*Simple Lambdas:*

* *Java is an object oriented programming by heart. Functional programming is a way of writing code more declaratively. You specify what you want to do rather than dealing with the state of objects.*
* *Functional Programming uses lambda expressions to write code. A lambda expression is a block of code that gets passed around. It is like an unnamed method. It has parameters and a body just like full-fledged methods do, but it doesn’t have a name like real method.*
* *Lambda expressions are referred to as lambdas in short. Lambda expression is like a method that you can pass if it were a variable.*

*Example (Without Lambdas):*

* *Print out all elements in a list according to some criteria*

|  |
| --- |
| *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; }* |

*The Animal class has three instance variables which are set in the constructor. It has two methods that get the state of whether the animal can hop or swim. It also has a toString() method so we can easily identify the Animal in programs.*

* *We plan to write a lot of different checks, so we would need an interface.*

|  |
| --- |
| *public interface CheckTrait {*  *boolean test(Animal a);*  *}*  *public class CheckIfHopper implements CheckTrait {*  *public boolean test(Animal a){*  *return a.canHop();*  *}}*  *Import java.util.\*;*  *public class TraditionalSearch{*  *public static void main(String args[]){*  *List<Animal> animals = new ArrayList<Animal>();*  *animals.add(new Animal(“fish”,false,true));*  *animals.add(new Animal(“kangaroo”,true,false));*  *animals.add(new Animal(“rabbit”, true, false));*  *animals.add(new Anima(“turtle”,false,true));*  *print(animals, new CheckIfHopper());*  *}*  *private static void print(List<Animal> animals, CheckTrait checker){*  *for(Animal animal: animals) {*  *if(checker.test(animal))*  *System.out.print(animal + “ “);*  *}*  *System.out.println();*  *}*  *}* |

*Now what if we want to print the Animals that swim? We need to write another class. Turns out that we can with lambda expressions repeat the whole class here and make you find the one line that changed.*

*print(animals, a-> a.canHop());*

*We are telling Java that we care only about Animals that can hop.*

* *Lambdas uses the concept called “Deferred execution”. Deferred execution means that the code is specified now but will run later. In this case later is when print is called.*

*Lambda Syntax:*

* *One of the simplest Lambda expressions you can write is a -> a.canHop();*
* *Lambdas work with interfaces that only have one abstract method. The lambda indicates that Java should call a method called Animal parameter that returns a boolean value.*
* *Java relies on the context when figuring out what lambda expressions mean. We are passing this lambda as the second parameter of the print() method. The method expects a CheckTrait as the second parameter. Since we are passing a lambda instead, Java tried to map our lambda to that interface.*

*boolean test(Animal a);*

* *Since that interface’s method takes an Animal, that means the lambda parameter has to be an Animal. And since that interface’s method returns a boolean, we know that the lambda returns boolean.*
* *The syntax of lambdas is tricky because many parts are optional. These two lines can do the same thing.*

|  |
| --- |
| *a -> a.canHop()*  *(Animal a) -> { return a.canHop(); }* |

* *A single parameter specified with the name a*

*The arrow operator to separate the parameter and body*

*A body that calls a single method and returns the result of that method.*

* *The second example shows the most verbose form of a lambda that returns a boolean*

*A single parameter specified with the name a and stating the type is Animal*

*The arrow operator to separate the parameter and body*

*A body that has one or more lines of code including a semicolon and a return statement.*

*These parentheses can be omitted only if there is a single parameter and its type isn’t explicitly stated. Java does this because developers commonly use lambda expressions this way and they can do a little typing as possible.*

*Java doesn’t require you to type return or use a semicolon when no braces are used. This special shortcut doesn’t work when we have two or more statements. At least this is consistent with using {} to create blocks of code elsewhere.*

|  |  |
| --- | --- |
| *Lambda* | *Parameters* |
| *() -> true* | *0* |
| *a -> a.startsWith(“test”)* | *1* |
| *(String a ) -> a.startsWith(“test”)* | *1* |
| *(a,b) -> a.startsWith(“test”)* | *2* |
| *(String a, String b) -> a.startsWith(“test”)* | *2* |

* *Invalid Lambdas that return boolean*

|  |  |
| --- | --- |
| *Invalid Lambda* | *Reason* |
| *a,b* | *Missing Parentheses* |
| *a -> {a.startsWith(“test”)}* | *Missing return* |
| *a -> { return a.startsWith(“test”)}* | *Missing semicolon* |

***Introducing Functional Interfaces:***

* *Lambdas work with interfaces that have only one abstract method. These are called functional interfaces. This is known as “Single Abstract Method(SAM)” rule.*
* *Java provides an annotation @FuntionalInterface on some but not all functional interfaces. However, just because you don’t see the annotation doesn’t mean it’s not a functional interface. Remember that having exactly one abstract method is what makes it a functional interface, not the annotation.*
* *There are four functional interfaces – Predicate, Consumer, Supplier, Comparator*

*Predicate:*

* *We would have created lot of interfaces like this to use lambdas. Java recognizes that this is a common problem and provides such an interface for us. It is the package java.util.function and the gist of it as follows:*

|  |
| --- |
| *Public interface Predicate<T>{*  *boolean Test(T t);*  *}* |

* *This means we don’t need to put our own interface anymore and can put everything related to our search in one class:*

|  |
| --- |
| *Import java.util.\*;*  *Import java.util.function.\*;*  *public class PredicateSearch {*  *public static void main(String args[]) {*  *List<Animal> animals = new ArrayList<Animal>();*  *animals.add(new Animal(“fish”,false,true);*  *print(animals, a-> a.canHop());*  *}*  *private static void print(List<Animal> animals, Predicate<Animal> checker){*  *for(Animal animal: animals){*  *if(checker.test(animal)){*  *System.out.print(animal+ “ “);*  *}*  *System.out.println();*  *}*  *}* |

***Consumer:***

* *The consumer functional interface has one method*

*void accept(T t)*

*Consumer<String> consumer = x -> System.out.println(x);*

*We have a declared functionality to print out the value we were given. Its okay that we don’t have a value yet. When the consumer is called, the value will be provided and printed then.*

|  |
| --- |
| *public static void main(String[] args) {*  *Consumer<String> consumer = x -> System.out.println(x);*  *print(consumer, “Hello World”);*  *}*  *private static void print(Consumer<String> consumer, String value){*  *consumer.accept(value);*  *}* |

* *The print() method accepts a Consumer that knows how to print a value. When the accept() method is called, the lambda actually runs, printing the value.*

***Supplier:***

* *The supplier functional interface has only one method:*

|  |
| --- |
| *T get()* |

*A good use case for supplier is when generating values.*

|  |
| --- |
| *Supplier<Integer> number = () -> 42;*  *Supplier<Integer> random = () -> new Random().nextInt();* |

* *The first example returns 42 each time the lambda is called. The second example generates a random number each time it is called. It could be the same number but it is likely a different one.*

|  |
| --- |
| *Public static void main(String args[]){*  *Supplier<Integer> number = () -> 42;*  *System.out.println(returnNumber(number));*  *}*  *private static int returnNumber(Supplier<Integer> supplier){*  *return supplier.get(); }* |

* *When the returnNumber() method is called, it invokes the lambda to get the desired value. In this case, method returns 42;*

***Comparator:***

* *We compared numbers we dint supply a Comparator because we were using default sort order. A negative number means the first number is smaller, zero means they are equal and a positive number means the first value is bigger.*

|  |
| --- |
| *int compare(T o1, T o2)* |

* *This interface is a functional interface since it has only one unimplemented method. It has many static and default methods to facilitate writing complex comparators.*
* *The comparator interface existed prior to lambdas being added to Java. As a result, it is in different package. You can find comparator in java.util*

|  |
| --- |
| *Comparator<Integer> ints = (i1, i2) -> i1 – i2;* |

* *The ints comparator only uses natural sort order. If the first number is bigger, it will return positive number and we are sorting in ascending order.*

|  |
| --- |
| *Comparator<String> strings = (s1,s2) -> s2.compareTo(s1);*  *Comparator<String> moreStrings = (s1, s2) -> -s1.compareTo(s2);* |

* *Both of these comparators actually do the same thing sort in descending order. In the first example, the call to compareTo() is backwards making it descending. In the second example, the call uses the default order however it applies a negative sign to the result which reverses it.*

|  |  |  |
| --- | --- | --- |
| *Functional Interface* | *#Parameters* | *Return type* |
| *Comparator* | *Two* | *Int* |
| *Consumer* | *One* | *Void* |
| *Predicate* | *One* | *boolean* |
| *Supplier* | *None* | *One(type varies)* |

*Working with Variables in Lambdas:*

* *Variables can appear in three places with respect to lambdas: the parameter list, its local variables declared inside the lambda body and variables referenced from the lambda body.*

*Parameter List:*

* *Var can be used in place of specific type. That means all three of the below are interchangeable*

|  |
| --- |
| *Predicate<String> p = x-> true*  *Predicate<String> p = (var x) -> true*  *Predicate<String> p = (String x) -> true* |

*A lambda infers the types from the surrounding context. That means you get to do the same. In this case, the lambda is being assigned to a Predicate that takes the String. The type here is String.*

|  |
| --- |
| *public void whatAmI(){*  *consume((var x) -> System.out.print(x), 123);*  *}*  *Public void consume(Consumer<Integer> c, int num){*  *c.accept(num); }* |

* *The type of x here is Integer. The method creates a lambda to be passed to consume() method expects an Integer as the generic.*

|  |
| --- |
| *public void counts(List<Integer> list){*  *list.sort((var x, var y) -> x.compareTo(y));*  *}* |

*The answer is again an Integer. Since we are sorting a list, we can use the type of the list to determine the type of the lambda parameter.*

***Local Variables inside the Lambda body:***

* *While it is common for a lambda body to be a single expression, it is legal to define a block. The block can have anything that is valid in normal Java block, including local variables declarations.*
* *The below code creates a local variable named c that is scoped to the lambda block*

*(a,b) -> { int c=0; return 5;}*

* *When writing your own code, a lambda block with a local variable is a good hint that you should extract that code into a method.*

*(a,b) -> {int a=0; return 5;} //Doesn’t compile*

*We tried to redeclare a, which isn’t allowed. Java doesn’t allow you to create a local variable with the same name as one already declared in that scope.*

|  |
| --- |
| *public void variables( int a) {*  *Int b=1;*  *Predicate<Integer> p1 = a->{*  *int b=0;*  *int c=0;*  *return b ==c;*  *}*  *}* |

*There are three syntax errors. The variable a was already used in this scope as a method parameter so it cannot be reused. The next syntax error comes where the code attempts to redeclare local variable b. The third syntax error is quite subtle. The variable p1 is missing a semicolon at the end. There is a semicolon before the }, but that is inside the block.*

***Variables referenced from the Lambda body:***

* *Lambda bodies are allowed to reference some variables from the surrounding code. The following code is legal*

|  |
| --- |
| *public class Crow{*  *private String color;*  *public void caw(String name){*  *String volume =”loudly”;*  *Consumer<String> consumer = s -> System.out.println(name+ “says” + volume +” that she is “+color);*  *}*  *}* |

*This shows that lambda can access an instance variable, method parameter, or local variable under certain conditions. Instance variables and class variables are always allowed.*

* *Method parameters and local variables are allowed to referenced if they are effectively final. This means that the value of a variable doesn’t change after it is set, regardless of whether it is explicitly marked as final.*
* *If you aren’t sure whether a variable is effectively final, add the final keyword. If the code would still compile, the variables is effectively final.*

|  |
| --- |
| *public class Crow{*  *private String color;*  *public void caw(final String name){*  *final String volume =”loudly”;*  *Consumer<String> consumer = s-> System.out.println(name+ “says” +volume+ “ that she is “+color);*  *}*  *}* |

* *It gets even more interesting when you look at where the compile errors occur when the variables are not effectively final.*

|  |
| --- |
| *public class Crow{*  *private String color;*  *public void caw(String name){*  *String volume =”loudly”;*  *name=”Caty”;*  *Color=”black”;*  *Consumer<String> consumer = s-> System.out.println(name+ “says” +volume+ “ that she is “+color);*  *Volume=”softly”;*  *}*  *}* |

*In the above example, name isn’t effectively final because it is set on line 5. However, the compiler error occurs on line 9. Its not a problem to ressign variable value to a non final variable. However, once the lambda tries to use it, we do have a problem. The variable is no longer effectively final, so the lambda is not allowed to use the variable.*

* *The variable volume is not effectively final either since it is updated. That’s before the assignment. Again the act of assigning a value is only a problem from the point of view of the lambda. Therefore, the lambda has to be the one to generate the compiler error.*

***Rules for accessing a variable from a lambda body inside a method:***

|  |  |
| --- | --- |
| *Variable Type* | *Rule* |
| *Instance Variable* | *Allowed* |
| *Static variable* | *Allowed* |
| *Local variable* | *Allowed if effectively final* |
| *Method Parameter* | *Allowed if effectively final* |
| *Lambda Parameter* | *Allowed* |

***Calling API’s with Lambdas:***

***REMOVEIF():***

* *List and Set declare a removeIf() method that takes a Predicate. Imagine we have a list of names for pet bunnies. We decide we want to remove all of the bunny names that don’t begin with the letter h.*

|  |
| --- |
| *List<String> bunnies = new ArrayList<>();*  *bunnies.add(“long ear”);*  *bunnies.add(“floppy”);*  *bunnies.add(“hoppy”);*  *System.out.println(bunnies);*  *bunnies.removeIf(s -> s.charAt(0) != ‘h’);*  *System.out.println(bunnies);* |

*It defines a predicate that takes a String and returns a boolean. The removeIf() method does the rest. The removeIf() method works the same way on a Set. It removed any values in the set that match the Predicate. There isn’t a removeIf() method on a Map. Remember that maps both have keys and values. It wouldn’t be clear which one was removing.*

***SORT():***

* *When you call Collections.sort(list) you can now sort directly on the list object.*

|  |
| --- |
| *List<String> bunnies = new ArrayList<>();*  *bunnies.add(“long ear”);*  *bunnies.add(“floppy”);*  *bunnies.add(“hoppy”);*  *System.out.println(bunnies);*  *bunnies.sort((b1,b2) -> b1.compareTo(b2));*  *System.out.println(bunnies);* |

*The sort() method takes Comparator that provides the sort order. Remember that Comparator takes two parameters and returns an int. There is not a sort method on Set or Map. Neither of those types has indexing, so it wouldn’t make sense to sort them.*

***FOREACH()***

* *Our final method is forEach(). It takes a Consumer and calls that lambda for each element encountered.*

|  |
| --- |
| *List<String> bunnies = new ArrayList<>();*  *bunnies.add("long ear");*  *bunnies.add("floppy");*  *bunnies.add("hoppy");*  *bunnies.forEach(b -> System.out.println(b));*  *System.out.println(bunnies);* |

* *We can use forEach() with a Set or Map. For a Set, it works the same way as List.*

|  |
| --- |
| *Set<String> bunnies = Set.of(“long ear”, “floppy”,”hoppy”);*  *bunnies.forEach(b -> System.out.println(b));* |

* *For a Map, you have to choose whether you want to go through the keys or values*

|  |
| --- |
| *Map<String, Integer> bunnies = new HashMap<>();*  *bunnies.put("long ear", 3);*  *bunnies.put("floppy", 8);*  *bunnies.put("hoppy", 1);*  *bunnies.keySet().forEach(b -> System.out.println(b));*  *bunnies.values().forEach(b -> System.out.println(b));* |

* *It turns out the keyset() and values() methods each return a Set.*

***Using forEach() with a Map directly***

* *Java has a functional interface called BiConsumer. It works just like Consumer except it takes two parameters. This functional interface allows you to use forEach() with key/value pairs from Map.*

|  |
| --- |
| *Map<String, Integer> bunnies = new HashMap<>();*  *bunnies.put("long ear", 3);*  *bunnies.put("floppy", 8);*  *bunnies.put("hoppy", 1);*  *bunnies.forEach((k, v) -> System.out.println(k + " "*  *+ v));* |

***Summary:***

* *Lambda expressions, or lambdas, allow passing around blocks of code. The full syntax looks like this:*

*(String a, String b) -> { return a.equals(b); }*

* *The parameter types can be omitted. When only one parameter is specified without a type the parentheses can also be omitted. The braces and return statement can be omitted for a single statement, making the short form as follows:*

*a -> a.equals(b)*

* *Lambdas are passed to a method expecting an instance of a functional interface. A functional interface is one with a single abstract method. Predicate is a common interface that returns a boolean and takes any type. Consumer takes any type and doesn’t return a value. Supplier returns a value and does not take any parameters. Comparator takes two parameters and returns an int.*
* *A lambda can define parameters or variables in the body as long as their names are different from existing local variables.*
* *The body of a lambda is allowed to use any instance or class variables. Additionally, it can use any local variables or method parameters that are effectively final. We covered three common APIs that use lambdas. The removeIf() method on a List and a Set takes a Predicate. The sort() method on a List interface takes a Comparator. The forEach() methods on a List and a Set interface both take a Consumer.*