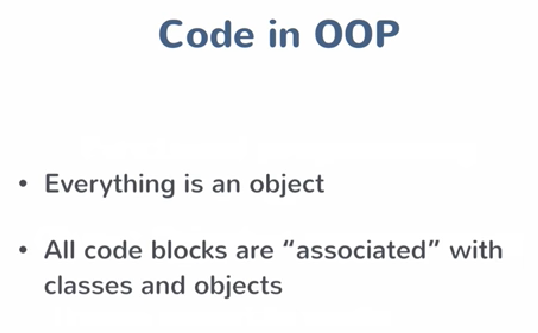


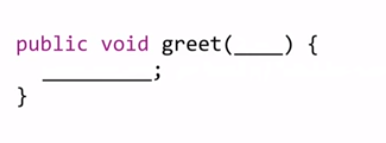
With multi-core CPUs

In certain situations.



There may be some instance when you do not need a class. You just needed a function.

There may be some instance when you do not need a class. You just needed a function.



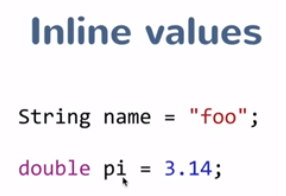
Generally, we pass parameters to functions. But what if we pass behaviour to a function. We cannot do it directly. We have to pass a class and then we will execute a method of that class, i.e. SomeClass.someMethod();

And if you want that greet to dynamically execute some behaviour then you will have an interface as an argument, and runtime polymorphism kicks in.

So when we pass a class or an interface we are not passing the behaviour. We are passing a “thing” which has a behaviour.

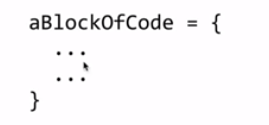


Lambdas help us to achieve this. They do not belong to a class. Lambda functions can be passed as values inline.

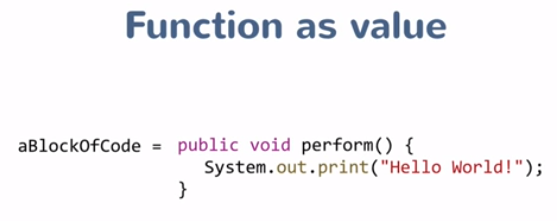


We can have Objects also defined inline. You can have objects assigned to a variable.

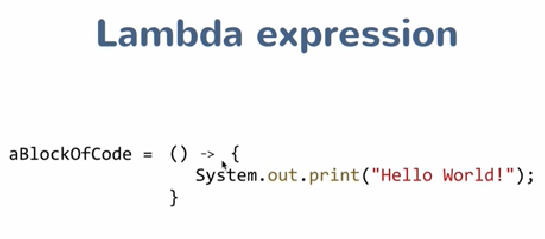
Can we have a block of code assigned to a variable as a value?



In the above case the line of code is assigned to a variable. Its not the execution result of the lines of code but its actual line of code. Something like this:



And then you can have that variable being passed around (as any other “traditional” variable) and get executed.

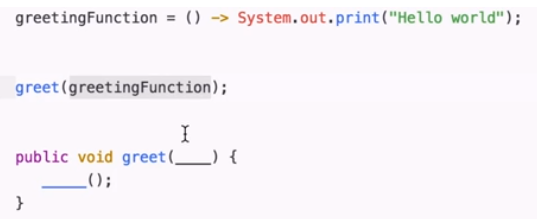


So if you have to convert a method into a lambda expression following these steps:

1. Remove the access modifiers (public, private, etc) of the method.
2. Remove the return type of the method because the compiler figures it out of the body of the method.
3. Remove the name of the method because you do not need a name now. You already have assigned a name to that piece of code, in this case the name is “aBlockOfCode”.
4. Put “->” between the () and block of code;
5. If you have just one line in the code then you do not need the curly braces



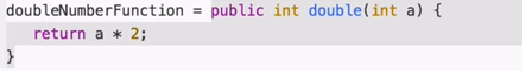
So now we can have something like this (passing a behaviour to a method):



Or…..

We do not need to declare it as a variable, if do not have to reuse it:





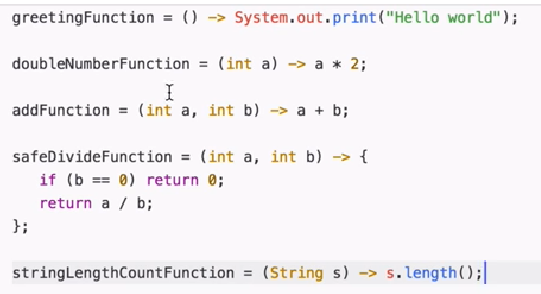
Gets converted to a Lambda expression



If the body of the method is just single line then need not have return statement as well. We can just have the body of the method and compiler will automatically return the outcome.



Some basic examples:



**What is the type of these variables?**

To wrap our lambda expression and provide it a type we need to create a Functional Interface.

Then you just declare the method in the interface, and then you create a variable of the interface type with the lambda expression matching the body:

public interface MyFunctionalInterface {

Void foo();

}

public class SomeClass {

Public void someMethod() {

MyFunctionalInterface lambdaExpr = () -> sysout(“Hello Lambda!!”);

}

}

Lambda expression becomes an implementation of the interface, but we are just implementing the function (inline) and not defining any class. This is similar to defining anonymous inner class implementing the interface:

public interface MyFunctionalInterface {

public void foo();

}

Public class MyImplementation implements MyFunctionalInterface {

public void foo() {

Sysout(“Hello from Implementation class!!”);

}

}

public class SomeClass {

public void someMethod() {

// We have 3 ways of using foo()

MyFunctionalInterface lambdaExpr = () -> sysout(“Hello Lambda!!”);

MyFunctionalInterface implClass = new MyImplementation();

MyFunctionalInterface inlineImpl= new MyFunctionalInterface() {

public void foo() {

sysout(“Hello from inline implementation!!”);

}

};

// Now we can use all the three implementations

lambdaExpr.foo();

implClass.foo();

inlineImpl.foo();

}

}

So sometimes you can also think that when we declare a lambda expression we are declaring an anonymous inner class. Both are not same though.

An advantage here is that you do not need to implement the interface to provide different implementations. You can provide dynamic/different implementation without defining classes implementing the interface:

public class SomeClass {

Public void someMethod() {

MyFunctionalInterface lambdaExpr = () -> sysout(“Hello Lambda!!”);

MyFunctionalInterface lambdaExpr2 = () -> sysout(“Hello World!!”);

MyFunctionalInterface lambdaExpr3 = () -> sysout(“Hello User!!”);

}

}

And if you have an interface which has a method having the same signature as the lambda expression you are declaring then you do not need to create another functional interface. You can use the already existing interface because the compiler does not care about the type of interface being used. It just cares about the signature (the input parameters and return type) of the lambda expression.

A new type was not created, instead they used an interface (called functional interface) so that Lambda expressions could have backward compatibility. This means that all the interfaces, like Runnable, can be used with Lambda expressions.

## Type Inference

With lambda the compiler tries to do type inference which means that it tries to identify the type of input and output parameters. Type inference makes it possible to pass lambda expression as input parameter:

**public** **interface** Greeting {

**public** **void** perform();

}

**public** **class** Greeter {

**public** **static** **void** main(String[] args) {

Greeter greeter = new Greeter();

greeter.greet(() -> System.out.println("Hello Lambda!!"));

}

**public** **void** greet(Greeting greeting) {

greeting.perform();

}

The compiler looks at the Type (i.e. Greeting) of the Input parameter accepted by greet() and automatically tries to validate the lambda expression based on the type. This is type inference.

By virtue of Type inference the compiler can gather the information about the lambda expression being passed.

The compiler is even able to determine the type of the argument. Hence if there is single argument then you do not need to mention the ()

**public** **interface** LengthLamdba {

**public** **int** getIntValue(String str);

}

**public** **class** LengthPrinter {

**public** **static** **void** main(String[] args) {

LengthLamdba strLen = s -> s.length();

System.***out***.println(strLen.getIntValue("Hello Lambda"));

*printInteger*(s -> s.indexOf('L'));

}

**private** **static** **void** printInteger(LengthLamdba lengthLamdba ) {

**int** intValue = lengthLamdba.getIntValue("Welcome to Lambda");

System.***out***.println(intValue);

}

}

And if there is no input parameter then we need to put ():

greeter.greet(() -> System.out.println("Hello Lambda!!"));

## Functional Interface

In Java 8 an interface can also have implemented methods along with abstract (unimplemented) methods. For Lambda expressions to work an interface should have one and only one abstract method.

If an interface has only one abstract method then it can act as a functional interface. But what if someone else adds another abstract method to the interface later without knowing that somewhere else in the code the interface was being used for lambda expressions. In order to avoid this we need to mark a functional interface with @@java.lang.FunctionalInterface.

Adding @FunctionalInterface prevent the interface from being modified. So if someone adds a new abstract method it will throw a compile time error. It is not mandatory but it conveys your intention that this interface is meant to be a functional interface.

In Java 8 there is a package called java.util.functions which contains ootb interfaces which can be used for common scenario like testing a condition.

## Method References

Are an alternative way writing lambda expressions which is very readable in certain scenarios.

## Streams

Streams are a sequence of elements supporting sequential and parallel aggregate operations.

Streams are like the conveyer belt/assembly line and then you have the actions performed on them while the iteration happens only once.

Every collection class comes with stream() which returns a sequential Streams.

Stream presents a new view of the source. It always has a source, which is a collection type.

Stream consists of three different elements:

1. Source - a backing collection
2. Intermediate Operations - that need to be performed on that stream
3. Terminal operation – End condition. This is basically the final action.