Functions and Lambda Notation

Java 8 includes a new syntax called *lambda notation* (sometimes called *closures*). These can be used similarly to anonymous local classes, and in many cases can replace ugly anonymous class syntax with an easier to use syntax.

Lambda expressions are anonymous functions which are intended to replace the bulkiness of anonymous inner classes, with a much more compact and efficient mechanism. (The name comes from the original mathematical work on these, where the author used the Greek letter λ (lambda) to denote these; for Java, a better name would be "arrow notation" since you denote these with an arrow.)

> Java now has real functions! Unlike a method, a ***function*** is not associated with any object. You can use lambda notation to define functions, which can be passed to or returned from methods (or functions) and stored in variables. While anonymous classes are stored in on-disk `.class` files, functions are not.
>
> To allow functions to be useful, methods that used to only accept objects that implemented a specific interface now (since Java 8) accept either such an object, or a function. This only works if the interface in question is a ***functional interface*****:** an interface with exactly one abstract method.
>
> While often used simply because the result is less (but clearer) code, adding functions was a huge change in Java.

Lambda expressions can come in several forms:

a. `(int x, int y) -> {return x + y;}`

b. `radians -> 3.14159 * radians`

c.
```
new Thread( () -> {
   System.out.println("Hello World!"); }
).start();
```

d. `btn.addActionListener( ae -> {...} );`

e.
```
Collections.sort( myList,
   (s1,s2) -> s1.length() - s2.length() );
```

The first expression (a) simply returns the addition of two integers. In this case, the return type is inferred to be an integer. The other expressions omit

the "`return`" keyword and parameter types. Example (b) omits the parenthesis and the brackets as well: the parens are optional when there is a single argument and you don't specify the type; the brackets when the method body is a single expression. (Brackets have be used if the lambda expression contains multiple statements.)

In essence, the compiler has been made much smarter about determining the types of parameters and returns, so you don't have to type them in. The method call tells the compiler the type expected, and the lambda expression is "mapped" to that.

Example (d) shows adding a new `ActionListener` object: The compiler knows the argument to `addActionListener` must be an `ActionListener` object, and that interface has a single method "`actionPerformed`". So that must be the method provided, even if you pass a lambda (an anonymous method with no name) as long as it takes one `ActionEvent` argument and returns void. Look at how much typing that saved!

In this example, notice how the local variable `minAge` is used:

```
    peopleList.removeAll(  e-> e.getAge() < minAge
);
```

Normally these need to be made `final`, but with Java 8, the compiler does that for you. (The official term is ***effectively final***. Note that the compiler may not catch all violations of this, but it's still a syntax error to modify variables from the enclosing scope.)

There are some differences between lambda expressions (a.k.a. functions) and anonymous classes, such as:

- Lambda functions can only instantiate ***functional interfaces*** (interfaces with a single (abstract) method), such as `Comparable`. If the interface contains multiple methods, or you need to declare fields, you must still use (anonymous) inner classes. Java 8 includes many pre-defined functional interfaces. You can define methods that take these as arguments, and later pass in a lambda. See `java.util.function`.
- Lambda functions have different scoping rules ("`this`" means something different). These rules are similar to the notion of *closures* from other languages, such as JavaScript.
- Lambda functions perform better then methods (due to the use of the `invokedynamic` JVM instruction used to implement them).

- Lambda functions don't compile to inner classes, so you avoid having `Foo$1`, `Foo$2`, ... classes. The compiler stores information about the method in the class, instead of creating a new class. At runtime, that information ("recipe") is used to construct an appropriate "method", which is then invoked. With the old way, the runtime must load a class, invoke a constructor, and then invoke the method.
- Lambda functions don't have to be anonymous. Consider this snippet of code, similar to example (e) above:

```
final Comparator<String> COMPARE_BY_LENGTH =
   (s1,s2) -> s1.length() - s2.length();
Collections.sort( myList, COMPARE_BY_LENGTH );
```

Method References

Method references are just a notational short-cut for a lambda expression that simply invokes a single method. That is, if one method does nothing more than pass its argument to another method like this:

```
btn.addActionListener(
 ae -> { System.out.println( ae ); } );
```

You can simplify your code in Java 8 using a method reference, like so:

```
btn.addActionListener( System.out::println );
```

Both will display the `ActionEvent` object as a `String` to the console.

See if you can work out what this code does (`myArray` is an array of `String`s):

```
Arrays.sort( myArray, String::compareToIgnoreCase
);
```