СУ-9-10

Функционални интерфейси и ламбда

```
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");
		}
}
```

```
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 компилират ли следните редове?

```
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

```
@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?

```
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

```
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);
}
```

```
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

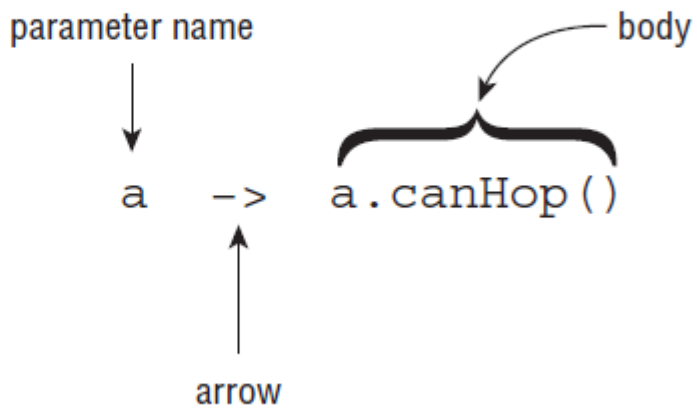
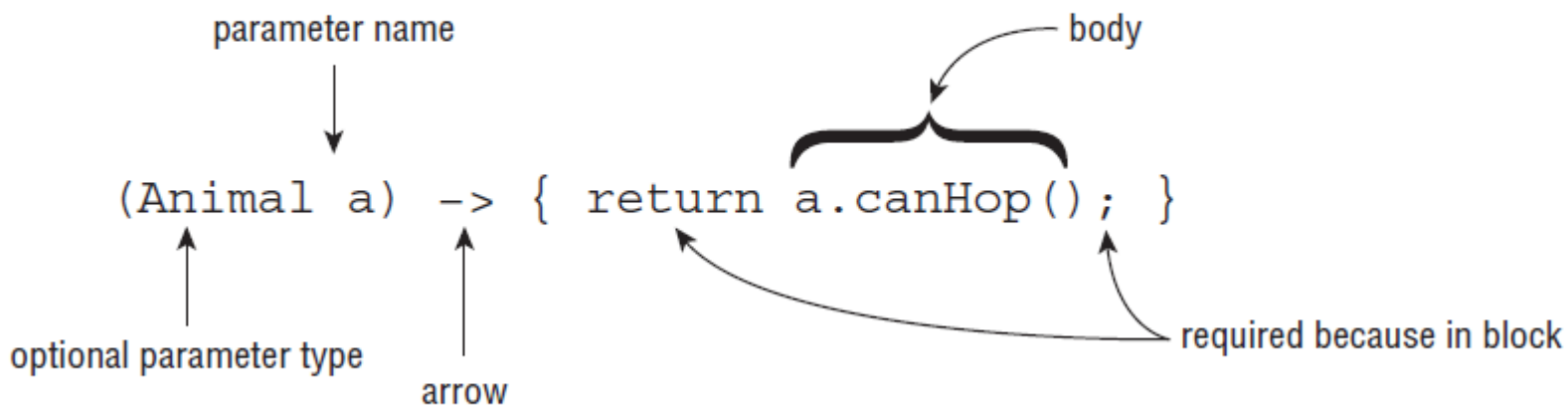


FIGURE 2.2 Lambda syntax, including optional parts



```
() -> 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;}
```

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

За предсказване се използва интерфейса:

```
public interface Predicate<T> {
        public boolean test(T t);
}
```

```
import java.util.function.Predicate;

public class FindMatchingAnimals {
        private static void print(Animal animal, Predicate<Animal> 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());
        }
}
```

Implementing Polymorphism

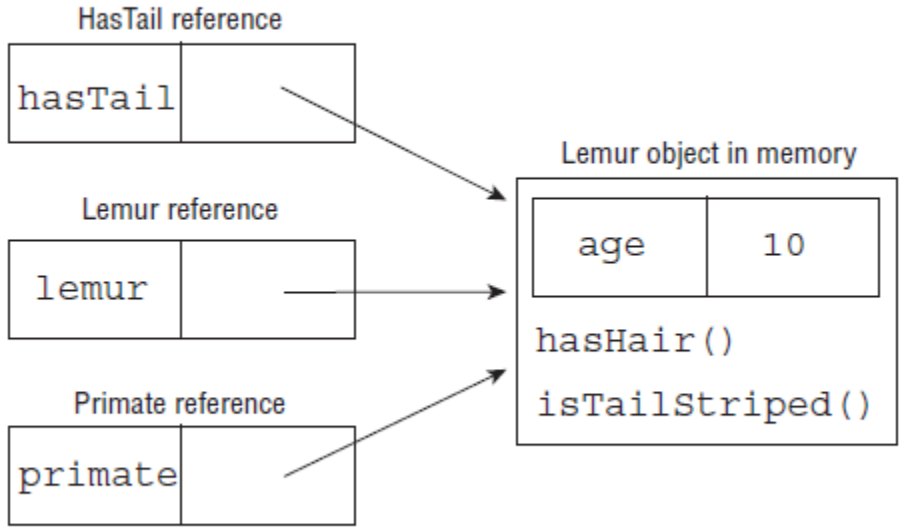
```
public interface LivesInOcean { public void makeSound(); }

public class Dolphin implements LivesInOcean {
        public void makeSound() { System.out.println("whistle"); }
}
public class Whale implements LivesInOcean {
        public void makeSound() { System.out.println("sing"); }
}
public class Oceanographer {
        public void checkSound(LivesInOcean animal) {
                animal.makeSound();
}
public void main(String[] args) {
        Oceanographer o = new Oceanographer();
        o.checkSound(new Dolphin());
        o.checkSound(new Whale());
}
}
```

```
public class Primate {
   public boolean hasHair() {
      return true;
   }
}

public interface HasTail {
   public boolean isTailStriped();
}

public class Lemur extends Primate implements HasTail {
   public int age = 10;

   public boolean isTailStriped() {
      return false;
   }

   public static void main(String[] args) {
      Lemur lemur = new Lemur();
      System.out.println(lemur.age);

      HasTail hasTail = lemur;
      System.out.println(hasTail.isTailStriped());

      Primate primate = lemur;
      System.out.println(primate.hasHair());
   }
}
```

```
HasTail hasTail = lemur;
System.out.println(hasTail.age);              // DOES NOT COMPILE

Primate primate = lemur;
System.out.println(primate.isTailStriped());  // DOES NOT COMPILE
```

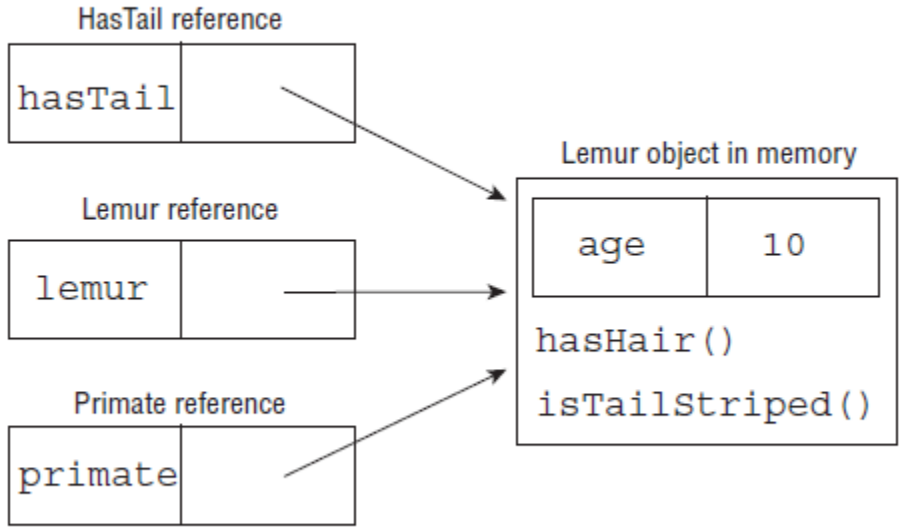
Distinguishing between an Object and a Reference

FIGURE 2.3 Object vs. reference



Distinguishing between an Object and a Reference

FIGURE 2.3 Object vs. reference



Casting Object References

```
Primate primate = lemur;

Lemur lemur2 = primate; // DOES NOT COMPILE

Lemur lemur3 = (Lemur)primate;
System.out.println(lemur3.age);
```

```
public class Bird {}

public class Fish {
   public static void main(String[] args) {
      Fish fish = new Fish();
      Bird bird = (Fish)bird;  // DOES NOT COMPILE
   }
}
```

```
public class Rodent {
}

public class Capybara extends Rodent {
   public static void main(String[] args) {
      Rodent rodent = new Rodent();
      Capybara capybara = (Capybara)rodent;  // Throws ClassCastException at
                                             // runtime
   }
}
```

```
if(rodent instanceof Capybara) {
   Capybara capybara = (Capybara)rodent;
}
```

Understanding Design Principles

- More logical code
- Code that is easier to understand
- Classes that are easier to reuse in other relationships and applications
- Code that is easier to maintain and that adapts more readily to changes in the
- application requirements