Workshop Java 8
==========

=> Introduction to java 8, Why i should care for it?
=> functional interface
=> Lambda expressions
=> diff bw ann inner class vs lambda expression
(lam exp == method ref)
=> performance
https://www.infoq.com/articles/Java-8-Lambdas-A-Peek-Under-the-Hood
=> Passing code with behavior parameterization
=> introduction to stream processing
=> functional interfaces defined in Java 8
=> Working with streams details
=> Primitive stream specializations
=> Parallel data processing and performance
=> join & fork
=> Optionals
=> Collecting data with streams
=> Book Application case study
=> Debugging and logging
=> Design pattern with java 8
=> fork and join example, issues
=> Basics of Parallle strams
=> consideration and issues, goachas parallel stream

==========
Introduction to java 8, Why i should care for it?
==========
Java 8: why should you care? NOW

As commodity CPUs have become multicore, Difficult to take advantage with traditional threads programming

Java 5 added building blocks like thread pools and concurrent collections

Java 7 added the fork/join framework, making parallelism more practical but still difficult

Why Java 8?
==========
more concise code and simpler use of multicore processors

What most important to us:
---
interface evolvotation
Lambda expression
Techniques for passing code to methods (behaviour parameterization)
The Streams API
Parallel stream processing and perfomrance
Improvement date/time api

Hello world: calculating prime Number 1 to 1_000_000

```
class Prime{
	public static boolean isPrime(Long n){
		boolean isPrimeNumber=true;
		for(int i=2;i<n;i++){
			if(n%i==0)
				isPrimeNumber=false;
		}
		return isPrimeNumber;
	}
}
```

System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "8");
or
java -Djava.util.concurrent.ForkJoinPool.common.parallelism=8 DemoTakingAdvOfParallelProcessingwithOutPhd

==========
functional interface
==========

Funcational Interface?

=> Java 8 introduces a new FunctionalInterface annotation type that lets you annotate an interface as being functional.

For example:

```
@FunctionalInterface
public interface Runnable
{
    public abstract void run();
}
```

=> A functional interface can define as many default and static methods as it requires but it must declare exactly one abstract method, or the compiler will complain that it isn't a functional interface.

Example of functional interface in Java?

Readable ,Callable ,Iterable, Closeable , Flushable, Formattable, Comparable, Comparator etc

Java-style functional programming

JSR 335 Lambda Expression Closure
JEP 107 :Bulk data operation for collections forEach, filter, map,reduce

Java DNA

Structured ==> Reflective=> Object Oriented =>Functional => Imperative=>Concurrent => Generic

What Java is now?

".....Is a blend of imperative and object oriented programming enhanced with functional flavors"

Methods vs. Functions

	Method	Pure Function
Mutation	A method mutates , i.e., it modifies data and produces other changes and side effects.	A pure function does not mutat it just inspects data and produces results in form of new data.
How vs. What	Methods describe how things are done.	Pure functions describe what is done rather than how it is done
Invocation Order	Methods are used imperatively order of invocation matters.	Pure functions are used declaratively

Code vs. Data

```
MCollections.sort(list, new Comparator<Student>() {

        @Override
        public int compare(Student o1, Student o2) {
                return 0;
        }
});
```

ethods are code , i.e.,
they are invoked and executed

Functions are code-as-data
they are not only executable,
but are also passed around like data.

Interface evaluation
==========

=> Problem with java interface?

You provide abstract method in the interface, if 10 method is delare then implementating class need to provide implementation to 10 method

if we need to change the interface (ie add new method prototype) the all the class herarchy is get effected!

=> Java 8 solve these issues by supporting default implementation and static method to interface

Ex: Java 8 interface support

```
public interface Addressable
{
        String getStreet();
        String getCity();

        default String getFullAddress()
        {
         return getStreet()+", "+getCity();
         }
}
```

=> Now we are free not to implement getFullAddess() method
=> We can override if it is required
=> Even we can even re-declare it as an abstract method in an abstract class

Default methods have two important use cases

Evolving existing interfaces

==> To implement the new Streams API, it was necessary to evolve the Collections Framework's java.util.Collection interface by adding new default Stream<E> stream() and default Stream<E> parallelStream() methods.

Without default methods, Collection implementers such as the java.util.ArrayList class would have been forced to implement these new methods or break source/binary compatibility

Increasing design flexibility

==> Abstract classes have traditionally been used to share functionality between various concrete subclasses. However, single-class extension has limited this design choice.

=> Default methods offer greater flexibility because you can implement an interface at any point in the class hierarchy and access the interface's default methods from the implementing classes.

Also, it's no longer necessary to create adapter classes for
multi-method event listener interfaces. Instead, you can add a default method for
each listener method to the event listener interface and
override these methods as necessary

Note:

We cannot use default methods to override any of the non-final methods in the java.lang.Object class

So it will not work:

```
public interface Foo
{
        default boolean equals(Object o)
        {
        return false;
        }
}
```

Static method inside interface: (Java 8)

In Java 8 a static methods be defined inside interfaces mainly to assist default methods.

For example:
---
the java.util.Comparator interface defines the following static method:

static <T> Comparator<T> comparingDouble(ToDoubleFunction<? super T> keyExtractor)

As well as being directly invocable, comparingDouble() is invoked from
this default method of Comparator

default Comparator<T> thenComparingDouble(ToDoubleFunction<? super T> keyExtractor)

Note:
When you implement an interface that contains a static method, the static method is still part of the interface and not part of the implementing class.

For this reason, you cannot prefix the method with the class name.
Instead, you must prefix the method with the interface name

```
interface X
{
        static void foo()
        {
                System.out.println("foo");
         }
}

class Y implements X
{
}

public class Z
{
         public static void main(String[] args)
        {
                 X.foo();
                // Y.foo(); // won't compile
        }
}
```

```
        // Expression Y.foo() will not compile because foo() is a static member
        // of interface X and not a static member of class Y
```

issues ? multiple inheritance 3 rules
---

1. Any class win over interface
   ie if there is a method with a body or an ab declaration
   in the superclass chain, we ignore the interface completely
2. subtype win over supertype
3. If previous 2 rules don't give us the answer, the subclass must implements the method/ declare it abst.

==========

Lambda Expressions

==========

Lambda Expressions and Functional Interfaces

==========

f(x)=3x+4
functional programming = immutability+ thread safe + dist programming?

FP==> declerative programming?
Explicit programming?

=> 3GL vs 4GL(SQL)

Lambda expression

What is lambda expression?

-> Conceptually, a lambda expression is a function .
It is an unnamed piece of reusable functionality.

Feature of Lambda Expression
---

1. its is anonymous (no name)

2. function : we call it as function as it is not assciated with a
   particular class like a method is.
   like a method a lambda has a list of parameters, a body, a
   return type and a possible list of exceptions that
   can be thrown

3. passed around: a lambda expression can be passed around or
   can be stored in an variable

4. Concise: d not need to write a lot of boilerplate like you
   do for annonymous inner class

5. cloure

Example:

```
public class LambdaDemo {
    public static void main(String[] args){
        Runnable r = new Runnable() {
            @Override
            public void run()
            {
            System.out.println("Running");
        }
```

```
                };

                        Thread t=new Thread(r);

                        Thread t2=new Thread(() -> System.out.println("Running"));
                }
        }
```

Example:

```
        interface Operation {
        int apply(int a, int b);
}

class Logic {
        public static int operation(int a, int b, Operation operation) {
                return operation.apply(a, b);
        }
}

public class DemoBehaviour {

        public static void main(String[] args) {
                int sum = Logic.operation(2, 2, (int a, int b) -> a + b);
                int mul = Logic.operation(2, 2, (int a, int b) -> a * b);
        }
}
```

Note:

=> Associated with a lambda is an implicit target type
(the type of object to which a lambda is bound).

=> Because the target type must be a functional interface and is
inferred from the context, lambdas can appear in only the following contexts

---

Give me details of all hidden file in a dir

```
File myDir = new File("\\user\\admin\\deploy");
if (myDir.isDirectory()) {
  File[] files = myDir.listFiles(
    new FileFilter() {
    public boolean accept(File f) { return f.isFile(); }
   }
  );
}
```

Using a lambda expression it looks like this:

```
File myDir = new File("\\user\\admin\\deploy");
if (myDir.isDirectory()) {
  File[] files = myDir.listFiles(
    (File f) -> { return f.isFile(); }
  );
}
```

Lambdas VS Anonymous Inner Classes
---
1. Syntax
Anonymous inner class:

```
File[] fs = myDir.listFiles(
  new FileFilter() {
   public boolean accept(File f) { return f.isFile(); }
  }
);
```

Lambda expression in various forms including method reference:

```
File[] files = myDir.listFiles(  (File f) -> {return f.isFile(); } );
File[] files = myDir.listFiles(  f -> f.isFile()  );
File[] files = myDir.listFiles(  File::isFile );
```

2. Runtime Overhead:
---
lambda expressions have the potential for optimizations
and reduced runtime overhead compared to anonymous inner classes

At runtime anonymous inner classes require:
-> class loading,
-> memory allocation and object initialization, and
-> invocation of a non-static method.

3. Variable Binding
---
variable capture: an inner class has access to all final variables of its enclosing context

```
void method() {
      final int  cnt = 16;
      Runnable r = new Runnable() {
         public void run() {
            System.out.println("count: " +  cnt );
         }
      };
      Thread t = new Thread(r);
      t.start();
       cnt ++;   // error: cnt is final
}
```

Lambda expressions, too

```
      Runnable r = () -> { System.out.println("count: " +  cnt ); };

      Thread t = new Thread(r);
      t.start();

       cnt ++;   // error: cnt is implicitly final
}
```

Difference:
---
ariables from the enclosing context that are used inside a
lambda expression are implicitly final (or effectively final)

==========
Passing code with behavior parameterization
==========

behavior parameterization(Techniques for passing code to methods)

What it is?

=> Suppose you want to write two methods that differ in only a few lines of code;
you can now just pass the code of the parts that differ as an argument

=> this programming technique is shorter, clearer, and less error prone than the common tendency to use copy and paste

=> Experts will here note that behavior parameterization could, prior to Java 8, be encoded using anonymous classes

Funcational programming?

Behavior parameterization

The Java 8 feature of passing code to methods (and also being able to return it and incorporate it into data structures) also provides access to a whole range of additional techniques that are commonly referred to as functional-style programming.

Passing code: an example

Problem: Farmer need to find out apple that are

1. green in color
2. having weight more then 150 gm

Attempt 1:

```
public class Apple {
	private String color;
	private int weight;
	//getter setter ctr etc
}

public class Inventrory {
	public static List<Apple> getGreenApple(List<Apple> apples){
		List<Apple>geenApples=new ArrayList<Apple>();
		for(Apple temp:apples){
			if(temp.getColor().equalsIgnoreCase("green"))
				geenApples.add(temp);
		}
		return geenApples;
	}
	public static List<Apple> getAppleswithMoreWeights(List<Apple> apples){
		List<Apple>applesWithMoreWt=new ArrayList<Apple>();
		for(Apple temp:apples){
			if(temp.getWeight()>150)
				applesWithMoreWt.add(temp);
		}
		return applesWithMoreWt;
	}
}
```

Problem:dangers of copy and paste, DRY voilation

Attempt 2;

=> Java 8 makes it possible to pass the code of the condition as an argument,
thus avoiding code duplication of the filter method

```
interface Predicate<T>{
	boolean test(T t);
}
```

Now improved method:

```
public static List<Apple> filterApples(List<Apple> apples, Predicate<Apple>p){
	List<Apple>applesOnCondiation=new ArrayList<Apple>();
	for(Apple temp:apples){
		if( p.test(temp))
			applesOnCondiation.add(temp);
	}
	return applesOnCondiation;
}
```

Problem:

How to pass predicate?

Traditional way? Using inner classes!

```
List<Apple>grrenApples=Inventrory2.filterApples(list, new Predicate<Apple>() {

		@Override
		public boolean test(Apple t) {
			return t.getColor().equals("green");
		}
	});
```

Better solution use lambda function
```
List<Apple>grrenApples=Inventrory2.filterApples(list, (Apple a)-> "green".equals(a.getColor()));
```

```
(Apple a) -> a.getWeight() > 150
(Apple a) -> a.getWeight() < 80 ||"brown".equals(a.getColor())
```

Even Better use method reference
```
List<Apple>grrenApples=Inventrory2.filterApples(list, Inventrory2::isGreenApple);
```

==========
Steam in Java 8 an introduction
==========

Streams API in Java 8 lets you write code that's:
=> Declarative More concise and readable
=>Composable Greater flexibility
=> Parallelizable Better performance

What are streams

=> Streams are an update to the Java API that lets you manipulate collections of data in a declarative way like what SQL Does to database!

=> you express a query rather than code an ad hoc implementation for it

=> For now you can think of them as fancy iterators over a collection of data.

=> In addition,streams can be processed in parallel transparently, without you having to write any multithreaded code! (parallelization)

Why Streams?

=> Reduce boilerplate and obscurity involving processing collections

=> difficulty leveraging multicore with the Streams API (java.util.stream).

=> The first design motivator is that there are many data processing patterns (similar to filterApples in the previous section, or operations familiar from database query languages such as SQL) that occur over and over again and that would benefit from forming part of a library: filtering data based on a criterion (for example, heavy apples), extracting data (for example, extracting the weight field from each apple in a list), or grouping data (for example, grouping a list of numbers into separate lists of even and odd numbers), and so on.

=> The second motivator is that such operations can often be parallelized.

collection internal iteration	vs	Streams External iteration
We have to maintain iteration logic using iterator/for loop		We don't have to manage iteration. Liberary is doing it for us
Like 3GL		Like 4GL(SQL like)
No chance for optimization from JVM		possible, easy
Very difficult to run on multicore system		possible,easy

Example:Sequential processing:

```
List<Apple>heavyApples=list.stream().filter((Apple a)-> a.getWeight()>150).
        collect(Collectors.toList());
```

Parallel processing:

```
List<Apple> heavyApples =inventory.parallelStream().....;
```

==========

functional interfaces defined in Java 8

==========

Most important Predefined functinal interface in Java 8 :

FI	Description	Method name	Use
Predicate<T>	T->boolean	test()	Used as filter
Consumer<T>	T-> void	accept()	Used in forEach
Function<T,R>	T->R	apply()	Used in map() operation
Supplier<T>	()->T	get()	

Predicate<T>
==========
```
@FunctionalInterface
public interface Predicate<T> {
	boolean test(T t);

	//...
}
```

Example:filter

```
Predicate<Integer>even=x->x%2==0;

List<Integer>evens2=list.stream().filter(x->x%2==0).collect(Collectors.toList());

Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();

List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);
```

Consumer<T>
==========
=> consumer interface defines an abstract method named accept that
takes an object of generic type T and returns no result (void).

=> You might use this interface when you need to access an object of type T and
perform some operations on it.

=> For example, you can use it to create a method forEach, which takes a list of Integers
and applies an operation on each element of that list.

```
@FunctionalInterface
public interface Consumer<T>{
	void accept(T t);
}

public static<T> void forEach(List<T> list, Consumer<T> c){

	for(T i:list){
		c.accept(i);
	}
}

forEach(Arrays.asList(1,2,3,5),(Integer i) -> sysout(i));
```

Function<T,R>
==========
Mapping process: Accept object of one type map it to another type
---

Ex:Want to print score of all players!

```
public class Player {
private String name;
private String team;
private int score;  //getter setter ctr etc
}
plist.stream().map(p-> p.getScore()).forEach(x-> System.out.println(x));
plist.stream().mapToInt(p-> p.getScore()).forEach(x-> System.out.println(x));
```

Reduction

```
	List<Integer>list=Arrays.asList(4,5,6,7,4,5,88);

	int sum=list.stream().reduce(0, (a,b)->a+b);
	System.out.println(sum);
```

Collectors

```
List<String>letter=Arrays.asList("a","b","c","d");

	String concat="";
	for(String temp:letter)
		concat+=temp;
	//java 8

	String concat2=letter.stream().collect(Collectors.joining());
```

==========
Working with streams introduction
==========
Consider:
```
public class Dish {
	private String name;
	private boolean vegetarian;
	private int calories;
	public enum Type { MEAT, FISH, OTHER }
	private Type type;
}


List<Dish> menu = Arrays.asList(
	new Dish("pork", false, 800, Dish.Type.MEAT),
	new Dish("beef", false, 700, Dish.Type.MEAT),
	new Dish("chicken", false, 400, Dish.Type.MEAT),
	new Dish("french fries", true, 530, Dish.Type.OTHER),
	new Dish("rice", true, 350, Dish.Type.OTHER),
	new Dish("season fruit", true, 120, Dish.Type.OTHER),
	new Dish("pizza", true, 550, Dish.Type.OTHER),
	new Dish("prawns", false, 300, Dish.Type.FISH),
	new Dish("salmon", false, 450, Dish.Type.FISH) );
```

Example: return the names of dishes that are low in calories (<400),
sorted by number of calories
Pre java 8 way:

```
List<Dish> lowCaloricDishes = new ArrayList<>();
	for(Dish d: menu){
	if(d.getCalories() < 400){
		lowCaloricDishes.add(d);
	}
}
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
	public int compare(Dish d1, Dish d2){
	return Integer.compare(d1.getCalories(), d2.getCalories());
	}
});

List<String> lowCaloricDishesName = new ArrayList<>();
for(Dish d: lowCaloricDishes){
	lowCaloricDishesName.add(d.getName());
}
```

Java 8:

```
List<String> lowCaloricDishesName =menu
                                    .stream()
                                            .filter(d -> d.getCalories() < 400)
                                                    .sorted(comparing(Dish::getCalories))
                                                            .map(Dish::getName)
                                                                    .collect(toList());
```

To exploit a multicore architecture and execute this code in parallel, you need only change stream() to parallelStream():

```
List<String> lowCaloricDishesName =menu
                                            .parallelStream()
                                                    .filter(d -> d.getCalories() < 400)
                                                            .sorted(comparing(Dishes::getCalories))
                                                                    .map(Dish::getName)
                                                                            .collect(toList());
```

what exactly is a stream?
==========
A short definition is :

a sequence of elements from a source that supports data processing operations.

Lets break down this definition step by step:

=> Sequence of elements:

Like a collection, a stream provides an interface to a sequenced set of values of a specific element type.

Because collections are data structures, they\92re mostly about storing and accessing elements with specific time/space complexities (for example, an ArrayList vs. a LinkedList).

But streams are about expressing computations such as filter, sorted, and map that you saw earlier.

Collections are about data; streams are about computations.

=> Source

Streams consume from a data-providing source such as collections, arrays, or I/O resources.

Note that generating a stream from an ordered collection preserves the ordering. The elements of a stream coming from a list will have the same order as the list.

=> Data processing operations

Streams support database-like operations and common operations from functional programming languages to manipulate data, such as filter, map, reduce, find, match, sort, and so on. Stream operations can be executed either sequentially or in parallel.

Stream operation

=> The Stream interface in java.util.stream.Stream defines many operations.
intermediate operations

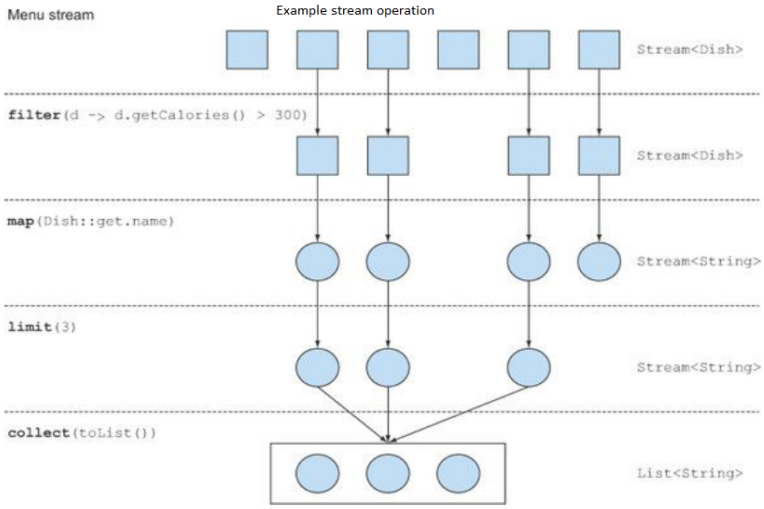
=> Filter, map, and limit can be connected together to form a pipeline
=> Stream operations that can be connected
=> lazy operation, intermediate operations can usually be merged and processed into a single pass by the terminal operation.

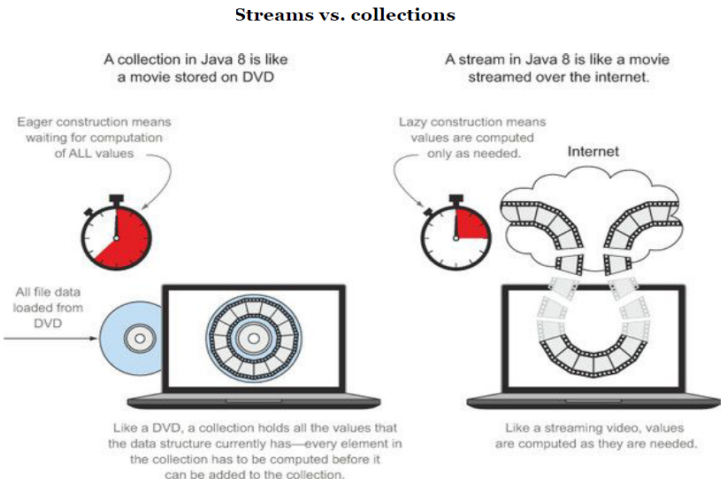
terminal operations

=> collect causes the pipeline to be executed and closes it.

```
List<String> names = menu.stream().
                                filter(d -> d.getCalories() > 300).
                                    map(Dish::getName).
                                        limit(3)
                                         .collect(toList());
```

Ref: diagram 1





Ref: diagram 2: comparing steams with collection

Note: stream should traversed once

List<String>list=.....;
Stream<String> s=list.stream();
Sysout(s.foreach(s->System.out::println());
Sysout(s.foreach(s->System.out::println());//////Will give an error

More on intermediate operations: Lazy calculation

Horizontal processing vs Vertical Processing (Beaware from Sorting)

=> Intermediate operations such as filter or sorted return another stream as the return type.

=>This allows the operations to be connected to form a query. What\92s important is that intermediate operations don\92t perform any processing until a terminal operation is invoked on the stream pipeline they re lazy.

=> This is because intermediate operations can usually be merged and
processed into a single pass by the terminal operation.

To understand whats happening in the stream pipeline, modify the code so each lambda also prints the current dish its processing (like many demonstration and debugging techniques, this is appalling programming style for production code but directly explains the order of evaluation when you are learning....

```
List<String> names = menu.stream()
                        .filter( d->{
                                        Sysout("filtering"+d.getName();
                                        return d -> d.getCalories() > 300;

                                })
                        .map(d->{
                                Sysout("mapping:"+d.getName());
                                return d.getName();
                                })
                        .limit(3)
                        .collect(toList());
```

=> several optimizations due to the lazy nature of streams.
=>First, despite the fact that many dishes have more than 300 calories, only the first three are selected!

This is because of the limit operation short-circuiting

Second, despite the fact that filter and map are two separate operations, they were merged into the same pass (we call this technique loop fusion).
==========
streams in detail
==========
What we are going to cover in this session:

=> Filtering, slicing, and matching
=> Finding, matching, and reducing
=> Using numeric streams such as ranges of numbers
=> Creating streams from multiple sources
=> Infinite streams

Filtering
=========

Ex:Getting all veg dishes

```
List<Dish>vegDishes=menu.stream().filter(Dish::isVegitable).collect(Collector.toList());
```

Filtering unique elements in a stream

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);

numbers.stream().filter(i -> i % 2 == 0).distinct().forEach(System.out::println);
```

Hew to create a custom mapper in java 8?
==========

http://www.leveluplunch.com/java/tutorials/016-transform-object-class-into-another-type-java8/
==========

```
class Foo {
	private int a, b, c;
	///
}
class Bar {
	private int p, q;
	///
}

public class DemoMapToObj {
	static Function<Foo, Bar> chnager = new Function<Foo, Bar>() {
		@Override
		public Bar apply(Foo f) {
			Bar bar = new Bar(f.getA(), f.getB());
			return bar;
		}
	};

	public static void main(String[] args) {

		List<Foo> foos = Arrays.asList

		(new Foo(1, 2, 3), new Foo(1, 2, 3), new Foo(1, 2, 3));

		List<Bar> gettingBars = foos.stream().map(chnager)
					.collect(Collectors.toList());


		gettingBars.forEach(x-> System.out.println(x));
	}
}
```

Truncating a stream
==========

=> Streams support the limit(n) method, which returns another
stream thats no longer than a given size.

=> The requested size is passed as argument to limit. If the stream is ordered,
the first elements are returned up to a maximum of n.

Ex: create a List by selecting the first three dishes that have more than 300 calories as follows:

```
List<Dish> dishes = menu.stream().filter(d -> d.getCalories() > 300).limit(3).collect(toList());
```

skipping elements
==========

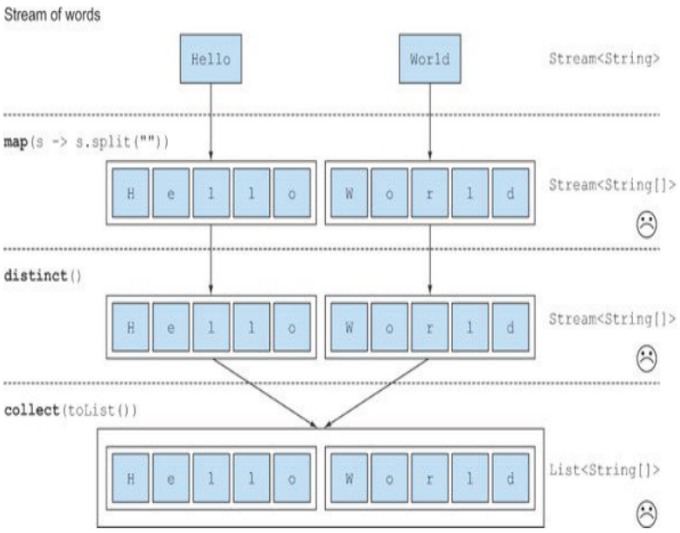
=> Streams support the skip(n) method to return a stream that discards the first n elements
List<Dish> dishes = menu.stream().filter(d -> d.getCalories() > 300).skip(2).collect(toList());

flatMap
=========

Ex: return a list of all the unique characters for a list of words?
For example, given the list of words ["Hello", "World"]
we like to return the list ["H", "e", "l", "o","W", "r", "d"]

How to do it?

Incorrect implementation : need of flatMap



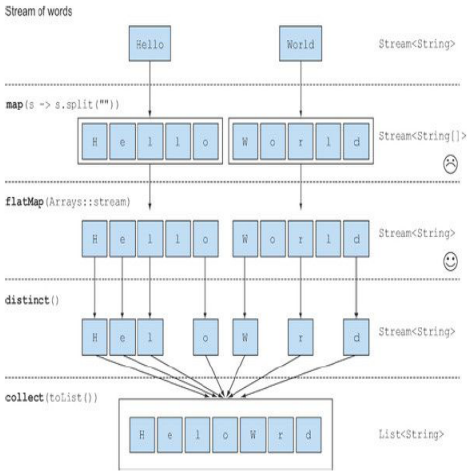

```
words.stream()
     .map(word -> word.split(""))
     .distinct()
     .collect(toList());
```

List<String> list=Arrays.asList("foo","bar","jar");
List<String>chars=
list.stream()
.map(s->s.split(" "))
.distinct()
.distinct()
.collect(Collectors.toList());

```
List<String> uniqueCharacters =
  words.stream()
       .map(w -> w.split(""))
       .flatMap(Arrays::stream)
       .distinct()
       .collect(Collectors.toList());
```

Converts each word into an array of its individual letters

Flattens each generated stream into a single stream



Correct implementation using flatMap:

Using the flatMap method has the effect of mapping each array not with a stream but *with the contents of that stream.* All the separate streams that were generated when using map(Arrays::stream) get amalgamated—flattened into a single stream

Whe
---

http://stackoverflow.com/questions/22382453/java-8-streams-flatmap-method-example

What happens in this code?

```
-------------------------
final Stream<Integer>stream = Stream.of(1,2,3,4,5,6,7,8,9,10);
stream.flatMap();
```

=> It doesn't make sense to flatMap a Stream that's already flat,
like the Stream<Integer>

=> However, if you had a Stream<List<Integer>> then it would make sense and you could do this:

```
Stream<List<Integer>> integerListStream = Stream.of(
                                Arrays.asList(1, 2),
                                Arrays.asList(3, 4),
                                 Arrays.asList(5)
        );


        Stream<Integer> integerStream = integerListStream .flatMap(Collection::stream);
        integerStream.forEach(System.out::println);
```

O/P: print 1 to 5

Finding and matching
==========

=> Another common data processing idiom is finding whether some
elements in a set of data match a given property.
=> The Streams API provides such facilities through the following methods:

allMatch,
anyMatch,
noneMatch,
findFirst,
findAny

anyMatch
=========

=> method can be used to answer the question
Is there an element in the stream matching the given predicate?

For example, you can use it to find out whether the menu has a vegetarian option:

```
if(menu.stream().anyMatch(Dish::isVegetarian)){
        System.out.println("The menu is (somewhat) vegetarian friendly!!");
}
```

allMatch
========

Checking to see if a predicate matches all elements

For example, you can use it to find out whether the menu ishealthy
(ie. all dishes are below 1000 calories):

```
boolean isHealthy = menu.stream().allMatch(d -> d.getCalories() < 1000);
```

noneMatch
========

opposite of allMatch

```
boolean isHealthy = menu.stream().noneMatch(d -> d.getCalories() >= 1000);
```

NullPE?
########
=> pain for developer
=> Tony Hoare 1965
=> Null References: The Billion Dollar Mistake
https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

findAny
=======
The findAny method returns an arbitrary element of the current stream

```
Optional<Dish> dish =menu.stream()
                        .filter(Dish::isVegetarian)
                            .findAny();
```

What is this optional?

AVOID NULLPOINTEREXCEPTION
==========
=> The Optional<T> class (java.util.Optional) is a container class
to represent the existence or absence of a value.

=> In the previous code, its possible that findAny doesnt find any element.

=> Our code can benefit from using Optional to avoid bugs related to null checking.

Methods available with Optional:

=> isPresent()
returns true if Optional contains a value, false otherwise.

=> ifPresent(Consumer<T> block)
executes the given block if a value is present.

=> T get()
returns the value if present; otherwise
it throws a NoSuchElement-Exception

=> T orElse(T other)
returns the value if present; otherwise it returns a default value.

Ex:

```
menu.stream()
        .filter(Dish::isVegetarian)
        .findAny().ifPresent(d->sysout(d.getName()));
```

findFirst
==========
Finding the first element

Ex: given a list of numbers,finds the first square that's divisible by 3:

```
List<Integer> someNumbers = Arrays.asList(1, 2, 3, 4, 5);

Optional<Integer> firstSquareDivisibleByThree =someNumbers.stream()
                        .map(x -> x * x)
                            .filter(x -> x % 3 == 0)
                                .findFirst(); // 9
```

reducing (fold operations)
==========
=> How we can combine elements of a stream to express more complicated queries such as:

Calculate the sum of all calories in the menu, or

What is the highest calorie dish in the menu?

Ex: finding sum of all element of the array:

```
int sum = 0;
for (int x : numbers) {
        sum += x;
}
```

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
int product = numbers.stream().reduce(1, (a, b) -> a * b);
```

reduce takes two arguments:

-> An initial value, here 0.

-> A BinaryOperator<T> to combine two elements and produce a new value;
here we use lambda (a, b) -> a + b

Reduction in case of No initial value

```
Optional<Integer> sum = numbers.stream().reduce((a, b) -> (a + b));
```

Why does it return an Optional<Integer>?

What happens if the case when the stream contains no elements?

=> The reduce operation can't return a sum because it doesn\92t have an initial value.
=> In that case result would be wrapped in an Optional object to indicate
that the sum may be absent.

Maximum and minimum
==========
```
Optional<Integer> max = numbers.stream().reduce(Integer::max);
Optional<Integer> min = numbers.stream().reduce(Integer::min);
```

Ex: How would you count the number of dishes in a
stream using the map and reduce methods?

```
int count = menu.stream()
                .map(d -> 1)
                        .reduce(0, (a, b) -> a + b);
```

Or better use build in method count() on stream object: long count = menu.stream().count();

Note :diagram 8 : List of all Intermediate and terminal operations

==========
Primitive stream specializations
==========
=> IntStream,DoubleStream, and LongStream,
specialize the elements of a stream to be int,long, and double

=> To avoid boxing costs

```
Eg: int calories = menu.stream()
        .map(Dish::getCalories)
                .reduce(0, Integer::sum);
```

Mapping to numeric stream

```
        Eg: int calories = menu.stream()                //return an Stream<Dish>
        .mapToInt(Dish::getCalories)                            //return an IntStream
                .sum();             //if the stream were empty, sum would return 0 by default
```

Default values: OptionalInt

=> Element of an IntStream by calling the max method,which returns an OptionalInt:

```
OptionalInt maxCalories = menu.stream()
        .mapToInt(Dish::getCalories)
                .max();
```

=> You now process the OptionalInt explicitly to define a default value if theres no maximum

```
int max=maxCalories.orElse(1);
```

Numeric ranges
==========
=> A common use case when dealing with numbers is working with ranges of numeric values.

=> Let we like to generate all numbers between 1 and 100.

=> Java 8 introduces two static methods available on IntStream and LongStream to help
generate such ranges:

1> range(start, end)
2> rangeClosed(start, end)

=> range is exclusive, whereas rangeClosed is inclusive

```
Ex: IntStream evenNumber=Intstream.rangeClosed(1,100).filter(n->n%2==0);
        Sysout(evenNumber.count());
```

Building streams
==========

3 Ways:

1> Streams from values

```
Stream<String> stream = Stream.of("Java 8 ", "Lambdas ", "In ", "Action");
```

2> Streams from arrays

```
int[]numbers={2,3,4,5,6,7};
int sum=Arrays.stream(numbers).sum();
```

3> Streams from files

Java NIO updated to take advantage of Stream API

Many static method in java.nio.file.Files return an stream

```
Eg: long uniqueWords=0;
try(Stream<String>lines=Files.line(Path.get("data.txt"))){
        uniqueWords=lines.flatMap(line-> Arrays.stream(line.split(" ")))
                .distint()
                .count();
        }
        catch(IOException e){}
}


============================
        Optional
============================
```

Optional : better alternative to null (NPE)
problem started in 1965:

Tony Hoare introduced null references back in 1965
while designing ALGOL (billion-dollar mistake)

Problem?

```
class Insurance{
        private String companyName;

        public String getCompanyName() {
                return companyName;
        }

        public void setCompanyName(String companyName) {
                this.companyName = companyName;
        }
}

class Car{
        private Insurance insurance;

        public Insurance getInsurance() {
                return insurance;
        }

        public void setInsurance(Insurance insurance) {
                this.insurance = insurance;
        }
}

class Person{
        private Car car;

        public Car getCar() {
                return car;
        }

        public void setCar(Car car) {
                this.car = car;
        }
}
```

What happens if we try to getInsuranceCompany without setting car to the person...
What is the problem?

=> not each person have a car
=> null do not properly reflect it

Solution How to model the absence of a value?

Use Optional class

Creating Optional objects

Empty optional

```
Optional<Car> optCar = Optional.empty();
```

Optional from a non-null value:

```
Optional<Car> optCar = Optional.of(car);
```
if car is null NPE is immediately thrown

Optional from null
---
create an Optional object that may hold a null value

```
Optional<Car> optCar = Optional.ofNullable(car);
```

If car is null, the resulting Optional object would be empty.

Extracting and transforming values from optionals with map

Pre java 8:

```
String name = null;
if(insurance != null){
name = insurance.getName();
}
```

Java 8:

```
Optional<Insurance> optInsurance = Optional.ofNullable(insurance);
Optional<String> name = optInsurance.map(Insurance::getName);
```

What it does?

=> The map operation applies the provided function to each element of a stream.

=> We can think an optional object as a particular collection of data,
containing at most a single element.

=> If the Optional contains a value, then the function passed as argument to map
transforms that value.

=> If the Optional is empty, then nothing happens

So what to do with

```
public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName();
}
```

Attempt 1:

```
Optional<Person> optPerson = Optional.of(person);
Optional<String> name =optPerson.map(Person::getCar).map(Car::getInsurance).map(Insurance::getName);
```

Will not work

-> The variable optPeople is of type Optional<People>, so it\92s perfectly fine to call the map method.

-> But getCar returns an object of type Optional<Car>

-> This means the result of the map operation is an object of type Optional<Optional<Car>>

Attept 2:use flatMap

```
public class NeedOfOptional {

public static void main(String[] args) {
	Person p = new Person();
	Car c = new Car();
	Insurance ic = null; // new Insurance();

	Optional<Person> pOpt = Optional.ofNullable(p);
	Optional<Car> carOpt = Optional.ofNullable(c);
	Optional<Insurance> icOpt = Optional.ofNullable(ic);

	p.setCar(carOpt);
	c.setInsurance(icOpt);


	System.out.println(getInsureanceCompany(pOpt));
}

public static String getInsureanceCompany(Optional<Person> p) {

	return p.flatMap
	(per -> per.getCar()).flatMap(c -> c.getInsurance()).map(i -> i.getCompanyName()).orElse("not
	found");
}

}
```

Combining two optionals

```
public Optional<Insurance> nullSafeFindCheapestInsurance(Optional<Person> person,
 Optional<Car> car) {

		if (person.isPresent() && car.isPresent()) {
			return Optional.of(findCheapestInsurance(person.get(), car.get()));
		} else {
			return Optional.empty();
		}
}
```

http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html

```
============================
    Collecting data with streams
============================
 What is collect operation?
 ----------------------
 =>collect is a reduction operation

 => just like reduce, that takes as argument various recipes for accumulating
  the elements of a stream into a summary result

 What we can achive using collectors?
 -------------------------------------
            => sum/reducing stream elements to a single value
            => Finding max/min
            => Joining Strings
            => Grouping elements
            => Partitioning elements


 Consider:
 =========
 enum CaloricLevel { DIET, NORMAL, FAT };
 class Dish {
            private final String name;
            private final boolean vegetarian;
            private final int calories;
            private final Type type;

            enum Type {MEAT, FISH, OTHER}

 Data initilizaton:
 ----------------
    Arrays.asList( new Dish("pork", false, 800, Dish.Type.MEAT),
           new Dish("beef", false, 700, Dish.Type.MEAT),
           new Dish("chicken", false, 400, Dish.Type.MEAT),
           new Dish("french fries", true, 530, Dish.Type.OTHER),
           new Dish("rice", true, 350, Dish.Type.OTHER),
           new Dish("season fruit", true, 120, Dish.Type.OTHER),
           new Dish("pizza", true, 550, Dish.Type.OTHER),
           new Dish("prawns", false, 400, Dish.Type.FISH),
           new Dish("salmon", false, 450, Dish.Type.FISH));

 What type of quaries Collect helps:
 -------------------------------------
 1. Dishes grouped by type
 2. Dishes grouped by caloric level
 3. Dishes grouped by type and then caloric level
 4. Count dishes in each groups
 5. Most caloric dishes by type
 6. Sum calories by type

 // Dishes grouped by type
 ----------------------------
 private static Map<Type, List<Dish>> dishesAsPerType(List<Dish> allDishes) {
            return allDishes.stream().collect(Collectors.groupingBy(Dish::getType));

 }

 //Dishes grouped by caloric level
 ----------------------------------
 private static Map<CaloricLevel, List<Dish>> dishesAsPerCalaroficValue(List<Dish> allDishes){
                        return allDishes.stream().collect(
              Collectors.groupingBy(dish -> {
```

```
                    if (dish.getCalories() <= 400) return CaloricLevel.DIET;
                    else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
                    else return CaloricLevel.FAT;
                  } ));
                }
        }


        //Dishes grouped by type and then caloric level
        -----------------------------------------------
        private static Map<Dish.Type, Map<CaloricLevel, List<Dish>>>
        groupDishedByTypeAndCaloricLevel(List<Dish> allDishes) {
           return allDishes.stream().collect(
                Collectors.groupingBy(Dish::getType,
                        Collectors.groupingBy((Dish dish) -> {
                    if (dish.getCalories() <= 400) return CaloricLevel.DIET;
                    else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
                    else return CaloricLevel.FAT;
                    } )
                )
           );
        }

        //Counting dishes in each group
        -----------------------------------------------
        private static Map<Dish.Type, Long> countDishesInGroups(List<Dish> allDishes) {
           return menu.stream().collect(groupingBy(Dish::getType, counting()));
        }

        //Grouping Most caloric dishes by type
        -----------------------------------------------
        private static Map<Dish.Type, Optional<Dish>> mostCaloricDishesByType(List<Dish> allDishes) {
           return menu.stream().collect(
                groupingBy(Dish::getType,
                    reducing((Dish d1, Dish d2) -> d1.getCalories() > d2.getCalories() ? d1 : d2)));
        }

        //Grouping Sum calories with type of Dish
        -----------------------------------------------
        private static Map<Dish.Type, Integer> sumCaloriesByType(List<Dish> allDishes) {
           return menu.stream().collect(groupingBy(Dish::getType,
                summingInt(Dish::getCalories)));
        }


==========================
 Book Application case study
==========================
Domain model:

enum Subject{
        JAVA, DOT_NET, ORACLE;
}
class Author {
        private String name;
        private String lastname;
        private String country;
}
class Book {
        private String title;
        private List<Author> authors;
        private int pages;
        private Subject subject;
```

```
	private int year;
	private String isbn;
}
```

Sample data:

```
	Author author1=new Author("kathy", "seria", "uk");
	Author author2=new Author("abc", "abc", "us");

	Book book1=new Book("head first", 500, Subject.JAVA, 2004, "121ABC");
	book1.getAuthors().add(author1);
	book1.getAuthors().add(author2);
	Author author3=new Author("pqr", "zyx", "uk");

	Book book2=new Book("oracle in action", 400, Subject.ORACLE, 2004, "121ZBC");
	book1.getAuthors().add(author3);

	List<Book> allBooks=Arrays.asList(book1, book2);
```

Problems:

```
// Find books with more then 400 pages
	List<Book> largeBooks = books.
					stream()
					.filter(b -> b.getPages() > 400)
						.collect(Collectors.toList());

// Find all books that are java books and more then 400 pages

		List<Book> largeJavaBooks = books.stream()
				.filter(b -> b.getPages() > 400 && Subject.JAVA == b.getSubject())
				.collect(Collectors.toList());
// We need the top three longest books
	List<Book> top3LongBooks = books.stream()
				.sorted((b1, b2) -> b2.getPages() - b1.getPages()).limit(3)
					.collect(Collectors.toList());
// We need from the fourth to the last longest books
	List<Book> from4thToLastLongBooks = books.stream()
				.sorted((b1, b2) -> b2.getPages() - b1.getPages())
					.skip(3)
						.collect(Collectors.toList());

// We need to get all the publishing years
	List<Integer> publishYear = books.stream().map(b -> b.getYear())
					.distinct()
						.collect(Collectors.toList());

// We need all the authors who have written a book
	List<Author> authors2 = books.stream()
				.flatMap(b -> b.getAuthors()
					.stream())
						.distinct()
							.collect(Collectors.toList());
```

```
// We need all the origin countries of the authors
-------------------------------------------------
        Set<String> getCountriresOfAuthors = books.stream()
                                        .flatMap(b -> b.getAuthors().stream())
                                                .map(a -> a.getCountry())
                                                        .distinct()
                                                                .collect(Collectors.toSet());
// We want the most recent published book.
-------------------------------------------------
        Optional<Book> bookRecent = books.stream()
                                .min(Comparator.comparing(Book::getYear));


// We want to know if all the books are written by more than one author
-----------------------------------------------------------------------
        boolean isAllBooksWrittenBy2Author = books.stream().
                                allMatch(b -> b.getAuthors().size() > 1);

We want one of the books written by more than one author.
-----------------------------------------------------------
                Optional<Book> multiAuthorBook = books.stream()
                                        .filter(b -> b.getAuthors().size() > 1)
                                                .findAny();

// We want the total number of pages published.
-------------------------------------------------
        Integer totalPubPages = books.stream()
                                                .map(b -> b.getPages())
                                                        .reduce(0, (a, b) -> a + b);

// We want to know how many pages the longest book has.
-------------------------------------------------
        Optional<Integer> longestBook = books.stream()
                                        .map(b -> b.getPages())
                                                .reduce(Integer::max);

// We want the average number of pages of the books
-------------------------------------------------
        Double avgPages = books.stream()
                                        .collect(Collectors.averagingDouble(Book::getPages));

 We want all the titles of the books
-------------------------------------
        String title = books.stream()
                                        .map(b -> b.getTitle())
                                                .collect(Collectors.joining(" "));

// We want the book with the higher number of authors?
-------------------------------------------------
        Optional<Book>bookWrittenByHighestAuthors=books.stream().
                 collect(Collectors.maxBy(Comparator.comparing(b->b.getAuthor().size())));

We want a Map of book per year.
-----------------------------------------
                Map<Integer, List<Book>> yearBooks = books.stream()
                                                        .collect(Collectors.groupingBy(Book::getYear));
```

We want a Map of how many books are published per year per SUBJECT

```
Map<Integer, Map<Subject, List<Book>>> map = books.stream()
        .collect(Collectors.groupingBy(Book::getYear),
                Collectors.groupingBy(Book::getSubject));
```

We want to count how many books are published per year.

```
Map<Integer, Long>yearBooks1=books.stream()
        .collect(Collectors.groupingBy(Book::getYear, Collectors.counting()));
```

We want to partition book by hard cover and non hard cover books.

```
Map<Boolean, List<Book>>hardCoverBooks=books.stream()
        .collect(Collectors.partitioningBy(Book::hasHardCover));
```

Refactoring, testing, and debugging

Refactoring steps:

1. Refactoring anonymous classes to lambda expressions
2. Refactoring lambda expressions to method references
3. Refactoring imperative-style data processing to streams

Refactoring: Example

```
//Dishes grouped by caloric level
private static Map<CaloricLevel, List<Dish>> dishesAsPerCalaroficValue(List<Dish> allDishes){
        return allDishes.stream().collect(
  Collectors.groupingBy(dish -> {
     if (dish.getCalories() <= 400) return CaloricLevel.DIET;
     else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
     else return CaloricLevel.FAT;
  } ));
 }
}
```

Refactoring:

=> Extract lambda expression in seperate method in Dish class

```
public class Dish{
    ...
    public CaloricLevel getCaloricLevel(){
        if (this.getCalories() <= 400) return CaloricLevel.DIET;
        else if (this.getCalories() <= 700) return CaloricLevel.NORMAL;
        else return CaloricLevel.FAT;
        }
}
```

2. Now use method reference:

```
allDishes.stream().collect(groupingBy(Dish::getCaloricLevel));
// lambda expression is extracted to method
```

debugging

Stack trace, logging

Stack trace not be that useful

as lambda expression dont have name :(

What happens ?

```
List<Point> points = Arrays.asList(new Point(12, 2), null);
points.stream().map(p -> p.getX()).forEach(System.out::println);
```

Logging information

Want to know what is going on? peek() method

```
List<Integer> result = Stream.of(2, 3, 4, 5)
        .peek(x -> System.out.println("taking from stream: " + x))
            .map(x -> x + 17)
        .peek(x -> System.out.println("after map: " + x))
            .filter(x -> x % 2 == 0)
        .peek(x -> System.out.println("after filter: " + x))
            .limit(3)
        .peek(x -> System.out.println("after limit: " + x))
            .collect(toList());
```

```
------> numbers-----> Map------> filter-------> limit------>collect
 |                 |           |                |                  |
peek              peek      peek              peek               peek
```

==========
Design pattern with java 8
==========
iterator, strategy, execute around method...

execute around method...

Consider : We have a resource class, we want to clean object once we done

Attempt 1: using finalizer()...

```
class Resource{
        public Resource(){
                System.out.println("resource is created.....");
        }
        public void op1(){
                System.out.println("op1......");
        }
        public void op2(){
                System.out.println("op2......");
        }

        @Override
        public void finalize() throws Throwable {
                System.out.println("finilized is called....");
        }
}
```

In main

```
Resource r=new Resource();
r.op1();
r.op2();
```

=> finalize() not guranteed

Attempt 2: create your own cleanup method.... close()

```
class Resource{
	public Resource(){
		System.out.println("resource is created.....");
	}
	public void op1(){
		System.out.println("op1......");
	}
	public void op2(){
		System.out.println("op2......");
	}

	public void close() throws Throwable {
		System.out.println("finilized is called....");
	}
}
```

In main

```
Resource r=new Resource();
r.op1();
r.op2();
```

What if programmer forget to call close()

Attempt 3: Java 7 comes with ARM

```
class Resource implements AutoCloseable {
	public Resource() {
		System.out.println("resource is created.....");
	}

	public void op1() {
		System.out.println("op1......");
	}

	public void op2() {
		System.out.println("op2......");
	}

	@Override
	public void close() {
		System.out.println("clean the resource ....");
	}

}
```

main:

```
try (Resource r = new Resource()) {
				r.op1();
				r.op2();
		}
```

What if programmer is creating resouce without try block

Java dont call your close() method

What to do?

Apply Java 8 execute around pattern

Step 1: mention close() method private

Step 2: mention ctr private

Step 3:Add public static use method that accept an Consumer

```
	public static void use(Consumer<Resource>block){
		Resource resource=new Resource();
		try{
			block.accept(resource);
		}finally{
			resource.close();
		}
	}
```

Final code:

```
----------
class Resource {
	private Resource() {
		System.out.println("resource is created.....");
	}

	public Resource op1() {
		System.out.println("op1......");
		return this;
	}

	public Resource op2() {
		System.out.println("op2......");
		return this;
	}

	private void close() {
		System.out.println("clean the resource ....");
	}

	public static void use(Consumer<Resource> block) {
		Resource resource = new Resource();
		try {
			block.accept(resource);
		} finally {
			resource.close();
		}
	}

}
public class ExecuteAroundPattern {
	public static void main(String[] args) {
		Resource.use(resource -> resource.op1().op2());
	}

}
```

https://www.mkyong.com/java8/java-8-stream-read-a-file-line-by-line/
http://winterbe.com/blog/

```
	========================
	   fork and join example, issues
	========================
```

Fork-Join Framework (Java 7 Stuff)
==========
What is fork join?

=> The framework is an implementation of the ExecutorService interface and provides an easy-to-use concurrent platform in order to exploit multiple processors.

=> useful for modeling divide-and-conquer problems.

=> Recursively break down the problem into sub problems

=> Solve the sub problems in parallel

=> Combine the solutions to sub-problems to arrive at final result

Java library for Fork Join framework

Key classes in the Fork/Join framework:

ForkJoinPool:

=>Most important class in fork join framework
=> It is a thread pool for running fork/join tasks and it executes an instance of ForkJoinTask

ForkJoinTask:

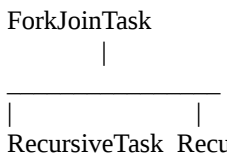
=> lightweight thread-like entity representing a task that defines methods such as fork() and join().

RecursiveTask

=> task that can run in a ForkJoinPool
=> its compute() do computation with returning an result of type V

RecursiveAction

=> is a task that can run in a ForkJoinPool
=> its compute() do computation without returning an result



Algo:
====

```
doRecursiveTask(input) {

        if (the task is small enough to be handled by a thread) {
                compute the small task;
                if there is a result to return, do so
        }

        else {
                divide (i.e., fork) the task into two parts
                call compute() on first task, join() on second task, return combined results
        }
}
```

When to go for FJ framework?

=> FJ framework is suitable if problem fits this description:

1. The problem can be designed as a recursive task where the task
can be subdivided into smaller units and the results can be combined together

2. The subdivided tasks are independent and can be computed separately without
the need for communication between the tasks when computation is in process
(Of course, after the computation is over, you will need to join them together)

How to code?

Steps:

1. Define a task class that extends either RecursiveTask or RecursiveAction
   If (a task returns a result)
      extend from RecursiveTask
   otherwise
      extend from RecursiveAction

2. Override the compute() method in the newly defined task class.

3. The compute() method actually performs the task if the task is small enough to be executed; or
splits the task into subtasks and invoke them.

4. The subtasks can be invoked either by invokeAll() or fork() method
   if (subtask returns a value)
      use fork()
   else
      use join()

   method to get the computed results

5. Merge the results, if computed from the subtasks.

6. Then instantiate ForkJoinPool, create an instance of the task class,
and start the execution of the task using the invoke() method on the
ForkJoinPool instance.

Hello world ?

Now lets try solving the problem of how to sum 1..N where N is a large number

```
class SumOfNUsingForkJoin {
    private static long N = 1000_000; // one million - we want to compute sum
    private static final int NUM_THREADS = 10;
    static class RecursiveSumOfN extends RecursiveTask<Long> {
        long from, to;
        public RecursiveSumOfN(long from, long to) {
            this.from = from;
            this.to = to;
        }
        public Long compute() {
            if ((to - from) <= N / NUM_THREADS) {

                // the range is something that can be handled by a thread, so do summation
                long localSum = 0;

                for (long i = from; i <= to; i++) {
                    localSum += i;
                }
                System.out.printf("\tSum of value range %d to %d is %d %n", from, to, localSum);
```

```
                        return localSum;

                } else {
                        // no, the range is too big for a thread to handle, so fork the computation
                        // we find the mid-point value in the range from..to

                        long mid = (from + to) / 2;
                        System.out.printf("Forking computation into two ranges: "+ "%d to %d and %d to
                        %d %n", from, mid, mid, to);
                        // determine the computation for first half with the range from..mid

                        RecursiveSumOfN firstHalf = new RecursiveSumOfN(from, mid);
                        // now, fork off that task
                        firstHalf.fork();

                        // determine the computation for second half with the range mid+1..to

                        RecursiveSumOfN secondHalf = new RecursiveSumOfN(mid + 1, to);

                        long resultSecond = secondHalf.compute();

                        // now, wait for the first half of computing sum to  complete, once done, add it to the
                        remaining part

                        return firstHalf.join() + resultSecond;
                }
        }
    }

    public static void main(String[] args) {
        // Create a fork-join pool that consists of NUM_THREADS
        ForkJoinPool pool = new ForkJoinPool(NUM_THREADS);

        // submit the computation task to the fork-join pool
        long computedSum = pool.invoke(new RecursiveSumOfN(0, N));

        // this is the formula sum for the range 1..N
        long formulaSum = (N * (N + 1)) / 2;
        // Compare the computed sum and the formula sum
        System.out.printf("Sum for range 1..%d; computed sum = %d, "+ "formula sum = %d %n", N,
computedSum, formulaSum);
    }
}
```

Note: programmatically getting the number of available processors?

```
    System.out.println("processor: "+Runtime.getRuntime().availableProcessors());
```

==========
Basics of Parallle strams
==========

Basics of Parallle strams

What is streams?

-> A stream is not a collection, sequence stream of objects
-> A stream is a abstraction that represent zero or more "values"
-> not (necessarily) a collection: values might not be sorted anywhere
-> not (necessarily) a sequence; order might not matter
-> values, not object; avoid mutation and side effects

Processing Pipeline?

A pipeline consist of :
-> a steram source
-> zero or more intermediate results
-> a terminal operation

```
colllections.stream()         //source
                        .filter(...) //intermediate
                         .map(...)  //intermediate
                           .collect(...) //terminal opeation
```

Parallel steams?

=> Source starts with steam(), parallelStream() or other stream factory

=> can be switch using parallel() or sequential()

=> parallel vs sequential is a property of entire pipeline()

-> can not switch bw parallel and sequential in middle

-> "last one wins"

=> Parallel makes it auto-magically go faster, right ? WRONG

In below example entire stream run sequentially:

```
colllections.stream()         //source
                        .filter(...) //intermediate
                         .parallel()
                         .map(...)  //intermediate
                          .sequential()
                           .collect(...) //terminal opeation
```

Parallel stream consideration/issues?

-> Parallel and sequentail stream should give the same result
-> parallelishm leads to nondeterminisum...bad... need to control it
-> Encounter order vs processing order
    -> for sequential steram they are same
    -> for parallel steam they need not to be same
-> stateless vs stateful: managing side effect
-> accumulation vs reduction
    -> for sequential steam we can use accumulation sum+i=sum
    -> for parallal steam must not use accumulation USE REDUCTION

==========

Parallel data processing and performance

==========

Basics:

Let consider problem to find prime no bw 2 to 1_000_000

```
class PrimeNumbers {
private static boolean isPrime(long val) {
                for(long i = 2; i <= val/2; i++) {
                if((val % i) == 0) {
                return false;
                }
        }
        return true;
}
```

Serial execution :

```
long numOfPrimes = LongStream.rangeClosed(2, 100_000)
                        .filter(PrimeNumbers::isPrime)
                                .count();
System.out.println(numOfPrimes);
```

paralled execution:

```
long numOfPrimes = LongStream.rangeClosed(2, 100_000)
                        .parallel()
                                .filter(PrimeNumbers::isPrime)
                                .count();
```

Note:

=> When we call the stream() method of the Collection class, we will get a sequential stream. When you call parallelStream() method of the Collection class, you will get a parallel stream.

=> isParallel()
check if the stream is sequential or parallel by calling the isParallel() method

Performing Correct Reductions

=> it is important not to depend on global state (Side effect)

Example:

```
class StringConcatenator {
        public static String result = "";

        public static void concatStr(String str) {
                result = result + " " + str;
        }
}
```

Serial execution:

```
String words[] = "the quick brown fox jumps over the lazy dog".split(" ");
Arrays.stream(words).forEach(StringConcatenator::concatStr);
System.out.println(StringConcatenator.result);
```

parallel execution:

```
Arrays.stream(words).parallel().forEach(StringConcatenator::concatStr);
```

Problem?

=> When the stream is parallel, the task is split into multiple sub-tasks and different threads execute it.

=> The calls to forEach(StringConcatenator::concatStr) now access the globally accessible variable result in StringConcatenator class.

=> Hence this program suffers from a race condition problem

Solution:

=> get rid of modifying the global state and keep the reduction localized

=> We can use the reduce() method

Correct way:

```
String words[] = "the quick brown fox jumps over the lazy dog".split(" ");

Optional<String> originalString =(Arrays.stream(words).parallel().reduce((a, b) -> a + " " + b));

System.out.println(originalString.get());
```

Parallel Streams and Performance
---
it is not always the case that the performance with parallel streams is
better than sequential streams.

=> Only if the operations are performed on a significantly
large number of elements, the operations are computationally expensive,
and the data structures are efficiently splittable,
we will see performance improvements with parallel streams; otherwise, execution
with a parallel stream may be slower than with sequential streams!

How to change number of processor used in parallel stream?

=> Under the hood paralled stream use fork/join thread pool

=> default configuration :
number of threads is == number of processors in your machine

=> checking number of processor:
Runtime.getRuntime().availableProcessors()

=> Changing default setting
```
System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "8");
```

or

```
java -Djava.util.concurrent.ForkJoinPool.common.parallelism=8 GetParallelism
```

Understanding encounter order:
---

let consider code:running serially

```
List<String> list=IntStream
                        .rangeClosed(0, 50)
                                .filter(i-> i%5==0)
                                        .mapToObj(i-> String.valueOf(i/5))
                                                .collect(Collectors.toList());

System.out.println(list);
```

let run it parallel way....

```
List<String> list=IntStream
                        .rangeClosed(0, 50)
                                .filter(i-> i%5==0)
                                        .mapToObj(i-> String.valueOf(i/5))
                                                .collect(Collectors.toList());

System.out.println(list);
```

encounter order vs procesing order:

ORDER IS MAINTAINED IN O/P..... HOW TO PROVE.. use peek()

```
List<String>peek=new ArrayList<String>();
    List<String> list=IntStream
            .range(0, 50)
                .parallel()

                .filter(i-> i%5==0)
                    .mapToObj(i-> String.valueOf(i/5))
                        .peek(s->peek.add(s))//multiple thread adding to list!
                        .collect(Collectors.toList());

    System.out.println(list);
    System.out.println("-----processing ----------");
    System.out.println(peek);

ArrayIndexOutOfBoundException:
    non thread ds added and manipulated ..so missing info, null etc
```

Change to synch collection : List<String>peek=Collections.synchronizedList(new ArrayList<String>());

Note: order in which data is processed in paralled stream (by multiple threads)can not predictable ....

O/P: resembly of result is not based on processing order but on postion ...(encounter order)

Accumulation vs Reduction

Let consider adding numbers from 1 to 1_000_000

Sequential: Pre Java 8

```
long sum=0;
    for(long i=0;i<=1_000_000;i++){
        sum=sum+i;
    }
    System.out.println(sum);
```

O/P: 500000500000

Let try streams:

```
long []sum=new long[]{0L};

LongStream.rangeClosed(1, 1_000_000).forEach(i->sum[0]+=i);

System.out.println(sum[0]);
```

Why array?

Let try parallel streams:
---

try parallel() What happens?

What to do? immutability?

```
AtomicLong sum=new AtomicLong(0L);
 LongStream.rangeClosed(1, 1_000_000).parallel()
        .forEach(i->sum.addAndGet(i));

 System.out.println(sum);
```

What is the problem with Accumulation ?

If we are using an variable like sum for accumulation, all threads accessing it to change it.....
to much memory trafic to that variable.....too slow

Reduction : better alternative to accumulation?

```
long sum=LongStream.rangeClosed(1L, 1_000_000)
                    .parallel()
                        .reduce(0, (x,y)->x+y);

System.out.println(sum);
```

Simplified: reduce() to sum is so common java have given sum() method....
```
long sum=LongStream.rangeClosed(1L, 1_000_000)
                    .parallel().sum();

System.out.println(sum);
```

Reduction : Identity (mathmatical concept) remember identity matrix....

-> the starting value of each partition of a parallel reduction
-> become the result if there is no value in the stream
-> it must be right identity value
-> must really identity value (immutable not mutable)

Identity: demo

Consider:
```
static List<String> strings =
    Arrays.asList("one", "two", "three","four",
        "five", "six", "seven", "eight", "nine", "ten");

.....
System.out.println(strings.stream().map(s-> "{"+s+"}").reduce("", (s1, s2)-> s1+s2));
```

Getting same result with parallel stream:

```
System.out.println(strings.stream().parallel().map(s-> "{"+s+"}").reduce("", (s1, s2)-> s1+s2));
```

Lets add wrong identity ..what happens:

```
System.out.println(strings.stream().parallel().map(s-> "{"+s+"}").reduce("xxxxx", (s1, s2)-> s1+s2));
```

Identity value is used many time:
---

xxxx{one}xxxx{two}xxxx{three}xxxx{four}xxxx{five}xxxx{six}xxxx{seven}xxxx{eight}xxxx{nine}xxxx{ten}
(Result may vary..wrong result)

Using stringbuilder?

```
System.out.println(strings.stream()
    .map(string -> new StringBuilder("{" + string + "}"))
    .reduce(new StringBuilder(), (s1, s2) -> s1.append(s2)));
```

Run in parallel

```
System.out.println(strings.stream()
                    .map(string -> new StringBuilder("{" + string + "}"))
```

```
                                .parallel()
                                .reduce(new StringBuilder(), (s1, s2) -> s1.append(s2)));


                // we are mutating identity :(
```

Garbage o/p? mutation be aware.....stringbuilder is not thread safe

What about stringBuffer? so called threadsafe

Even we are mutating in thread safe manner , mutating identity is a bad idea!
sub string of intermediate result is appended may time

Demo: associativity

Consider :

```
static List<String> strings = Arrays.asList("a", "b", "c", "d",
                        "e", "f", "g", "h", "i", "j");


System.out.println(strings.stream().reduce("", (x,y)->x+y));
```

Correct associativity: correct output....

Sequential association

Incorrect associativity: predictable wrong ans

```
System.out.println(strings.stream().reduce("", (x,y)->x+y));
```

parallel association

```
System.out.println(strings.stream().parallel().reduce("", (x,y)->y+x+y));
```

Wrong o/p ;(

==========

Java 8 date and time API

==========

date and time api java 8

---

what is the problem?

=> date api is 20 years old50005016 api introduced with JDK 1.0.
=> original authors of Date API is none other than James Gosling himself

```
public class DateSucks {
public static void main(String[] args) {

Date date = new Date(12, 12, 12);
System.out.println(date);

}
}
```

What it print?
0012-12-12 No

it print:
Sun Jan 12 00:00:00 IST 1913 was WTF???

issues?
---
1. What each 12 means? Is it month, year, date or date, month, year or any other combination
2. Date API month index starts at 0. So, December is actually 11.
3. Date API rolls over i.e. 12 will become January.
4. Year starts with 1900. And because month also roll over so year becomes 1900 + 12 + 1 == 1913.
5. Who asked for time? I just asked for date but program prints time as well.
6. Why is there time zone? Who asked for it? The time zone is JVM\92s default time zone, IST

Java 8 Date Time API

=> domain-driven design principles with domain classes like LocalDate, LocalTime applicable to solve problems related to their speci??c domains of date and time

most imp in the new API are LocalDate, LocalTime, and LocalDateTime
---
LocalDate: It represents a date with no time or timezone.
LocalTime: It represents time with no date or timezone
LocalDateTime: It is the combination of LocalDate and LocalTime i.e. date with time without time zone.

```
LocalDate ld=LocalDate.of(1931, Month.OCTOBER, 15);
System.out.println(ld);

System.out.println(LocalDate.ofYearDay(2015, 1));
System.out.println(LocalDate.ofEpochDay(1));// will give 1970-01-02 as starting value of is 1970-01-01
System.out.println(LocalDate.now());// create current date from the system clock using now static factory method

System.out.println(LocalTime.of(1, 15));
System.out.println(LocalTime.now());
```

==========
Nashorn
==========
Nashorn introduction:
---
jjs
jjs> print('Hello World');

running from java code
```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
engine.eval("print('Hello World!');");
```

from an js file:
```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
engine.eval(new FileReader("script.js"));
```

Invoking Javascript Functions from Java
---
script.js

```
var fun1 = function(name) {
print('Hi there from Javascript, ' + name);
return "greetings from javascript";
};
var fun2 = function (object) {
print("JS Class Definition: " + Object.prototype.toString.call(object));
};
```

calling from java program:

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
engine.eval(new FileReader("script.js"));

Invocable invocable = (Invocable) engine;//cast the script engine to Invocable

//The Invocable
//interface is implemented by the NashornScriptEngine implementation and defines a method
//invokeFunction to call a javascript function for a given name

Object result = invocable.invokeFunction("fun1", "Peter Parker");
System.out.println(result);
System.out.println(result.getClass());
```

Now let's call the second function by passing arbitrary java objects
---

```
invocable.invokeFunction("fun2", new Date());         // [object java.util.Date]

invocable.invokeFunction("fun2", new Person()); // [object com.winterbe.java8.Person]
```

Invoking Java Methods from Javascript
---
defind an static method in a class:

```
static String fun1(String name) {
System.out.format("Hi there from Java, %s", name);
return "greetings from java";
}
```

Now calling it:

```
var MyJavaClass = Java.type('my.package.MyJavaClass');
var result = MyJavaClass.fun1('John Doe');
print(result);
```

How does Nashorn handle type conversion when calling java methods with native javascript types?

The following java method simply prints the actual class type of the method parameter:

```
static void fun2(Object object) {
System.out.println(object.getClass());
}


MyJavaClass.fun2(123);   // class java.lang.Integer

MyJavaClass.fun2(49.99);// class java.lang.Double

MyJavaClass.fun2(true);// class java.lang.Boolean

MyJavaClass.fun2("hi there")// class java.lang.String

MyJavaClass.fun2(new Number(23));// class jdk.nashorn.internal.objects.NativeNumber

MyJavaClass.fun2(new Date());// class jdk.nashorn.internal.objects.NativeDate

MyJavaClass.fun2(new RegExp());// class jdk.nashorn.internal.objects.NativeRegExp
```

MyJavaClass.fun2({foo: 'bar'});// class jdk.nashorn.internal.scripts.JO4

==========
base 64
==========

=> Java 8 now has inbuilt encoder and decoder for Base64 encoding.

3 types of Base64 encoding

Simple - Output is mapped to a set of characters lying in A-Za-z0-9+/.
The encoder does not add any line feed in output, and the decoder rejects any character other than A-Za-z0-9+/.

URL - Output is mapped to set of characters lying in A-Za-z0-9+_.
Output is URL and filename safe.

MIME - Output is mapped to MIME friendly format. Output is represented in
lines of no more than 76 characters each, and uses a carriage return '\r'
followed by a linefeed '\n' as the line separator.
No line separator is present to the end of the encoded output.

imp Classes
---
static class Base64.Decoder
=> This class implements a decoder for decoding byte data
using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.

static class Base64.Encoder
=> This class implements an encoder for encoding byte data using the Base64
encoding scheme as specified in RFC 4648 and RFC 2045.

imp Methods
---

static Base64.Decoder getDecoder()
=> Returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme.

static Base64.Encoder getEncoder()
=> Returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme.

static Base64.Decoder getMimeDecoder()
=> Returns a Base64.Decoder that decodes using the MIME type base64 decoding scheme.

static Base64.Encoder getMimeEncoder()
=> Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme.

static Base64.Encoder getMimeEncoder(int lineLength, byte[] lineSeparator)
=> Returns a Base64.Encoder that encodes using the MIME type base64 encoding
scheme with specified line length and line separators.

static Base64.Decoder getUrlDecoder()
=> Returns a Base64.Decoder that decodes using the URL and Filename safe type base64 encoding
scheme.

static Base64.Encoder getUrlEncoder()
=> Returns a Base64.Encoder that encodes using the URL and Filename safe type base64 encoding
scheme.

Hello world basic:

// Encode using basic encoder

```
byte[] val="java is my love".getBytes("utf-8");

String encodededString = Base64.getEncoder().encodeToString(val);
System.out.println("Base64 Encoded value" + encodededString);
        ====================================
// Decode
byte[] decodedBytes = Base64.getDecoder().decode(encodededString);

System.out.println("Original String: " + new String(decodedBytes, "utf-8"));
```

Hello world encodedUrl:

```
    String base64encodedString = Base64.getUrlEncoder().encodeToString("java is fun".getBytes("utf-8"));
System.out.println("Base64 Encoded String (URL) :" + base64encodedString);
```

Hello world mime type:

```
StringBuilder stringBuilder = new StringBuilder();

for (int i = 0; i < 10; ++i) {
  stringBuilder.append(UUID.randomUUID().toString());
}

byte[] mimeBytes = stringBuilder.toString().getBytes("utf-8");
String mimeEncodedString = Base64.getMimeEncoder().encodeToString(mimeBytes);
System.out.println("Base64 Encoded String (MIME) :" + mimeEncodedString);
```