**Java 8**

**Anonymous Class & Lambda Expression**

Say you have an interface (abstract class bhi chalega) say “Par”, we implement it and create a new class say “ChildC” where we override all members from “Par”. In main function we create an instance of ChildC and access all members.

This is usually what we do.

By using Anonymous Class we can ignore creating implemented class and directly override the members of “Par” at the time of Instantiaition and use it the very same way as above. Eg,

package com.company;  
  
interface Par{  
 abstract void meth1();  
}  
// Traditional way  
//class ChildC implements Par{  
// public void meth1(){  
// System.out.println("I'm Meth1");  
// }  
//}  
public class AnonymousClass {  
 public static void main(String[] args) {  
 // Traditional way  
// ChildC obj = new ChildC();  
// obj.meth1();

// Anonymous Class   
 Par obj = new Par() {  
 @Override  
 void meth1() {  
 System.*out*.println("I'm Meth1");  
 }  
 }

obj.meth1();

// Lambda Expression

Par obj1 = ()->{System.*out*.println("I'm Meth1 using Lambda Expression");}; // for a single line we can ignore {}  
obj1.meth1();

// Suppose khuch parameters pass krna hota to

Par obj1 = (a)-> System.*out*.println("I'm Meth1 using Lambda Expression " + a); // No need to mention datatpe of argument Lambda khud se dekh leta h data type  
obj1.meth1(5);

}  
}

Basicay by using Anonymous Class we can avoid creating Child class. (Though not recommended)

**Note: Lambda Expression is only meant for Functional Interface (pg 41) while Anonymous class is for both Abstract class and interface.**

Basically, both is used to make code compact and concise (thode me bahut info)

**Functional Interface**(interface with only 1 abstract method)

Functional interfaces can do lot more than you think.

1. **Consumer Interface:**

Represents an operation that accepts input argument/arguments and returns nothing. Unlike most other functional interfaces.

**Abstract method: accept(T paraName)**. Jaisa Consumer interface ka reference type vaisa hi data type input leta h.

**Eg 1,**

Consumer<String> consumer = (s) -> System.*out*.println(s.toUpperCase()); // Consumer reference type String h to accept ko bhi string pass karenge.   
consumer.accept("pAppu");

**Eg 2,**

Consider a class Student with setter,getter & toString() of Student details.

Consider StudentDataBase Class with getAllStudents method which returns List of student details. (List contains details in object form).

package com.learnjava8.functionalinterfaces;  
// All necessary imports  
public class ConsumerInterfaceEg {

public static void printStudentDetail(){  
 Consumer<Student> c2 = (stu) -> System.*out*.println(stu); // here mentioned type is Class to phir accept abstract method bhi object hi lega. Therefore, here stu is a object  
 List<Student> studentList = StudentDataBase.*getAllStudents*();  
 studentList.forEach(c2);

// Let’s see what happens inside forEach Loop upar upar se

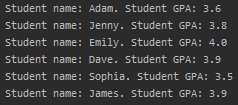
It accepts a Consumer interface or Lambda expression. If passed Consumer, forEach does this internally,

for(List studentList : li) c2.accept(li); // Obviously list end tk iterate karega.

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

**See what’s Inside Class Student and StudentDataBase:** [**https://github.com/dilipsundarraj1/java-8/tree/master/java-8/src/com/learnJava/data**](https://github.com/dilipsundarraj1/java-8/tree/master/java-8/src/com/learnJava/data)

**Consumer Chaining(**ek Consumer ko dusre consumer se jodhna)

In above printStudentDetail if we had done this Output below

Consumer<Student> c2 = (stu) -> System.*out*.print("Student name: " + stu.getName());  
Consumer<Student> c3 = (stu) -> System.*out*.println(". Student GPA: " + stu.getGpa());  
List<Student> studentList = StudentDataBase.*getAllStudents*();

studentList.forEach(c2.andThen(c3)); // Consumer Chaining

**Using Condition in Consumer**

**Note: Agr forEach me consumer pass kare to “accept” automically call ho jaata h but if we pass lamda expression we have to expilcitly call accept**.

In above printStudentDetail if we had done this

Consumer<Student> *c2* = (stu) -> System.*out*.print("Student name: " + stu.getName());  
Consumer<Student> *c3* = (stu) -> System.*out*.println(". Student GPA: " + stu.getGpa());

*studentList*.forEach(stu -> {  
 if(stu.getGpa() > 3.6) *c2*.andThen(*c3*).accept(stu);   
 });  
}

**Method Reference**

**It is used to simply the code written inside {} of Lambda Expression**.

**Syntax :**

**ClassName::methodName**

Ex,

// Using Lambda Expression  
Function<String,String> usingLambda = word -> word.toUpperCase();  
  
// Using Method Reference  
**Function<String,String> usingMethodReference = String::toUpperCase;**

// In Function Interface  
// Using Lambda Expression  
 Function<String,String> usingLambda = word -> word.toUpperCase();  
 System.*out*.println(usingLambda.apply("AadaPadaKonPada"));  
  
 // Using Method Reference  
 **Function<String,String> usingMethodReference = String::toUpperCase;**  
 System.*out*.println(usingMethodReference.apply("bantiWedsBabli"));  
  
 // In Consumer Interface  
 List<Student> studentList = StudentDataBase.*getAllStudents*();  
 // Using Lambda Expression  
 Consumer<Student> usingLambda1 = curStudent -> System.*out*.println(curStudent);  
 studentList.forEach(usingLambda1);  
  
 // Using Method Reference  
 Consumer<Student> usingMethodReference1 = System.*out*::println;  
 studentList.forEach(usingMethodReference1);  
  
 Consumer<Student> usingLambda2 = curStudent -> curStudent.printActivities();  
 studentList.forEach(usingLambda2);  
  
 Consumer<Student> usingMethodReference2 = Student::printActivities;  
 studentList.forEach(usingMethodReference2);

**Implementing Method Reference where it cannot be directly implemented**

public class MethodReferenceIndirectImplementation {  
 // Implementing Method Reference where it cannot be directly implemented  
 static Supplier<Student> *studentSupplier* = () -> new Student("bimlesh",3,4.4,"male", Arrays.*asList*("kirket", "tebal tanis","fud"));  
  
  
 // We cannot directly create method reference for Predicate Interface. But we can create boolean method and use it.  
 // Using Lambda Expression  
 static Predicate<Student> *usingLamda* = student -> student.getGpa() > 3;  
  
 // Using Method Reference Expression  
 static boolean gpaGreater3 (Student s){  
 return s.getGpa() > 3;  
 }  
 static Predicate<Student> *usingMethodReference* = MethodReferenceIndirectImplementation::*gpaGreater3*;  
  
 public static void main(String[] args) {  
 System.*out*.println(*usingLamda*.test(*studentSupplier*.get()));  
 System.*out*.println(*usingMethodReference*.test(*studentSupplier*.get()));  
 }  
}

**Note: It’s not necessary that every code must be compatible with Method Reference. Bss ye aur short code likhne ke liye bana h.**

**Constructor Reference**

**It is used to create a new object of a class but only using Functional Interface. Bss itna hi.**

**Syntax:**

**ClassName::new**

Eg,

Supplier<Student> newStudent = Student::new;  
// Note upar koi input parameter nhi h. Therefore, we must have a Default Constructor in that mentioned class. (here in Student Class).

System.*out*.println(newStudent.get());  
  
Function<String,Student> newStu = Student::new; // Created a new object which takes  
// Note upar ek input parameter h. Therefore, we must have a parameterized constructor with one parameter in that mentioned class (here in Student Class).

System.*out*.println(newStu.apply("champak"));



Output ->

**Limitation using Lamda Expression**

1. It’s not allowed to use the same local variable name as a lambda parameter or insidethe lambda body. Eg,

public static void main(String[] args) {  
 int i = 5;  
 Consumer<Integer> cs = i -> System.*out*.println(i); // Error -> java: variable i is already defined in method main  
 cs.accept(34);  
}

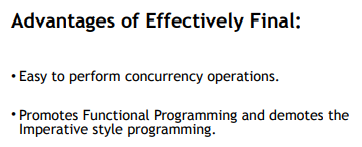
1. It’s not allowed to reassign a new value to a local variable in the lambda expression. Eg,

public static void main(String[] args) {  
 int i = 5;  
 Consumer<Integer> cs = j -> {  
 i =10; // Error dega  
 System.*out*.println(j);  
 };

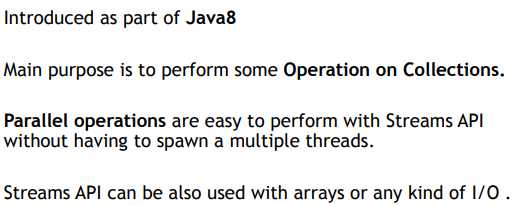
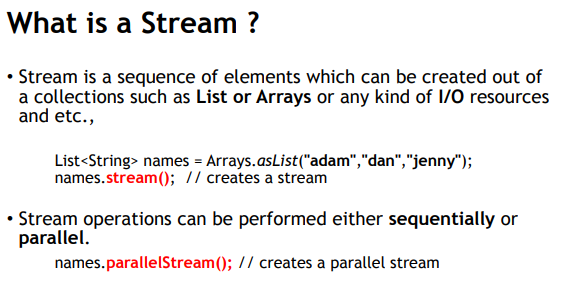
.

But we can reassign Reference Variable in Lambda Body.

public class LamdaRestriction {  
 static int *z* = 6;  
 public static void main(String[] args) {  
 Consumer<Integer> c = x -> {  
 *z*++; // chalega  
 System.*out*.println(i);  
 };  
 }  
}

****

**Effectively Final:** Even though the local variables are not declared final, they are still not allowed to be modified in the Lamda Body. Bss issi ko bolte Effectively Final

**Streams API**

**Terminology in Stream API**

**Stream Pipeline:** All the steps right from creating stream till Terminal Operation.

**Terminal Operation:** A terminal operation in Java 8 is a method that is applied to a stream as the final step. Terminal operations return a result or produce a side effect.

**Note:**  Termina Operation are the last operation in a stream pipeline and cannot be followed by any other operation. The result of a terminal operation can be a single value, a collection, or some other type of object.

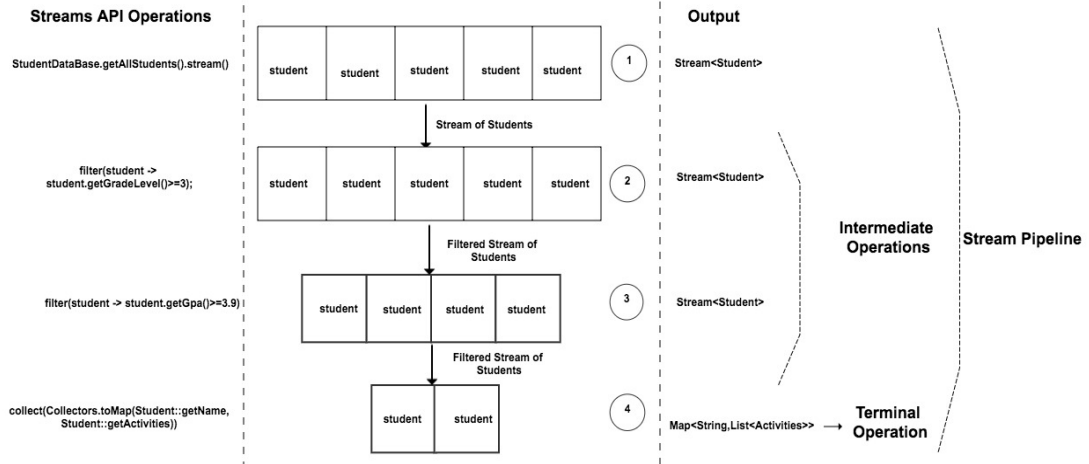
**Intermediate Operations:** All Operations performed after creating Stream and before Terminal Operation (like collect).

**Working of Stream API**

Predicate<Student> gradeGreaterEq3 = student -> student.getGradeLevel() >= 3;  
Predicate<Student> gpaGreaterEq3Dec9 = student -> student.getGpa() >= 3.9;

**Map<String,List<String>> studentNameAndActivities = StudentDataBase.*getAllStudents*().stream()**  // creates stream **.filter(gradeGreaterEq3)**  // creates stream **.filter(gpaGreaterEq3Dec9)**  // creates stream **.collect(Collectors.*toMap*(Student::getName,Student::getActivities));**  // creates Map  
  
System.*out*.println(studentNameAndActivities);

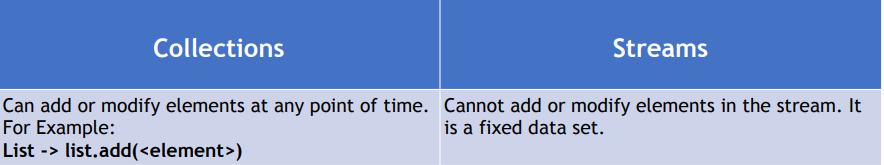
Let’s see the working using above code

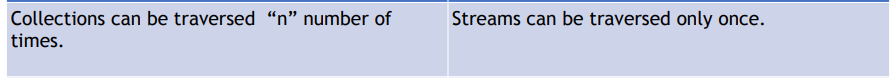


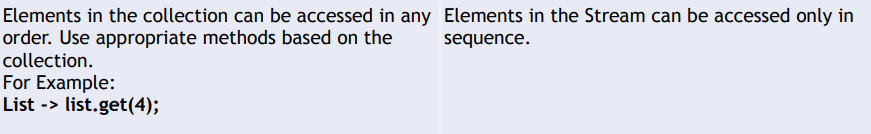
**Note:** Terminal Operation(TO) is the one which starts the complete process, meaning this is the one which starts the pipeline. If the TO method wasn't there, then there is no way this pipeline would have started and it wouldn't have created the output. All of the stream operation above won't happen at all.

**Inshort agar me collect comment krdu to khuch hoga hi nhi (stream bhi create nhi hoga).**

**Note:** Stream API uses declarative way of programming where we only focus on getting the desired output using inbuilt methods and classes without caring what those methods and classes are doing behind the scene.

**Difference b/w Collection and Stream**

****Stream me khuch changes nhi kr sakte. Stream ke help se khuch krna ho to vo kr sakte h

****

List<String> studentName = Arrays.*asList*("Titu","Bitu","Pittu","Gittu");  
Stream<String> nameStream = studentName.stream(); // Created Stream

nameStream.forEach(System.*out*::println);  
nameStream.forEach(System.*out*::println); // will throw error-> stream has already been operated upon or closed

**Stream API Operations**

1. **map()**: converts one type to another.

Eg,

Set<String> studentName = StudentDataBase.*getAllStudents*().stream() // Created stream of Student  
 .map(Student::getName) // Converted Stream of Student to Stream of String  
 .collect(*toSet*()); // created Set.

System.*out*.println(studentName);

**Note:** map accepts Function Interface