще се копмпилират ли следните редове?

```java
public interface Sleep extends Lion {}

public class Tiger extends Walk {}
```

- Интерфейсите се използват, за да се избегне ограничението от множественото наследяването е Java.

```
public interface Swim {
}


public interface Hop {
}


public class Frog implements Swim, Hop {
}
```

Functional Programming

- *functional interface е този който съдържа един единствен абстрактен метод.*

- Functional interfaces се използват като базисни за lambda expressions в функционалното програмиране.

- *lambda expression е блок от кода, който взима подадените данни*, подобно на анонимен метод.

Дефиниране на функционален интерфейс

```java
@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());
    }
}
```

Разгледайте интерфейсите и Sprint вече е дефиниран, кой от тях е функционален интерфейс?

```
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) {}
}
```

Тези интерфейси функционални ли са?

```java
public interface Walk {}

public interface Dance extends Sprint {
    public void dance(Animal animal);
}


public interface Crawl {
    public void crawl();
    public int getCount();
}
```

Реализиране на функционални интерфейси с Lambdas

```java
public class Animal {
    private String species;
    private boolean canHop;
    private boolean canSwim;
    public Animal(String speciesName, boolean hopper, boolean swimmer) {
        species = speciesName;
        canHop = hopper;
        canSwim = swimmer;
    }
    public boolean canHop() { return canHop; }
    public boolean canSwim() { return canSwim; }
    public String toString() { return species; }
}

public interface CheckTrait {
    public boolean test(Animal a);
}
```

```java
public class FindMatchingAnimals {
    private static void print(Animal animal, CheckTrait trait) {
        if(trait.test(animal))
            System.out.println(animal);
    }


    public static void main(String[] args) {
        print(new Animal("fish", false, true), a -> a.canHop());
        print(new Animal("kangaroo", true, false), a -> a.canHop());
    }
}
```

```
a -> a.canHop()

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

FIGURE 2.1   Lambda syntax omitting optional parts
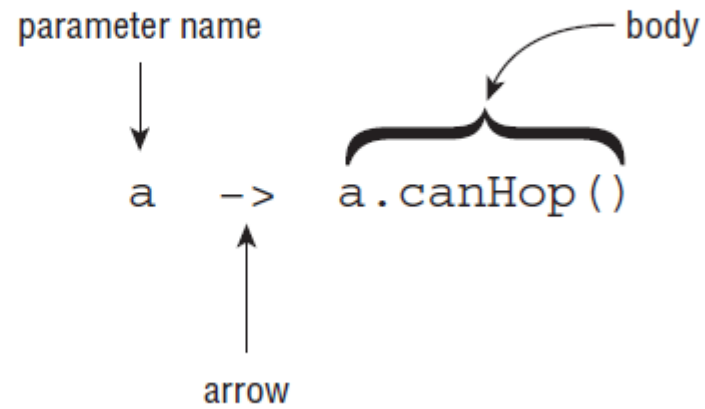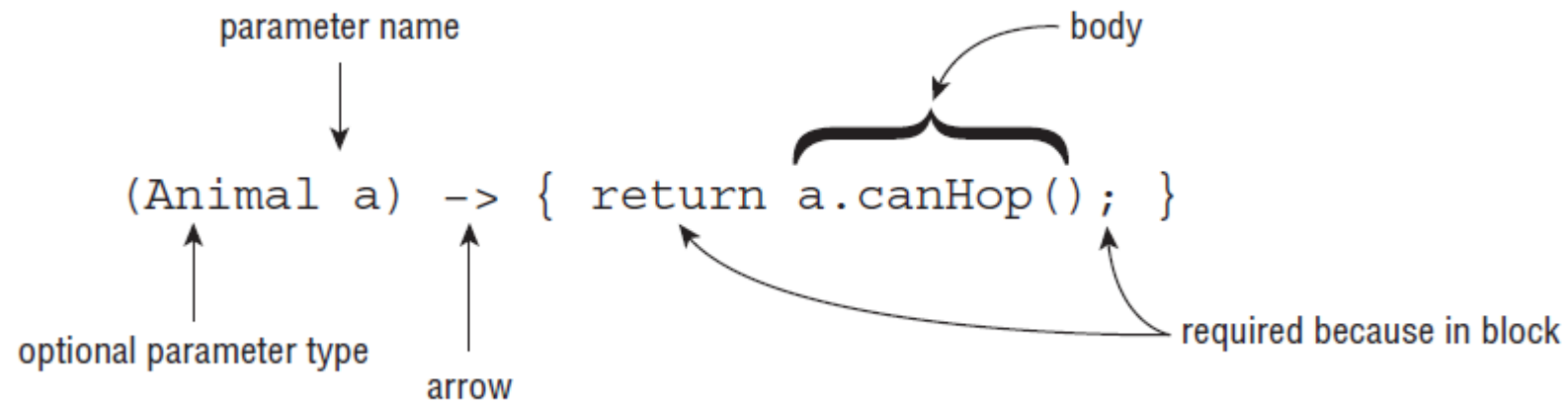
parameter name

body

a   ->   a.canHop()

arrow

FIGURE 2.2   Lambda syntax, including optional parts

parameter name

body

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

optional parameter type

arrow

required because in block

```
() -> new Duck()
d -> {return d.quack();}
(Duck d) -> d.quack()
(Animal a, Duck d) -> d.quack()
```

```
Duck d -> d.quack()
a,d -> d.quack()
Animal a, Duck d -> d.quack()
```

Още примери:

```
() -> true                                  // 0 parameters
a -> {return a.startsWith("test");}         // 1 parameter
(String a) -> a.startsWith("test")          // 1 parameter
(int x) -> {}                               // 1 parameter
(int y) -> {return;}                        // 1 parameter
```

As mentioned, the data types for the input parameters of a lambda expression are optional. When one parameter has a data type listed, though, all parameters must provide a data type. The following lambda expressions are each invalid for this reason:

```
(int y, z) -> {int x=1; return y+10; }        // DOES NOT COMPILE
(String s, z) -> { return s.length()+z; }      // DOES NOT COMPILE
(a, Animal b, c) -> a.getName()                // DOES NOT COMPILE
```

If we add or remove all of the data types, then these lambda expressions do compile. For example, the following rewritten lambda expressions are each valid:

```
(y, z) -> {int x=1; return y+10; }
(String s, int z) -> { return s.length()+z; }
(a, b, c) -> a.getName()
```

There is one more issue you might see with lambdas. We've been defining an argument list in our lambda expressions. Since Java doesn't allow us to re-declare a local variable, the following is an issue:

```
(a, b) -> { int a = 0; return 5;}              // DOES NOT COMPILE
```

We tried to re-declare a, which is not allowed. By contrast, the following line is permitted because it uses a different variable name:

```
(a, b) -> { int c = 0; return 5;}
```