7. Lambdas

- Lambdas are anonymous functions, which does not have any name.
- Lambda are not associated with any class. Lambdas are functions.
- They have a list of input parameters, body, return type and possible set of exceptions.
- Lambdas can be assigned to variables or passed as parameter.
- Lambdas are the basic building blocks of functional programming support in Java
- Lambdas along with immutable streams provides a powerful support of functional programming in Java.
- Java 8 provides ability to pass a piece of code as parameter to any method.
- Passing code in methods is currently not very friendly in Java, Labmda help us to make the code cleaner and more flexible.
- Labmdas encourage behaviour parameterization over value parameterization.

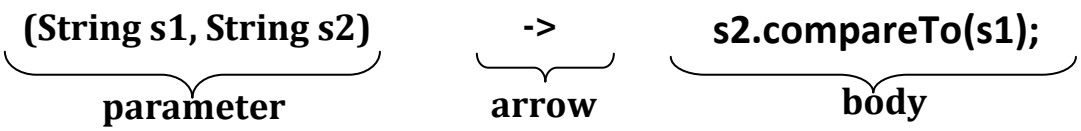
Java8Lab28.java

```
// import statements
public class Java8Lab28 {
	public static void main(String[] args) {
		List<String> names = Arrays.asList("Dande","Ranjan","Nivas","Sri","Manish");
		NameDescComparator comp = new NameDescComparator();
		Collections.sort(names, comp);
		System.out.println(names);
	}
}
class NameDescComparator implements Comparator<String>{
	@Override
	public int compare(String s1, String s2) {
		return s2.compareTo(s1);
	}
}
```

Java8Lab29.java

```
// import statements
public class Java8Lab29 {
	public static void main(String[] args) {
		List<String> names = Arrays.asList("Dande","Ranjan","Nivas","Sri","Manish");
		Comparator<String> comp = (String s1,String s2) -> s2.compareTo(s1);
		Collections.sort(names, comp);
		System.out.println(names);
	}
}
```

- Lambdas is made of parameters, an arrow and its body.



General Syntax:

(parameters) -> expression

or

(parameters) -> {statements;}

- Some valid and invalid lambdas.

```
() -> {}                                    // VALID
() -> "Manish"                              // VALID
() -> {return "Manish";}                    // VALID
(int a, int b) -> a+b                       // VALID
(a, b) -> a+b                               // VALID
(Double d) -> {return "Manish" + d;}// VALID
(Double d) -> return "Manish" + d;  // INVALID
(String s) -> {"Manish";}                   // INVALID
```

Usages	Example Code
Object Creation	() -> new Student(99)
Object Consumption	(Student s) -> {SOP (s);}
Boolean Expression	(List<String> list) -> list.isEmpty()
Combining values	(int a, int b) -> a+b
Comparing two objects	(String s1, String s2) -> s1.compareTo(s2)
Extracting vaues from object	(String s) -> s.length()

- Labmdas support Type Inference in Java.
- Type inference support available in Java from Java 7.
- Type inference is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable.

Java8Lab30.java

```
// import statements
public class Java8Lab30 {
public static void main(String[] args) {
List<Student> students = Arrays.asList(new Student(56, "Manish"), new Student(12, "Nivas"), new Student(23,
"Sri"), new Student(45, "Abhi"));
SidComparator comp = new SidComparator();
Collections.sort(students, comp);
for (Student student : students) {
 System.out.println(student);
}
} }
class SidComparator implements Comparator<Student> {
@Override
public int compare(Student s1, Student s2) {
return s1.getSid() - s2.getSid();
} }
class Student {
private int sid;
private String name;
Student(int sid, String name) {
this.sid = sid;
this.name = name;
}
// Setters and Getters
@Override
public String toString() {
return sid + "\t" + name;
}
}
```

Java8Lab31.java

```
// import statements
public class Java8Lab31 {
public static void main(String[] args) {
List<Student> students = Arrays.asList(new Student(56, "Manish"), new Student(12, "Nivas"), new Student(23,
"Sri"), new Student( 45, "Abhi"));
Comparator<Student> comp = (s1,s2) -> s1.getSid()-s2.getSid();
Collections.sort(students, comp);
for (Student student : students) {
System.out.println(student);
}
}
}
// Using Same Student class from Jaba8Lab30
```

Java8Lab32.java

```
// import statements
public class Java8Lab32 {
public static void main(String[] args) {
List<Student> students = Arrays.asList(new Student(56, "Manish"), new Student(12, "Nivas"), new Student(23,
"Sri"), new Student( 45, "Abhi"));
students.sort((s1,s2) -> s1.getSid()-s2.getSid());
students.sort((Student s1,Student s2) -> s1.getSid()-s2.getSid());
for (Student student : students) {
System.out.println(student);
} } }
// Using Same Student class from Jaba8Lab30
```

Java8Lab33.java

```
// import statements
import static java.util.Comparator.comparing;
public class Java8Lab33 {
public static void main(String[] args) {
List<Student> students = Arrays.asList(new Student(56, "Manish"), new Student(12, "Nivas"), new Student(23,
"Sri"), new Student( 45, "Abhi"));
Comparator<Student> comp = comparing(Student::getSid);
students.sort(comp);
students.sort(comparing(Student::getSid));
for (Student student : students) {
System.out.println(student);
} } }
// Using Same Student class from Jaba8Lab30
```

7.3. Functional Interface

- A *functional interface* is an interface that defines one and only one abstract method.
- Functional interfaces provide support for lambda expressions and method references.
- The major benefit of functional interface is we can use lambda expressions to instantiate them and avoid using anonymous class implementation.
- A new annotation @FunctionalInterface has introduced to mark an interface as Functional Interface
- Java has many functional interfaces in java.util.function package.
- Funational interface can have many default method or static method.

// INVALID
@FunctionalInterface
interface Inter2{ }
	// VALID
	@FunctionalInterface
	interface Inter3{
	void show();
	}
		// VALID
		@FunctionalInterface
		interface Inter4{
		void show();
		default void display(){ }
		}

// VALID
@FunctionalInterface
interface Inter4{
void show();
static void display(){ }
}
	// INVALID
	@FunctionalInterface
	interface Inter1 {
	void show();
	void display();
	}	

- Following are the example of functional interface that has single abstract method run()
  - java.lang.Runnable
  - java.util.concurrent.Callable
  - java.util.Comparator
  - java.util.function.Predicate
  - java.util.function.Consumer
  - java.util.function.Function
- Functional interfaces are important in Lambdas.
- Labmda expression allow programmers o provide the implementations of the abstract method of Functional interface.
- The in-line implementation of an abstract method is treated as an instance of a concrete implementation of the functional interface.

```
Runnable r = () -> System.out.println("Hello");
Comparator<String> comp = (s1,s2) -> s1.compareTo(s2);
```

7.4. Function Descriptor

- The signature of the abstract method in the Functional interface decides the signature of the lambda expression.
- The method is called as Function Descriptor.

  In Runnable Functiona interface run() method is the function descriptor.

  In Comparator Functiona interface compare() method is the function descriptor.

7.5. Working with Functional Interface

- Predicate functional interface defines only one abstract method **test** which accepts generic object and returns a Boolean value. This can be used to represent and function which returns a Boolean and excepts any generic object.
- Consumer functional interface defines only one abstract method **accepts** which accepts generic object and returns a void. This can be used to represent any function which returns void and excepts any generic object.
- Function functional interface defines only one abstract method **apply** which accepts generic object and returns another generic object. This can be used to represent any function which returns some object and return some object.

Java8Lab34.java

```
// import statements
public class Java8Lab34 {
public static void main(String[] args) {
List<String> names = Arrays.asList("Manish", "Nivas", "", "Sri","Abhi", null);
Predicate<String> nonEmptyString = (s) -> s != null && !s.isEmpty();
System.out.println(filter(names,nonEmptyString));

List<Integer> values = Arrays.asList(123, 0, 45, 3, 0, 23, 0, 34);
Predicate<Integer> nonZeroValues = (in) -> in != 0;
System.out.println(filter(values,nonZeroValues));
}
public static <T> List<T> filter(List<T> list, Predicate<T> predicate) {
List<T> results = new ArrayList<>();
for (T obj : list) {
if(predicate.test(obj))
results.add(obj);
}
return results;
}
}
```

Java8Lab35.java

```
// import statements
public class Java8Lab35 {
public static void main(String[] args) {
Consumer<Number> printNumConsumer = (Number val) -> System.out.println(val);
Consumer<String> printStringConsumer = (val) -> System.out.println(val);
List<String> names = Arrays.asList("Manish", "Nivas", "", "Sri","Abhi", null);
List<Number> values = Arrays.asList(123, 0, 45, 3, 0, 23, 0, 34);
printEach(names, printStringConsumer);
System.out.println();
printEach(values, printNumConsumer);
}
public static <T> void printEach(List<T> list, Consumer<T> consumer) {
for (T obj : list) {
consumer.accept(obj);
}
}
}
```

Java8Lab36.java

```
// import statements
public class Java8Lab36 {
public static void main(String[] args) {
Function<String, Integer> length = (s1) -> s1.length();
List<String> names = Arrays.asList("Sri","Kumar","Dande","Abhi");
System.out.println( findLength(names, length));
}
public static List<Integer> findLength(List<String> list,Function<String,Integer> function){
List<Integer> result = new ArrayList<>();
for (String value : list) {
result.add(function.apply(value));
}
return result;
}
}
```

- All of these interface we have seen are for generic types.
- Generic are good for generic objects, we often face the situation to deal with primitives in Java.
- Using the boxing and unboxing approach for primitives has performance issue for large data application.
- So, most of the functional interfaces have their primitive specialization which avoid the boxing/unboxing and these performs better while dealing with primitive data types.

Functional Interface	Function Descriptor	Specializations
Predicate<T>	T -> boolean	IntPredicate, LongPredicate, DoublePredicate
Consumer<T	T -> void	IntConsumer, LongConsumer, DoubleConsumer
Function<T,R>	T -> R	INtFunction<R>, IntToDoubleFunction, IntToLongFunction, ...
Supplier<T>	() -> T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T -> T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T , T) -> T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator

Java8Lab37.java

```
// import statements
public class Java8Lab37 {
public static void main(String[] args) {
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
UnaryOperator<Integer> operator = in -> in * 10;
List<Integer> results = testUnary(numbers, operator);
for (Integer val : results) {
System.out.println(val);
} }
public static List<Integer> testUnary(List<Integer> list, UnaryOperator<Integer> operator) {
List<Integer> results = new ArrayList<>();
for (Integer val : list) {
results.add(operator.apply(val));
}
return results;
} }
```

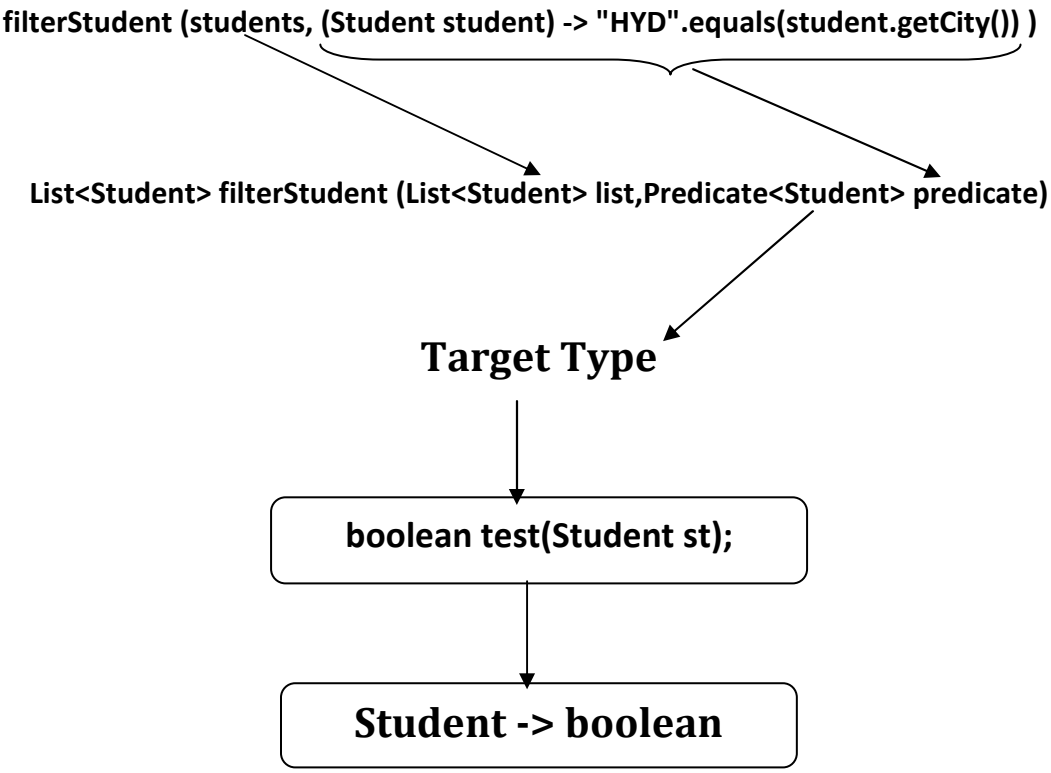
Java8Lab38.java

```
// import statements
public class Java8Lab38 {
public static void main(String[] args) {
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
List<Integer> results = testUnary(numbers, in -> in * 10);
results.forEach(in -> System.out.println(in));
System.out.println();
results = testUnary(numbers, (in) -> in + 10);
results.forEach(in -> System.out.println(in));
}
public static List<Integer> testUnary(List<Integer> list,UnaryOperator<Integer> operator) {
List<Integer> results = new ArrayList<>();
list.forEach(in -> results.add(operator.apply(in)));
return results;
} }
```

Java8Lab39.java

```
// import statements
public class Java8Lab39 {
public static void main(String[] args) {
BiFunction<Integer, Integer, String> biFunc = (a,b) -> ""+a* b;
String result = calculate(biFunc, 10, 20);      System.out.println(result);
System.out.println(calculate( (a,b) -> a+ b+"" ,  10,  20));
}
static String calculate(BiFunction<Integer, Integer, String> biFunc, Integer val1, Integer val2){
return biFunc.apply(val1, val2);
} }
```

7.6. Type Checking for Lambdas



- We need to look at the declaration of filterStudent() method.
- Filter method accepts second parameter as Predicate<Student>. It is target type.
- Predicate<Student> is a functional interface.
- The abstract method **test** describes a function descriptor which accepts Student and returns an boolean.
- Any Lambda following the same signature will be valid one.
- If the function descriptor has thrown any exception, lambdas also need to handle them.

```
Object run = () -> System.out.println("Hello Manish");
```

Here problem is target type. Object is not a functional interface. So when we change the target type to Runnable, which has the same signature function descriptor then problem will be solved.

```
Runnable run = () -> System.out.println("Hello Manish");
```

Type checking of lambdas is done based on its target type.

- Java compiler takes advantages of Type checking based on target type of lambda and can estimate the signature of a lambda based on the target type.
- So below are valid lambda:

```
Comparator<String> comp = (val1,val2) -> val1.compareTo(val2);
```

is same as

```
Comparator<String> comp = (String val1, String val2)  -> val1.compareTo(val2);

Comparator<Integer> comp2 = (val1,val2) -> val1.compareTo(val2);
```

is same as

```
Comparator<Integer> comp3 = (Integer val1, Integer val2)  -> val1.compareTo(val2);

// Comparator<Integer> comp3 = (String val1, String val2)  -> val1.compareTo(val2);
```

Below are the same lambdas representation:

```
UnaryOperator<Integer> operator = in -> in * 10;
UnaryOperator<Integer> operator = (in) -> in * 10;
UnaryOperator<Integer> operator = (Integer in) -> in * 10;
```

7.7. Method and Consutuctor Reference

- In Java 8 we can refer to existing methods as lambdas.
- A method reference is the shorthand syntax for a lambda expression that executes just ONE method.
- It makes the code more readable.
- Lambdas are the actual representation of the method.
- Method references allow the creation of lambda expression from any existing method.
- We do not need the brackets for the method. It does not call the method.
- Syntax:

  <Class> :: <method Name>
- Method references can be done in any of the following cases:

Kind	Example
Reference to a static method	ContainingClass::staticMethodName
Reference to an instance method of a particular object	containingObject::instanceMethodName
Reference to an instance method of an arbitrary object of a particular type	ContainingType::methodName
Reference to a constructor	ClassName::new

Java8Lab40.java

```
// import statements
public class Java8Lab40 {
public static void main(String[] args) {
Function<String, Integer> stringConversion = (String st) -> Integer.parseInt(st);
Function<String, Integer> stringConversion1 = Integer::parseInt;
Integer in1=transform(stringConversion, "123");
Integer in2=transform(Integer::parseInt, "123");
System.out.println(in1);
System.out.println(in2);
}
static Integer transform(Function<String, Integer> function,String str){
return function.apply(str);
}
}
```

Java8Lab41.java

```
// import statements
public class Java8Lab41 {
public static void main(String[] args) {
List<Student> students = Arrays.asList(new Student(56, "Manish"), new Student(12, "Nivas"), new Student(23,
"Sri"), new Student(45, "Abhi"));
students.sort((st1,st2) -> Student.compareBySid(st1, st2));
students.forEach(st -> System.out.println(st));
System.out.println();
students.sort(Student::compareBySidDesc);
students.forEach(st -> System.out.println(st));
Comparator<Student> comp=Student::compareBySid;
System.out.println(comp);
System.out.println();
students.sort(Comparator.comparing(Student::getName));
students.forEach(st -> System.out.println(st));
}
}
class Student {
private int sid;
private String name;
public Student(int sid, String name) {
this.sid = sid;
this.name = name;
}
// Getter and Setter methods
public static int compareBySid(Student st1, Student st2) {
return st1.getSid()-st2.getSid();
}
```

```
public static int compareBySidDesc(Student st1, Student st2) {
return -st1.getSid()-st2.getSid();
}
@Override
public String toString() {
return sid+"\t"+name;
}
}
```

Java8Lab42.java

```
// import statements
public class Java8Lab42 {
public static void main(String[] args) {
List<Student> students = Arrays.asList(new Student(56, "Manish"), new Student(12, "Nivas"), new Student(23,
"Sri"), new Student(45, "Abhi"));
ComparisonProvider provider = new ComparisonProvider();
students.sort((st1,st2) -> provider.compareBySid(st1, st2));
students.forEach(st -> System.out.println(st));
System.out.println();
students.sort(provider::compareBySidDesc);
students.forEach(st -> System.out.println(st));
}
}

class Student {
private int sid;
private String name;
public Student(int sid, String name) {
this.sid = sid;
this.name = name;
}
// Getter and Setter method
@Override
public String toString() {
return sid+"\t"+name;
}
}
class ComparisonProvider {
public  int compareBySid(Student st1, Student st2) {
return st1.getSid()-st2.getSid();
}
public  int compareBySidDesc(Student st1, Student st2) {
return -st1.getSid()-st2.getSid();
}
}
```

Java8Lab43.java

```
// import statements
public class Java8Lab43 {
public static void main(String[] args) {
List<String> names = Arrays.asList( "Manish", "Nivas", "Sri", "Abhi");
Comparator<String> com = String::compareTo;
names.sort(String::compareTo);
System.out.println(names);
Consumer<String> cons = System.out::println;
names.forEach(cons);
System.out.println();
names.forEach(System.out::println);
}
}
```

- The equivalent lambda expression for the method reference String::compareTo would have the formal parameter list (String a, String b).
- The method reference would invoke the method a.compareTo(b).

7.8. Constructor Reference

- We can refer to existing constructors just like method references using <ClassName>:: new.
- It is almost similar to static method reference.
- There is a functional interface java.util.function.Supplier. It has a get() method that returns object of specified type using generic.

Java8Lab44.java

```
// import statements
public class Java8Lab44 {
public static void main(String[] args) {
Supplier<Student> s1 = () -> new Student();
Student st1 = s1.get();
System.out.println(st1);
Supplier<Student> s2 = Student::new;
Student st2 = s2.get();
System.out.println(st2);
}
}
class Student { }
```

- The constructr of your class may requires argument. In such case
  - You need to identify any functional interface that has a method to accept the required parameter and returnd some object.
  - When no built-in functional interface available then you need to define the custom functional interface. This interface should have one method with the required argument as parameter and returns the object of required type.

Java8Lab45.java

```
// import statements
public class Java8Lab45 {
public static void main(String[] args) {
// Supplier<Student> sp1 = Student::new;
CustomInterface<Student> cust = Student::new;
Student st = cust.get(12, "Manish");
System.out.println(st);
CustomInterface<Employee> cust1 = Employee::new;
Employee emp = cust1.get(99, "Ranjan");
System.out.println(emp);
}
}
interface CustomInterface<T> {
T get(int value, String data);
}
class Student {
int sid;
String sname;
Student(int sid, String sname) {
this.sid = sid;
this.sname = sname;
}
@Override
public String toString() {
return "Sid :" + sid + "\tSname :" + sname;
}
}
class Employee {
int eid;
String ename;
Employee(int eid, String ename) {
this.eid = eid;
this.ename = ename;
}
@Override
public String toString() {
return "Eid :" + eid + "\tEname :" + ename;
} }
```

Java8Lab46.java

```
// import statements
public class Java8Lab46 {
public static void main(String[] args) {
BiFunction<Integer,String,Student> func = Student::new;
Student st = func.apply(12, "Manish");
System.out.println(st);
BiFunction<Integer,String,Employee> func2 = Employee::new;
Employee emp = func2.apply(99, "Ranjan");
System.out.println(emp);
}
}
// Student & Employee Class from Previous Lab
```

8. Stream API

- Java Collection is good for storing the data but their manipulation is difficult.
- We have Iterator for performing the operation like grouping, highest marks student, A grade students etc with collection.
- RDBMS allows us to perform the same operation in a declarative fashion using SQL.
- Stream is a sequence of objects or primitive types. Operations can be performed on the elements sequentially or in parallel.
- Java 8 introduces Stream API in java.utils.stream package. It is introduced to handle and manipulate the collections in a declarative manner.
- Stream API provides many in-built methods to form any pipeline of transformations.
- Stream API handles the parallelism behind the scene in case of multi core processor.
- It provides parallelism virtually free as compared to using the threads.
- A stream is created from a source that can be an array, a collection, IO channel etc.
- Streams have a lifecycle which consists of creation, intermediate operations and terminal operation.
- Streams Operations can be classified into two categories:
    - Intermediate Operations returns another stream as return type
        - These are lazy. These will not do any processing till a terminal operation is invoked
    - Terminal Operations produces the result from the input stream.
        - These output are non-stream type i.e List,Integer, Long, String etc
- Streams can be created from
    - Collections using the default Stream<E> stream() method in the Collection interface
    - Arrays use the Arrays.stream(T[] array) method.
- Once you create a stream you can perform intermediate operations on it.
    - Stream<T> filter(Predicate<? super T> predicate)
        - This operation keeps those elements in the stream that match the predicate.
    - Stream<T> distinct()

- The last step in the lifecyle is called a terminal operation. The stream no longer exists after the terminal operation. The terminal operation can be a count operation or a collect operation.
- There are certain operations that are called short-circuit terminal operations.
    - boolean allMatch(Predicate<? super T> predicate)
        - Returns true if all values in the Collection return true for the passed lambda expression (predicate)
    - boolean anyMatch(Predicate<? super T> predicate)
        - Returns true if any value in the Collection returns true for the passed lambda expression (predicate). All elements may not need to be analysed.
    - void forEach(Consumer<? super T> action)
        - Iterates over each element of the List and calls the lambda expression specified by 'action'.
    - boolean removeIf(Predicate<? super E> filter)
        - Iterates through the Collection and removes the element that matches the filter.
    - void replaceAll(UnaryOperator<String> operator)
        - Replaces each element of this list with the result of applying the operator to that element.
    - void sort(Comparator<? super E> c)
        - Sorts the element using the provided comparator.
- **Streams are one time traversable just lilke Iterator.**

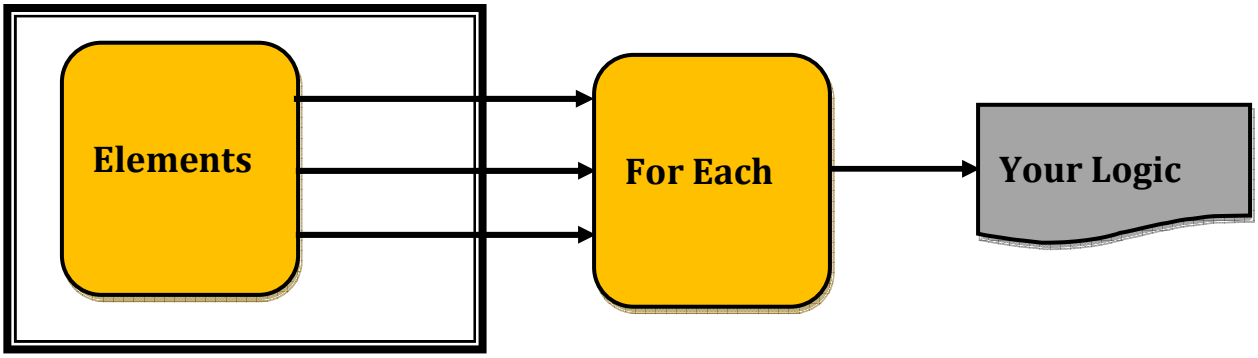
Java8Lab47.java

```
// import statements
public class Java8Lab47 {
public static void main(String[] args) {
List<String> list = Arrays.asList("Sri", "Manish", "Dande", "Ranjan","Sri", "Ram", "Abhi", "Shyam","Manish");
System.out.println(list);
list.forEach(System.out::println);
List<String> result = list.stream().filter(nm -> nm.startsWith("S")).collect(Collectors.toList());
System.out.println(result);
long count=list.stream().filter(nm -> nm.startsWith("S")).count();
System.out.println(count);
System.out.println(list);
List<String> distinctList=list.stream().distinct().collect(Collectors.toList());
System.out.println(distinctList);
System.out.println(list.stream().allMatch(nm -> nm.contains("a")));
System.out.println(list.stream().anyMatch(nm -> nm.contains("a")));
List<String> list = Arrays.asList("Sri", "Manish", "Dande", "Ranjan","Sri", "Ram", "Abhi", "Shyam","Manish");
List<String> result=list.stream().limit(4).collect(toList());
System.out.println(list);              System.out.println(result);
result=list.stream().skip(5).collect(toList());          System.out.println(result);
Stream<String> stream=list.stream();
stream.forEach(System.out::println);
stream.forEach(System.out::println);        // IllegalStateException  } }
```

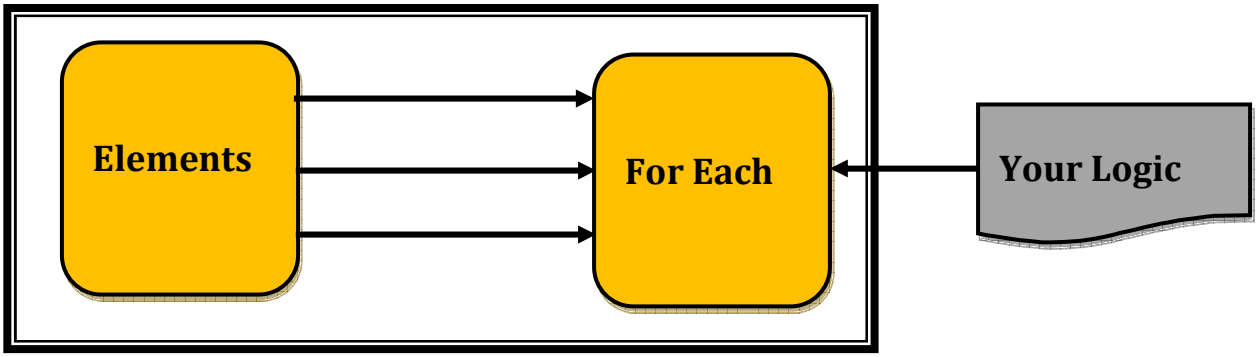
External V/S Internal Iteration

- **Collections requires you to explicitly iterate. It requires creating suitable iterator ad iteration logic.**
- **Streams do the iteration internally. Streams API comes with an extensive list of operations to support complicated data processing queries.**

External Iteration in Collection



Internal Iteration in Stream



- void forEach(BiConsumer<? super K, ? super V> action)
  - This method performs the operation specified in the 'action' on each Map Entry (key and value pair).
- V getOrDefault(Object key, V defaultValue)
  - Returns the value mapped to the key.
  - If the key is not present, returns the default value.
- V putIfAbsent(K key, V value)
  - If the key is not present or if the key is mapped to null, then the key-value pair is added to the map and the result returns null.
  - If the key is already mapped to a value, then that value is returned
- V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)
  - If the key is not present or if the value for the key is null, then adds the key-value pair to the map.
  - If the key is present then replaces the value with the value from the remapping function. If the remapping function return null then the key is removed from the map.
- boolean remove(Object key, Object value)
  - Removes the key only if its associated with the given value
- V replace(K key, V newValue)
  - If the key is present then the value is replaced by newValue. If the key is not present, does nothing.
- boolean replace(K key, V oldValue, V newValue)
  - If the key is present and is mapped to the oldValue, then it is remapped to the newValue.

Java8Lab48.java

```
// import statements
public class Java8Lab48 {
public static void main(String[] args) {
Map<Integer, String> map = new LinkedHashMap<>();
map.put(123, "Sri");                    map.put(99, "Abhi");
map.put(345, "Kumar");                  map.put(88, "Dande");
map.put(44, "Manish");                  map.put(189, "Rahul");
map.forEach((key,value) -> System.out.println("Sid "+key+",\tName :"+value));
System.out.println(map.get(123));
System.out.println(map.getOrDefault(123,"JLC"));
System.out.println(map.get(89));
System.out.println(map.getOrDefault(89,"JLC"));
System.out.println("**********");
System.out.println(map);
map.put(123, "Nivas");
System.out.println(map);
map.putIfAbsent(123, "Ranjan");
```

```
map.putIfAbsent(78, "Manoj");
System.out.println(map);
map.remove(88);
System.out.println(map);
map.remove(44,"Sri");
System.out.println(map);
}
}
```

Java8Lab49.java

```
// import statements
public class Java8Lab49 {
public static void main(String[] args) {
List<Student> list = Arrays.asList(
new Student(45,"Sri","HYD"),        new Student(90,"Manish","DEL"),
new Student(76,"Abhi","HYD"),       new Student(96,"Rahul","BLR"),
new Student(56,"Manoj","HYD"),  new Student(90,"Kumar","DEL"),
new Student(91,"Nivas","HYD")     );
List<String> names= getNameForAGradeSortedByCity(list);
System.out.println(names);
}
private static List<String> getNameForAGradeSortedByCity(List<Student> list){
List<Student> aGradeStudents = new ArrayList<>();
for(Student st : list){
if(st.getMarks() >= 90)
aGradeStudents.add(st);
}
Comparator<Student> comp = new Comparator<Student>(){
public int compare(Student s1, Student s2) {
return s1.getCity().compareTo(s2.getCity());
}
};
Collections.sort(aGradeStudents,comp);
List<String> names = new ArrayList<>();
for(Student st : aGradeStudents){
names.add(st.getSname());
}
return names;
} }
class Student {
int marks;          String sname;      String city;
// Getter and Setters
// Parameterized Consutuctor
}
```

Java8Lab50.java

```
// import statements
public class Java8Lab50 {
public static void main(String[] args) {
List<Student> list = Arrays.asList(
new Student(45,"Sri","HYD"),                new Student(90,"Manish","DEL"),
new Student(76,"Abhi","HYD"),               new Student(96,"Rahul","BLR"),
new Student(56,"Manoj","HYD"),              new Student(90,"Kumar","DEL"),
new Student(91,"Nivas","HYD")
);
List<String> names= list.stream().filter(st ->
st.getMarks()>=0).sorted(comparing(Student::getCity)).map(Student::getSname).collect(toList());
System.out.println(names);
} }
```

Java8Lab51.java

```
// import statements
public class Java8Lab51 {
public static void main(String[] args) {
List<Student> list = Arrays.asList(
new Student(45,"Sri","HYD"),                new Student(90,"Manish","DEL"),
new Student(76,"Abhi","HYD"),               new Student(96,"Rahul","BLR"),
new Student(56,"Manoj","HYD"),              new Student(90,"Kumar","DEL"),
new Student(91,"Nivas","HYD")
);
List<String> names= list.stream().map(Student::getSname).collect(toList());
System.out.println(names);
names= list.stream().map(st -> st.getSname().toUpperCase()).collect(toList());
System.out.println(names);
}
}
```

Java8Lab52.java

```
// import statements
public class Java8Lab52 {
public static void main(String[] args) {
List<Student> list = Arrays.asList(new Student(45, "Sri", "HYD"),
new Student(90, "Manish", "DEL"),                   new Student(76, "Abhi", "HYD"),
new Student(96, "Rahul", "BLR"),                    new Student(56, "Manoj", "HYD"),
new Student(90, "Kumar", "DEL"),                    new Student(91, "Nivas", "HYD"));
System.out.println(list.stream().collect(counting()));
Map<String, List<Student>> cityWise = list.stream().collect(groupingBy(Student::getCity));
System.out.println(cityWise);
Integer sum = list.stream().collect(summingInt(Student::getMarks));
System.out.println(sum);
} }
```

Java8Lab53.java

```
// import statements
public class Java8Lab53{
public static void main(String[] args) {
List<String> words = Arrays.asList("Hello","World","Manish");
List<String[]> result=words.stream().map(word -> word.split("")).distinct().collect(toList());
System.out.println(result);
List<String> resultChars=words.stream().map(word ->
word.split("")).flatMap(Arrays::stream).distinct().collect(toList());
System.out.println(resultChars);
} }
```

Numeric Stream

Java8Lab53.java

```
// import statements
public class Java8Lab53 {
public static void main(String[] args) {
List<Student> list = Arrays.asList(new Student("Sri", 100),
new Student("Manish", 200),
new Student( "Abhi", 50),
new Student("Rahul", 100));
int totalFee=list.stream().map(Student::getFeePaid).reduce(0, Integer::sum);
System.out.println(totalFee);
int totalFee1=list.stream().mapToInt(Student::getFeePaid).sum();
System.out.println(totalFee1);
}
}
class Student {
String sname;
int feePaid;
Student(String sname, int feePaid) {
this.sname = sname;
this.feePaid = feePaid;
}
// Setters and Getters
public void setFeePaid(int feePaid) {
this.feePaid = feePaid;
}
}
```