Interfaces and anonymous functions

In this discussion, we consider only interfaces that have one abstract method[1], like interface `F1` to the right. Below it is a class `C` that implements `F1` and therefore overrides method `m`.

```
interface F1 {
   Integer m(String s);
}
```

In another class `D`, we create an instance of `C` and then call its method `m`:

```
class C implements F1 {
   public Integer m(String s) {
      return Integer.valueOf(s);
   }
}
```

(1)
```
public class D {
   public static void main(String[] pars) {
      C v1= new C();
      int k= v1.m("34");
      …
   } }
```

Now, because interface `F1` has only *one* abstract method, we don't need to use class `C`. Instead, we can declare `v1` with type `F1` and assign an anonymous function to `v1`; below, the anonymous function is written in red. It's the same function that resides in class `C`.

(2)
```
public class D {
   public static void main(String[] pars) {
      F1 v1= s -> Integer.valueOf(s);     // no need to give the type of parameter s; it's inferred from F1.
      int k= v1.m("34");
      …
}}}
```

We claim that class `D` in (2) is equivalent to class `D` in (1). The difference is that (1) requires the use of a class `C` to contain the function to be called, while (2) gives the same function as an anonymous function. Class `D` in (2) is preferred because it is shorted and doesn't require class `C`.

What is the class of variable v1 in (2)?

A variable such as `v1` in (2) should point at an object of a class that implements `F1`. But the programmer did not have to write such a class! What's going on? Below, we insert a statement after the assignment to `v1` to print the class name of `v1`. Also, we put a second set of statements, this time assigning to a local variable `v2`.

A call on `main` prints the output shown to the right. We see this: For each anonymous function assigned to a variable, `v`, Java *introduced* a class that contains the anonymous function (giving it a name) and created one instance of the class and assigned it `v`. One can surmise that the integer at the end of each class name is the place in memory of the object. Remember, only one object is created of each class.

```
v1's class: D$$Lambda$2/00000000C1C1B740
v2's class: D$$Lambda$3/00000000C1C31780
```

```
public class D {
   public static void main(String[] pars) {
      F1 v1= s -> Integer.valueOf(s);
      int k= v1.m("34");
      System.out.println("v1's class: " + v1.getClass().getName());

      F1 v2= s -> Integer.valueOf(s);
      int h= v2.m("45");
      System.out.println("v2's class: " + v2.getClass().getName());
   }
}
```

In the JavaHyperText entry for anonymous function, on the line that links to this pdf file, you will find a zip file the contains a file `D.java` that contains the above class. Thus, you can see for yourself what is printed.

---

[1] Java 8 and beyond refers to an interface with one abstract function a "Functional Interface" and will put the annotation @FunctionalInterface on it in theAPI documentation. Note two points. (1) default methods have an implementation, so they are not abstract. (2) An interface may declare an abstract method that overrides a public method of java.lang.Object; it does not count as an abstract method.

An anonymous function as argument of a method call

One doesn't often see anonymous functions used as shown on the previous page. More generally, one uses anonymous functions as arguments in method calls. We illustrate below that the same mechanism shown above is in play.

To the right is an interface `Pred` (standing for *predicate* —a function that returns a boolean value). Underneath it, method `m` has parameter `p` of type `Pred`. How is the call

```
m( b -> b % 2 == 0)
```

executed? You know that one step of the call is to assign the argument to the parameter. Thus, this statement is executed:

```
p= b -> b % 2 == 0;
```

```
interface Pred {
    boolean test(int k);
}
```

```
void m(Pred p) {
    …
}
```

This is the kind of assignment discussed on the previous page: the assignment of an anonymous function to a variable whose type is an interface with one abstract method in it. Therefore, Java treats this just as on the previous page, creating a class with method test in it, where method `test` is the anonymous function.

Examples illustrate the simplification provided by anonymous functions as arguments

We show how anonymous functions make some tasks extremely simple, using interface `Pred`, given above.

Study function `check`, shown to the right. Its second parameter, `p`, has interface type `Pred`. The function returns true iff each call `p.test(b[k])` returns true —in other words: "iff every element of array `b` satisfies predicate `p`."

```
/** Return true iff every element b[k]
    * satisfies p, i.e. p(b[k]) is true. */
public static boolean check(int[] b, Pred p) {
    for (int k= 0; k < b.length; k= k + 1) {
        if (!p.test(b[k])) return false;
    }
    return true;
}
```

Below, we show a method `main` with five calls on `check`, to check whether an array contains only odd values, to check whether an array contains only positive values, and to check whether an array contains only powers of 2.

The last anonymous function in method `main` calls function `isPowerOf2`, shown to the right.

```
/** Return true iff v is a power of 2. */
public static boolean isPowerOf2(int b) {
    if (b <= 0) return false;
    // invariant: The original b is a power of 2
    //                iff the current value of b is.
    while (b % 2 == 0) { b= b / 2; }
    return b == 1;
}
```

Thus, once function `check` has been written, it's easy to write a call on it to check whether each array element satisfies just about any property; just describe that property with an anonymous function.

```
public static void main(String[] pars) {
    int[] c1= { 3, 5, 7, -9 };
    int[] c2= { 3, 5, 7, 10 };
    int[] c3= { 1, 2, 4, 16, 64 };

    System.out.println("All elements of c1 are odd: " + check(c1, v -> v % 2 == 1));
    System.out.println("All elements of c2 are odd: " + check(c2, v -> v % 2 == 1));
    System.out.println("All elements of c1 are positive: " + check(c1, v -> v > 0));
    System.out.println("All elements of c2 are positive: " + check(c2, v -> v > 0));
    System.out.println("All elements of c3 are powers of 2: " + check(c3, v -> isPowerOf2(v)));
}
```

In the JavaHyperText entry for anonymous function , on the line that links to this pdf file, a zip file contains a file `E.java` that contains interface `Pred` and methods `main`, `check`, and `isPowerOf2`.