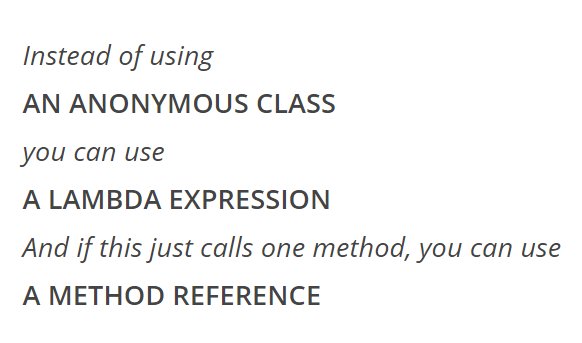
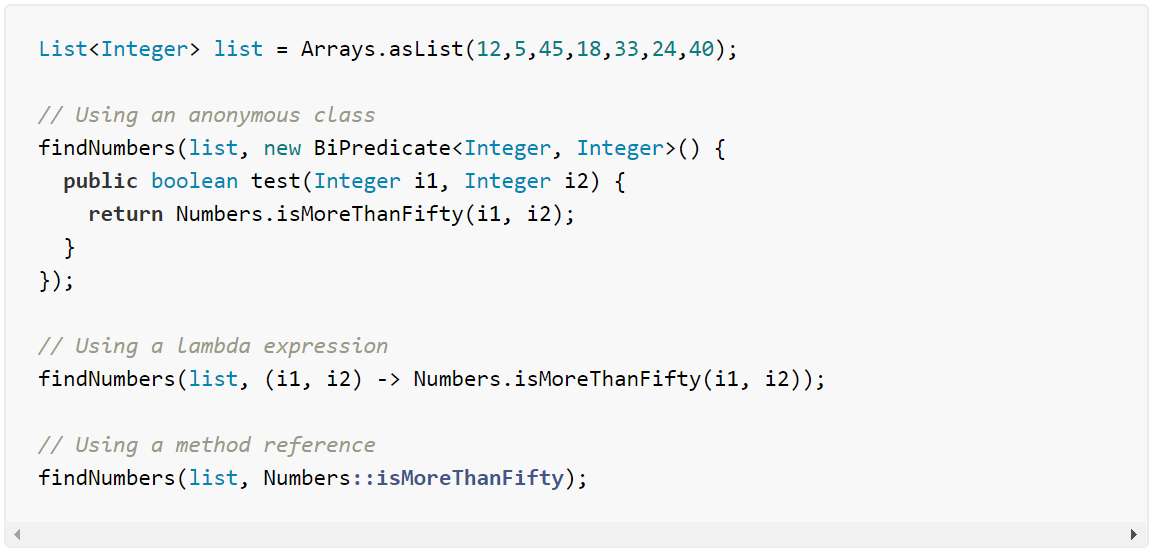
1. **Lambda expression**: A lambda expression is an anonymous method whose target type is a functional interface.
   1. Addition of lambda expression resulted in ability to take more advantage of parallel processing and new stream API which supports pipelining operations on data.
   2. LE results in a form of anonymous class.
   3. **Functional Interface**: A functional interface is an interface have only one abstract method.
      1. **Ex**: Runnable, Callable, Comparator
   4. A LE can only be used in context of functional interface. This context is created once LE is applied to FI reference. Type of the lambda must be compatible with that of FI’s abstract method.
      1. FuctionalInterface fi = () -> {…}
   5. There can be multiple LE corresponding to a FI:
      1. FuctionalInterface fi = n -> n/2
      2. FuctionalInterface fi = n -> n%2
   6. The type of parameter and return type generally type inferred by compiler by looking at FI abstract method.
   7. A FI can have public methods of Object class because they are anyway implemented by the instance of FI. This is called type inference.
   8. **LE gives us a way to pass executable code as an argument to method**.
   9. If LE throws any checked exception than it must be compatible with FI’s AM.
   10. When a lambda expression is applied to target type then an instance is created implementing functional interface where lambda expression defines the behavior of the abstract method.
2. **Syntax**: ()[args] ->[goes to] {} [body]
3. **Variable capture**: Local variable of enclosing must be effectively final for LE to use it.
   1. LE as implicit and explicit reference to ‘this’ which is enclosing class. It can call access enclosing class members. A LE doesn’t have ‘this’ of their own.
4. Checked exception thrown by LE must be compatible with the FI abstract method.
5. **Method Reference**: A method reference provides a way to refer a method without executing it.
   1. lambda is nothing but a code which you pass to a function to execute. If you already have that code in form of a method then instead of passing new code as lambda you can pass method reference.
   2. We know that we can use lambda expressions instead of using an anonymous class. But sometimes, the lambda expression is a call to some method, for example: Consumer<**String**> c = s -> System.out.println(s);

To make the code clearer, you can turn that lambda expression into a method reference: Consumer<**String**> c = System.out::println;

* 1. 



* 1. To use a method reference, you first need a lambda expression. To use a lambda expression, you need a functional interface, an interface with one abstract method.
  2. **Static method reference**: For lambda expression like the one below: **(args) -> Class.staticMethod(args)**

This can be turned into the following method reference: **Class::staticMethod**

* 1. **Instance method reference**:
     1. **Instance method reference of an existing** **object**: In this case, we have a lambda expression like the following: **(args) -> obj.instanceMethod(args)**

This can be turned into the following method reference:

objRef::methodName

**The key, in this case, is to use any object visible by an anonymous class/lambda expression and pass some arguments to an instance method of that object.**

* + 1. **Instance method ref of an object of particular type:** In this case, we have a lambda expression like the following:

**(obj, args) -> obj.instanceMethod(args)**

Where an instance of an object is passed, and one of its methods is executed with some optional(s) parameter(s). This can be turned into the following method reference:

ClassName::instanceMethodName

**The key point here is that an instance of the object is the parameter of the lambda expression, and we form the reference to the instance method with the type of the instance.**

* 1. **Constructor method reference**: In this case, we have a lambda expression like the following: **(args) -> new ClassName(args)**

That can be turned into the following method reference: **ClassName::new**

**The only thing this lambda expression does is to create a new object and we just reference a constructor of the class with the keyword new. Like in the other cases, arguments (if any) are not passed in the method reference**

* + 1. MyClass<Integer>::new
    2. Type[]::new

1. **Predefined FI**:
   1. **UnaryOperator**<T>: operate on object of type T then return result of type T.
      1. Method: T apply(T t)
   2. **BinaryOpeartor**<T>: operate on two object of type T then return result of type T
      1. Method: T apply(T t1, T t2)
   3. **Consumer**<T>: Apply an operation on an object of type T and return nothing.
      1. Method: void accept(T t)
   4. **Supplier**<T>: Return an object of type T and takes no parameter
      1. Method: T get()
   5. **Function**<T, R>: Apply operation on object of type T and return a result of type R
      1. Method: R apply(T t)
   6. **Predicate**<T>: Determine if object of type T fulfills some constraint.
      1. Method: Boolean test(T t)
2. Guidelines: <http://www.baeldung.com/java-8-lambda-expressions-tips>