InfoMover Technologies

its all about technology

Core Java 1.8

Presented By Muhammed Shakir

email : shakir@infomover.io *fb : https://www.facebook.com/infomover*

Functional Programming

- *One of the core features added to Java 1.8 is functional programming.*
- *The ability to write code (functions) as objects. Nothing new but the style has changed*
    - *Anonymous classes out*
    - *Lambda expressions in*
- *Functional programming is a term that means different things to different people.*
- *At the heart of functional programming is thinking about your problem domain in terms of*
    - *immutable values and*
    - *functions that translate between them*

- *Working with functional interfaces*
- *OOTB functional interfaces in 1.8*
- *Your own functional interface*
    - *Providing implementation of functional interface with anonymous class*
    - *Enter lambda - providing implementation of functional interface with lambda expression*
- *About @FunctionalInterface*

- *Create a new project*
- *Get the code of the class mentioned in the footer*
- *Fill in the blank*

- *java.util.function.Function<T, R>*
- *Different types of functional interfaces*
  - *Some functional interface - no parameter but there is a return value*
  - *Some - there is a parameter but no return value*
    - *Some - returning and accepting specific types*
- *Lets explore some of the OOTB functional interfaces : BiFunction, Predicate, UnaryOperator, BinaryOperator*
- *1.8 replete with libraries with APIs taking parameters of type Function*

DIY NEXT

Write lambda expressions for the following functional interfaces in java.util.function package

Introducing & Working With Streams

- *What are streams !*
- *Streams v/s Collections*
- *Getting started with streams*
- *Basic operations with streams*
    - *Filter, Map, Ordering, Parallelism*
- *Common Pattern - Reduce*
- *Finding & Matching*

- *The most important core library changes are focused around the Collection API and its new addition - streams.*

- *Streams allow us to write collections-processing code at a higher level of abstraction.*

- ***External iteration** over a collection is taking an iterator and iterating over it*

- ***Internal iteration is using a Stream** to iterate over the collection. It does not return an iterator to your code. It is the stream the iterates over the collection and hence - internal iteration*

Streams	Collections
Streams do not contain data. It is provides a reference that iterates over the collection internally	*Collections are containers of data*
Streams can process the elements of a given Collection in parallel.	*The default iterator of Collection is by sequential only.*
Streams does not allow modifying the underlying data-structure. It provides rich API to process the data - filter, map, sort, reduce etc.	*Collections do not provide a rich API to process the data. The API(s) are rich in traversing, adding, removing, getting. The implementations may be different*

DIY NEXT

Go to Collection interface and check - it has a new default methods called stream()& parallelStream. Lets get started

- Unfold the expression into code snippet
- Write an SOP and run the program
- After filter, add a map to convert name to upper case - use a Function<? extends String, ? extends R>
- Write an SOP in filter too and run the code
- Change the order of operations and see results - First do mapping and then filter.

- Each element is processed one after the other
- The element ordering is dependent on how the source provides data
- Set may provide data in a different order

DIY NEXT

DIY

- Add Thread.currentThread().getName() to the SOP(s)
- Run the code and make note of the thread name
- Change from stream() to parallelStream() and now run. Make note of thread name now
- Try to process the stream again and see the exception you get

DIY NEXT

- You will find the this package with model classes - streams.com.infomover.training.java8.model
- Understand the Employee class, HealthPlan & Dependent.
- Make not of a class called HealthData along with its static methods
- We will be making heavy use of these classes in this training.

- Find employee with least dependent
- Make note of the get method. min is actually terminal
- Min returns an Optional. Optional value is like an alien. We will discuss more later

- A common pattern appears. There is boiler plate code here
- It can be used for any object. Only the condition changes (the function of comparing)

```
List<Employee> emps = HealthData.employeeList;

Employee emp = emps.get(0);
for (Employee e : emps) {
    if (e.getDependentList().size() <
                emp.getDependentList().size()) {
        emp = e;
    }
}
```

```
Object accumulator = initialValue;
for(Object element : collection) {
    accumulator = combine(accumulator, element);
}
```

DIY

The things that differ between implementations of this pattern are the

- *initialValue and the*
- *combine function.*

```
Object accumulator = initialValue;
for(Object element : collection) {
    accumulator = combine(accumulator, element);
}
```

```
int sum = Stream.of(1, 2, 3)
               .reduce(0, (acc, element) -> acc + element);
```

*Run : **Ch2App3CommonPatternAppearsEnterReduce.java** and see the results*

DIY NEXT

- *Putting functions together. Once again refer to the HealthData model and check the fact that employees have health plans and health plan has a name*
- *Any health plan name that starts with "Compre" is considered as comprehensive health plan*
- *A health plan is associated to State*
- *We want to get all the states that offer comprehensive health plans*

- *Let us get all the name of all dependents that are of age greater than 15 of all given employees. Note that dependents are associated to Employee*
- *Code snippet 1 : check the age of the dependent and add the name of dependent to the Set*
- *Code snippet 2 : This time you are using stream but still doing the same thing. Note that what you are doing with legacy kind of code is a good candidate for **filter***
- *Code snippet 3: Use a) filter b) map c) for each. Note that using for each on dependents again is a good candidate for **flatMap***
- *Code snippet 4 : Execute chain of operations on Employee stream to flatMap, filter, map and then collect*

```
String[][] data = new String[][]{{"a", "b"}, {"c", "d"}, {"e", "f"}};

//Stream<String[]>
Stream<String[]> temp = Arrays.stream(data);

//filter a stream of string[], and return a string[]?
Stream<String[]> stream = temp.filter(x -> "a".equals(x.toString()));

stream.forEach(System.out::println);
```

empty result

```
//Stream<String>, GOOD!
Stream<String> stringStream = temp.flatMap(x -> Arrays.stream(x));

Stream<String> stream = stringStream.filter(x -> "a".equals(x.to

stream.forEach(System.out::println);
```

will print “a”

- *In java it will always be a List<Integer> and not List <int>*
- *Integer is an object - although an int takes 4 bytes of memory, an Integer takes much more as it is an Object*
- *Arrays of numbers is expensive, as each element of a primitive array is just the size of the primitive, while each element of a* ***boxed*** *array is actually an in-memory pointer to another object on the Java heap.*
- *There is also a* ***computational overhead*** *when converting from a primitive type to a boxed type, called boxing, and vice versa, called unboxing*
- *Algorithms that perform lots of numerical operations, the cost of boxing and unboxing combined with the additional memory bandwidth used by allocated boxed objects can make the code* ***significantly slower.***

*Only the **int, long, and double** types have been chosen as the focus of the primitive specialization implementation in Java 8 because the impact is most noticeable in numerical algorithms*

*If the return type is a primitive, the interface is prefixed with To and the primitive type, as in **ToLongFunction***

*If the argument type is a primitive type, the name prefix is just the type name, as in **LongFunction***

*If the higher-order function uses a primitive type, it is suffixed with To and the primitive type, as in **mapToLong**.*

IntStream, LongStream & DoubleStream


```
// 2nd argument is IntUnaryOperator
IntStream iStream= IntStream.iterate(0, (x -> x + 1));
iStream.limit(50).forEach(System.out::println);

LongStream lStream = LongStream.of(12, 13, 14, 17);
DoubleStream dStream = DoubleStream.of(12.2, 13.4, 15.5, 17.5);
```

DIY NEXT

Get the statistics of age of all Dependents of all Employees

The "in IntegerBiFunction" function gets executed following the rule that if there are several possible target types, the most specific is inferred

```
public class Ch3App10verloadResolution {

    public static void main(String[] args) {
        overloaded((x, y) -> x + y);
    }

    static void overloaded(IntegerBiFunction lambda) {
        System.out.println("in IntegerBiFunction");
    }

    static void overloaded(BinaryOperator<Integer> lambda) {
        System.out.println("in BinaryOperator");
    }
}

interface IntegerBiFunction extends BinaryOperator<Integer> {


}
```

BinaryOperator has min & max functions

@FunctionalInterface

Just the annotation to make sure that you accidentally do not define 2 abstract methods in an interface that is supposed to be functional

- *In 1.8 Collection has new methods like stream() and parallelStream()*
- *This means that if you had your own implementation extending from ArrayList, it wont compile in 1.8 unless you implement stream() and parallelStream()*
- *Enter default methods. Now interfaces can have default implementations of methods. This helps a big way in binary compatibility*
- *Check Collection class and default implementations of the aforementioned two methods*

- *If interface Child extends from Parent and Child has an overriding interface. Now if a class implements Child, the default of Child will execute*
- *If the class overrides the implementation of default in interface then class wins*
- *Suppose we had a custom list implementation called MyCustomList and had implemented a custom addAll method, and the new List interface provided a default addAll that delegated to the add method. If the default method in class wasn't guaranteed to be overridden by this addAll method, we could break the your custom implementation.*

Run the through the class and check the code

Implementation of DentalInsurance gets executed

```
interface AccidentInsurance {

    default public int getMaxInsuranceAmount() {
        // in real-life, based on some business logic
        return 100000;
    }
}

interface DentalInsurance extends AccidentInsurance {

    default public int getMaxInsuranceAmount() {
        // in real-life, based on some business logic
        return 5000;
    }
}

class TheHealthInsurance implements AccidentInsurance, DentalInsurance {


}
```

Duplicate default methods

```
interface AccidentInsurance {

    default public int getMaxInsuranceAmount() {
        // in real-life, based on some business logic
        System.out.println("accident insurance");
        return 100000;
    }
}
```

```
interface DentalInsurance {

    default public int getMaxInsuranceAmount() {
        // in real-life, based on some business logic
        System.out.println("dental insurance");
        return 5000;
    }
}

class TheHealthInsurance implements AccidentInsurance, DentalInsurance {


}
```

Compilation error

Implementation of default method must in case of duplicates

```
interface AccidentInsurance {

    default public int getMaxInsuranceAmount() {
        // in real-life, based on some business logic
        System.out.println("accident insurance");
        return 100000;
    }
}
interface DentalInsurance {
    default public int getMaxInsuranceAmount() {
        // in real-life, based on some business logic
        System.out.println("dental insurance");
        return 5000;
    }
}
class TheHealthInsurance implements AccidentInsurance, DentalInsurance {

    @Override
    public int getMaxInsuranceAmount() {

        // It can very well be DentalInsurance.super.getMaxInsuranceAmount()
        return AccidentInsurance.super.getMaxInsuranceAmount();
    }
}
```

Enhanced super syntax

1. *Any class wins over any interface. So if there's a method with a body, or an abstract declaration, in the superclass chain, we can ignore the interfaces completely.*

2. *Subtype wins over supertype. If we have a situation in which two interfaces are competing to provide a default method and one interface extends the other, the subclass wins.*

3. *No rule 3. If the previous two rules don't give us the answer, the subclass must either implement the method or declare it abstract.*

- *Stream is an interface and you have* ***Stream.of*** *: Another new language change that has made its way into Java 8, primarily in order to help library developers*

- *When there's a good semantic reason for a method to relate to a concept,* ***it should always be put in the same class*** *or interface rather than hidden in a utility class to the side. This helps structure your code in a way that's easier for someone reading it to find the relevant method.*

```
interface HealthInsurance {

    public void calcualteInsurance() ;

    public static List<String> availablePolicies() {
        // some implementation
        return Collections.emptyList();
    }


}
```

```
System.out.println(Optional.empty().isPresent());
Optional.empty().ifPresent(x -> System.out.println(x));
System.out.println(Optional.of("Some Value").get());
String someString = null; // put Farhan and check
Optional alsoEmpty = Optional.ofNullable(someString);

System.out.println(alsoEmpty);
System.out.println(alsoEmpty.orElse("Shakir"));
System.out.println(alsoEmpty.orElseGet(() -> "Value provided by supplier"));
```

- *Change the value of **strings** to new ArrayList<>() and see the exception that you get*

- *Make changes to the way the maxLenString is used in SOP to avoid exception.*

Types of method references

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

DIY NEXT

Follow the instructions as illustrated in the class

- *Order in which stream processes each element in the source is dependent on source.*

- *Elements of ArrayList will be accessed in an order*

- *Elements of HashSet may not be in order*

DIY

```
List<Integer> numbers = Arrays.asList(10, 10, 10, 412, 322, 112);
```

- *Convert List to Set using Collectors.toSet() and print*
- *Convert to Set using Collectors.toCollection(TreeSet::new) and print*
- *Convert to Map<Integer, Integer> using Collectors.toMap(Function<T, U> keyMapper, Function<T, U> valueMapper) and print*

```
Stream<Employee> employees = HealthData.employees;
```

- *Get the employee with max Dependents using collect(Collectors.maxBy)*
- *Get the average number of Dependents using collect(Collectors.averagingInt) to get the average age of dependents*

DIY NEXT

Partition by employee that has dependents and those that does not have dependents.

- *You will Get a Map<Boolean List<Employee>> the result will be as follows*
- *false = [List of employees with no dependents]*
- *true = [List of employees that has dependents]*

Get employees grouped by their Primary HealthPlan in a Map<HealthPlan, List<Employee>>

- Use Collectors.groupingBy and pass a Function<? extends Employee, ? extends HealthPlan>

Get list of employees by their age groups 25 to 35 and 35 to 45

Join the strings. Nothing can be simpler that this

- *java.util.stream.**Collector<T, A, R>** is an interface that represents a mutable reduction operation*
  - *T is the input type*
  - *A is the mutable accumulation type*
  - *R is the result*
- *java.util.stream.**Collectors** provides implementation of many mutable reduction operations*
- *Accumulates input elements into a mutable result container*
- *Concatenating strings using a `StringBuilder`; computing summary information about elements such as sum, min, max, or average*

Lets understand the execution of Collectors

```
// Run this code and see the results
Set<Integer> set =
        Stream.of(100, 2, 3, 4, 5, 6, 7, 8, 9)
            .filter(x -> {
                System.out.println("filtering : " + x);
                return x <= 100;
            })
            .map(x -> {
                System.out.println("    *** mapping");
                return x + 10;
            })
            .collect(Collectors.toCollection(TreeSet::new));

System.out.println("Set : " + set);
```

Change from TreeSet to MyTreeSet and then see the results

- *Count the number of employees by primary health plan instead of getting Map<HealthPlan, List<Employee>> counting is actually a collector - Collectors.counting()*

- *Get the Set of Employee last names grouped by Health Plan (Do it the neater way - using groupingBy(keyExtractor, mapper)*

  - *keyExtractor will provide the key for grouping*

  - *mapper will convert Employee to lastName of employee and collect them all in a Set*

  - *What you finally get is a Map<HealthPlan, Set<String>> where key is the health plan and value is set of all employees who have opted for that health plan*

- *We will go through a 4 step process of understanding - how to write custom Collectors*
- *The objective is not to write a complex Collector but understand the structure of Collector and standard mechanism of implementing the same*
- *Lets us start with **Ch4App6RefactoringAndCustomCollectorsPart1.java***
    - *In java 1.7 we would concatenate string with a prefix & suffix while running through the loop as shown in code*
    - *In 1.8 we can use a map to map employee objects to employee names and then run forEach to do the same*
- *We are essentially **REDUCING** the stream to one single String.*

- *Code snippet 1 : shows how to use Stream.reduce(BinaryOperation<String> accumulator) - 1 parameter*

  returns an Optional<String> assuming the stream is of String

- *Code Snippet 2: shows how to use Stream.reduce(String identity, BinaryOperator<String> accumulator) - 2 parameters*

  This returns a String (& not Optional) as the identity can be used to infer that there will be some value. Even if null is passed identity, it is converted to String

- *Code Snippet 3 : shows how to use Stream.reduce(U identity, BiFunction<? super String, U>, BinaryOperator<U> combiner) - 3 arguments*

  Here the objective is to return a different type other than String. U represents that other data type

Code Snippet 4

- Use a map to map Employee to employee name
- Use a reduce operation with 3 arguments as shown in Code snippet 3
- Prefix & Suffix “[“ & “]” respectively and print the reduced String

StringCombiner.java

- *Constructor : Initialize the container…. now your StringCombiner has a private StringBuilder. The StringCombiner is the container*
- *add : This add function provides the accumulator functionality. It adds the word in the String to the StringBuilder*
- *merge : This function plays the role of a Combiner that merges the two containers*

Ch4App6RefactoringAndCustomCollectorsPart3.java

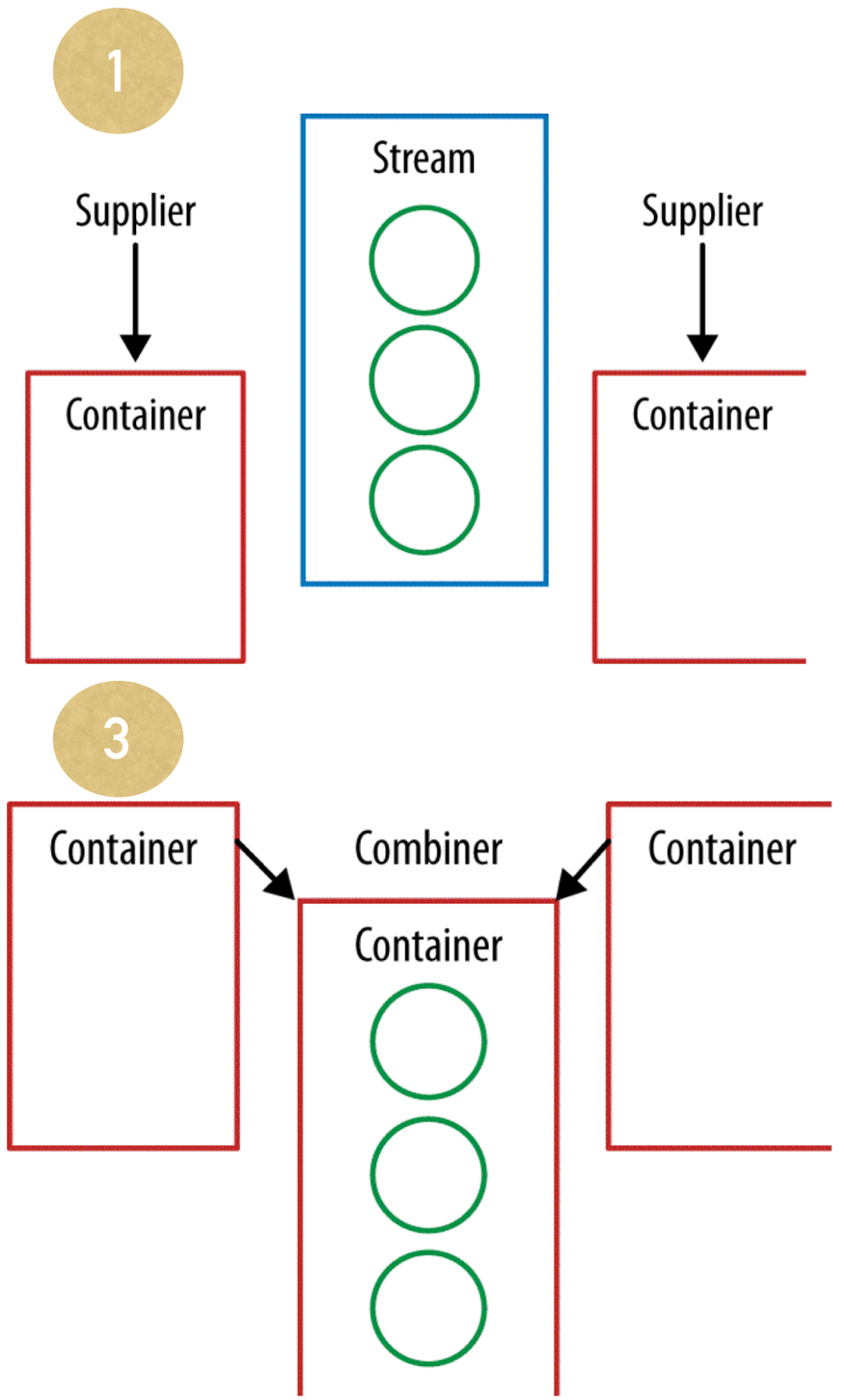
- *Use the constructor of StringCombiner in place of identity*
- *Use StringCombiner::add in place of accumulator*
- *Use StringCombiner::merge in place of combiner*
- *The return value of this reducer happens to be StringCombiner*
- *Fire toString method on combiner to get the reduced String*

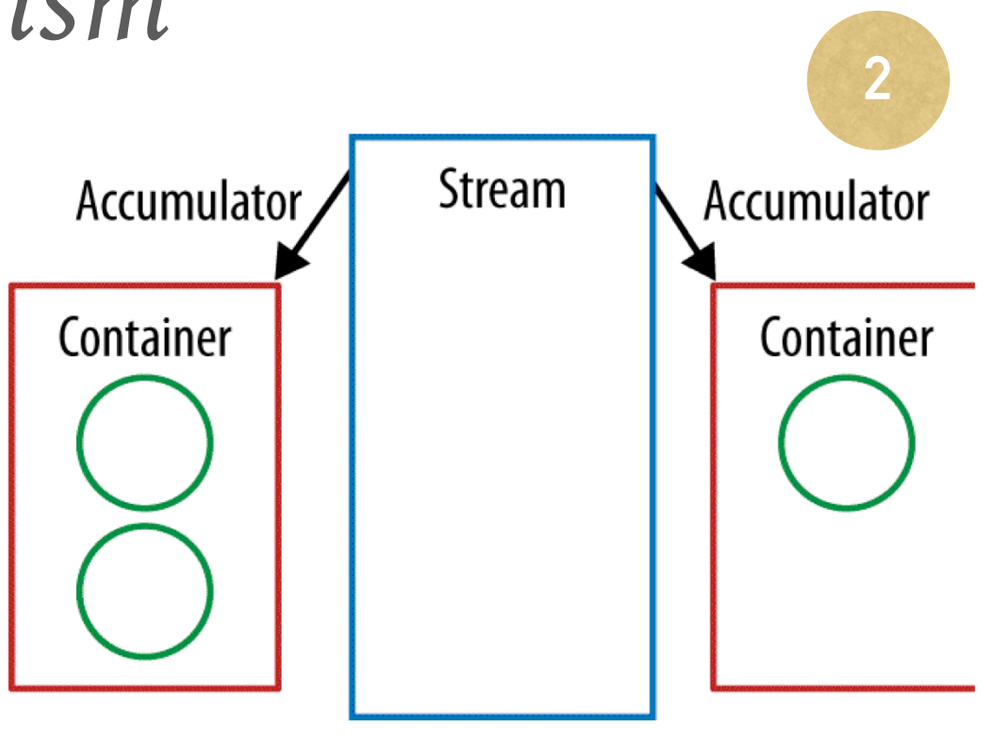
Ch4App6RefactoringAndCustomCollectorsPart3.java

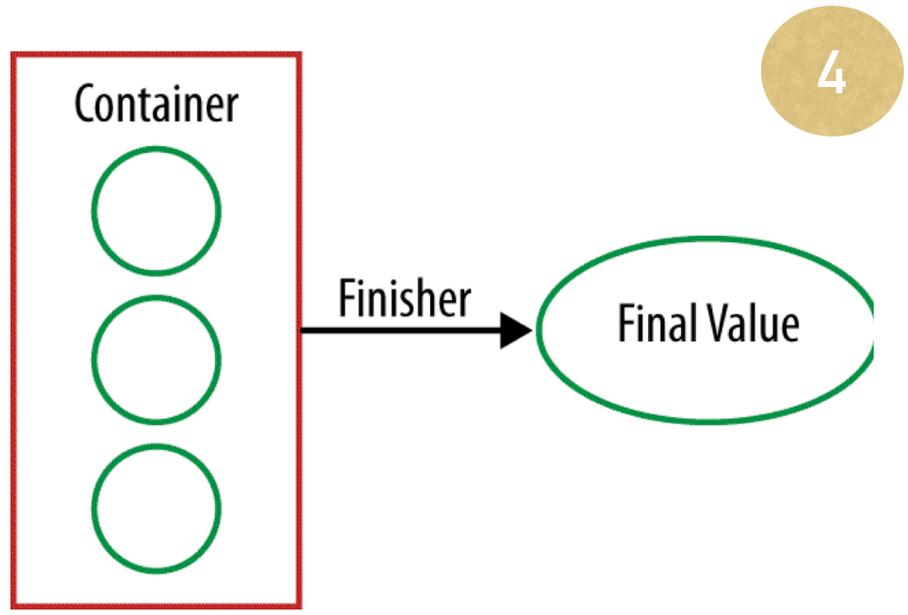
- *Now the real fun - Go back to StringCombiner and put SOP in merge function and tell me whether it executes at all in the very first place*
- *Now convert the stream to parallel and tell me what happens*
- *Now it executes the merge (**if** there is no exception)*
- *So whats happening here ???*

Stay Tuned

Parallelism







MyStringCollector.java

In order for Stream to work elegantly with parallelism as well, we must create a custom java.util.stream.Collector as expected by the Stream and implement the following methods

- *supplier() : a factory method that returns the container of the accumulated objects*
- *accumulator() : a function that returns BiConsumer. In our case BiConsumer<StringCombiner, String>. This function provides the accumulation logic*
- *combiner() : especially for parallelism to add the objects of one container (on the left) to another container (on the right) as shown in figures of previous slide*
- *finisher() : a function that transforms the combined objects into one single String*
- *characteristics() : returns characteristics that can be used by Stream to determine certain properties of Collector*

Ch4App6RefactoringAndCustomCollectorsPart4.java

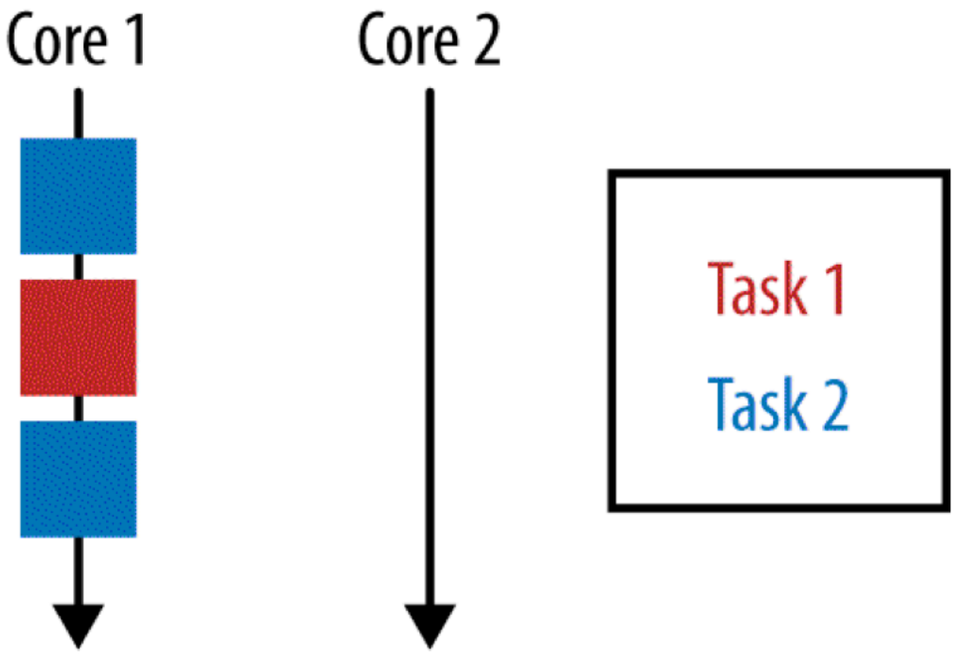
Use the MyStringCollector as the collector and see the results.
Yes - with Parallelism !!!

Take a look at Map.computeIfAbsent, Map.computeIfPresent, Map.compute…

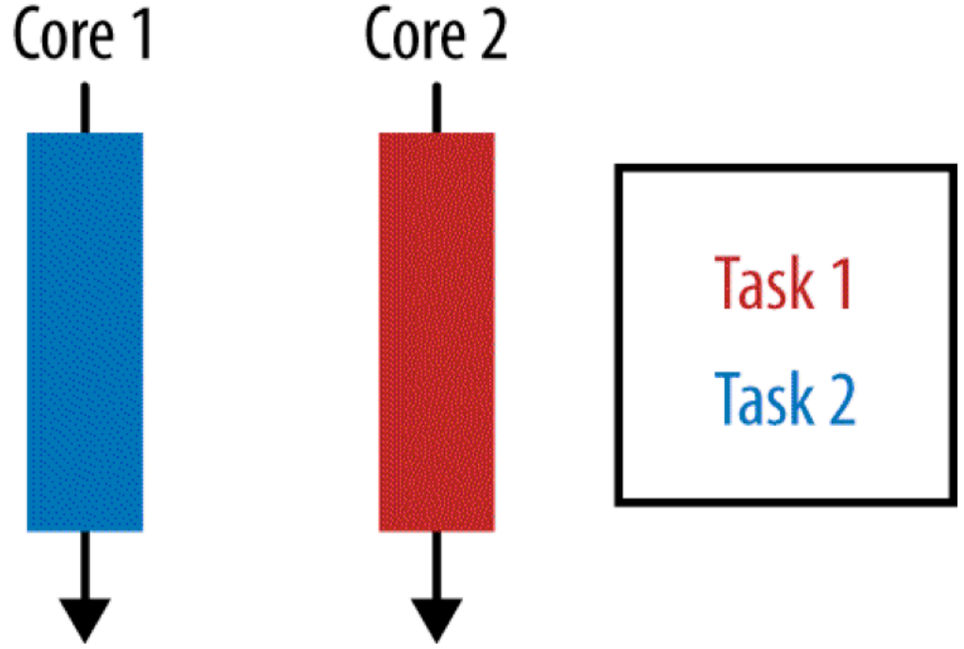
computeIfAbsent can help build cache with clean code (no checking if null)

- *Concurrency arises when two tasks are making progress at overlapping time periods.*
- *Parallelism arises when two tasks are happening at literally the same time, such as on a multicore CPU.*

Concurrent but not Parallel

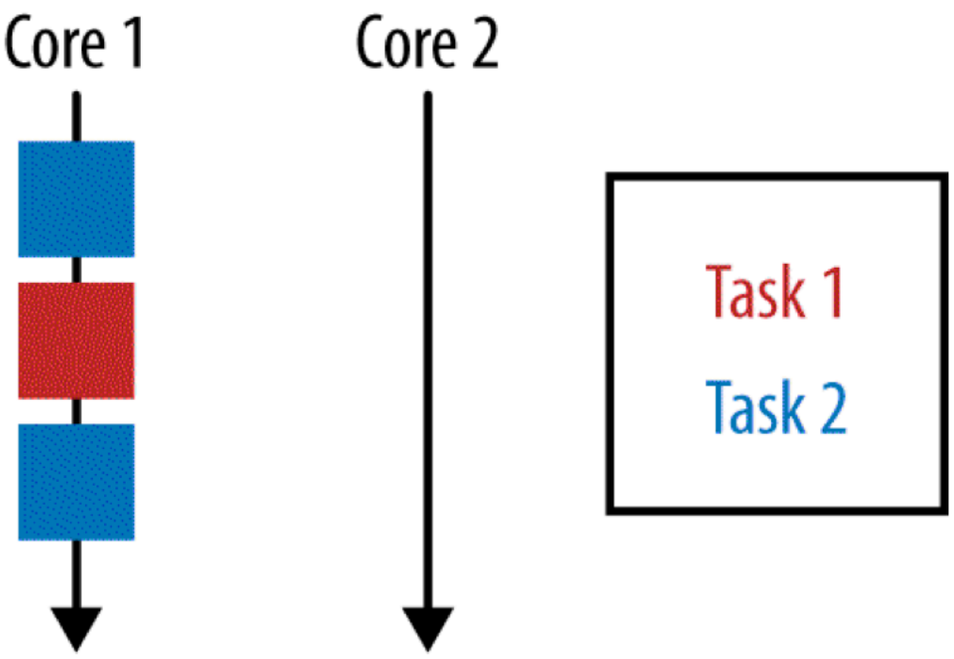


Parallel and Concurrent

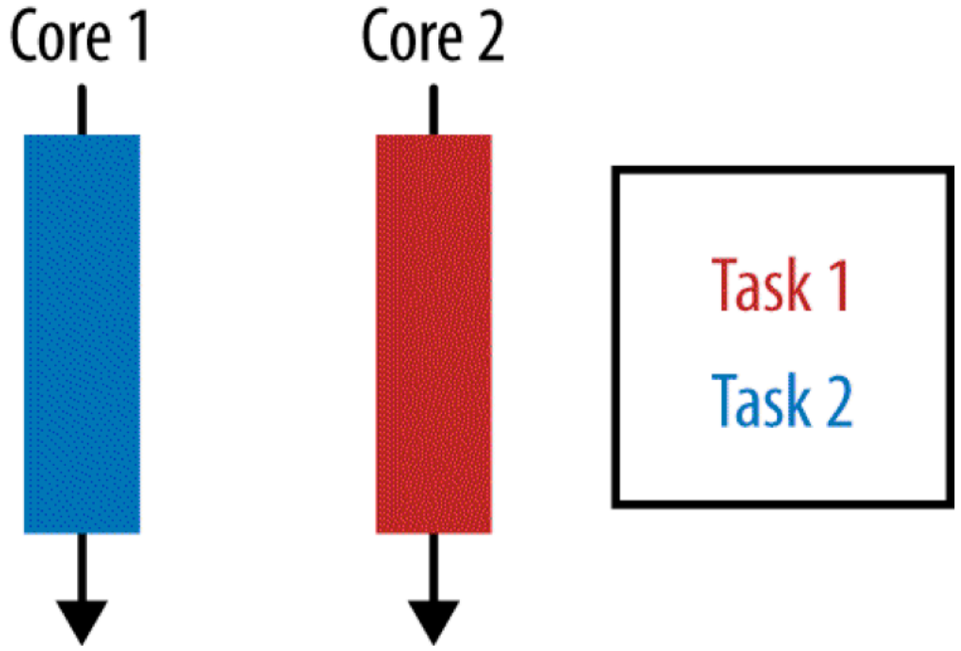


- *To complete the work faster*
- *To make efficient use of all the cores on the machine*
- *To get max throughput*

Concurrent but not Parallel



Parallel and Concurrent

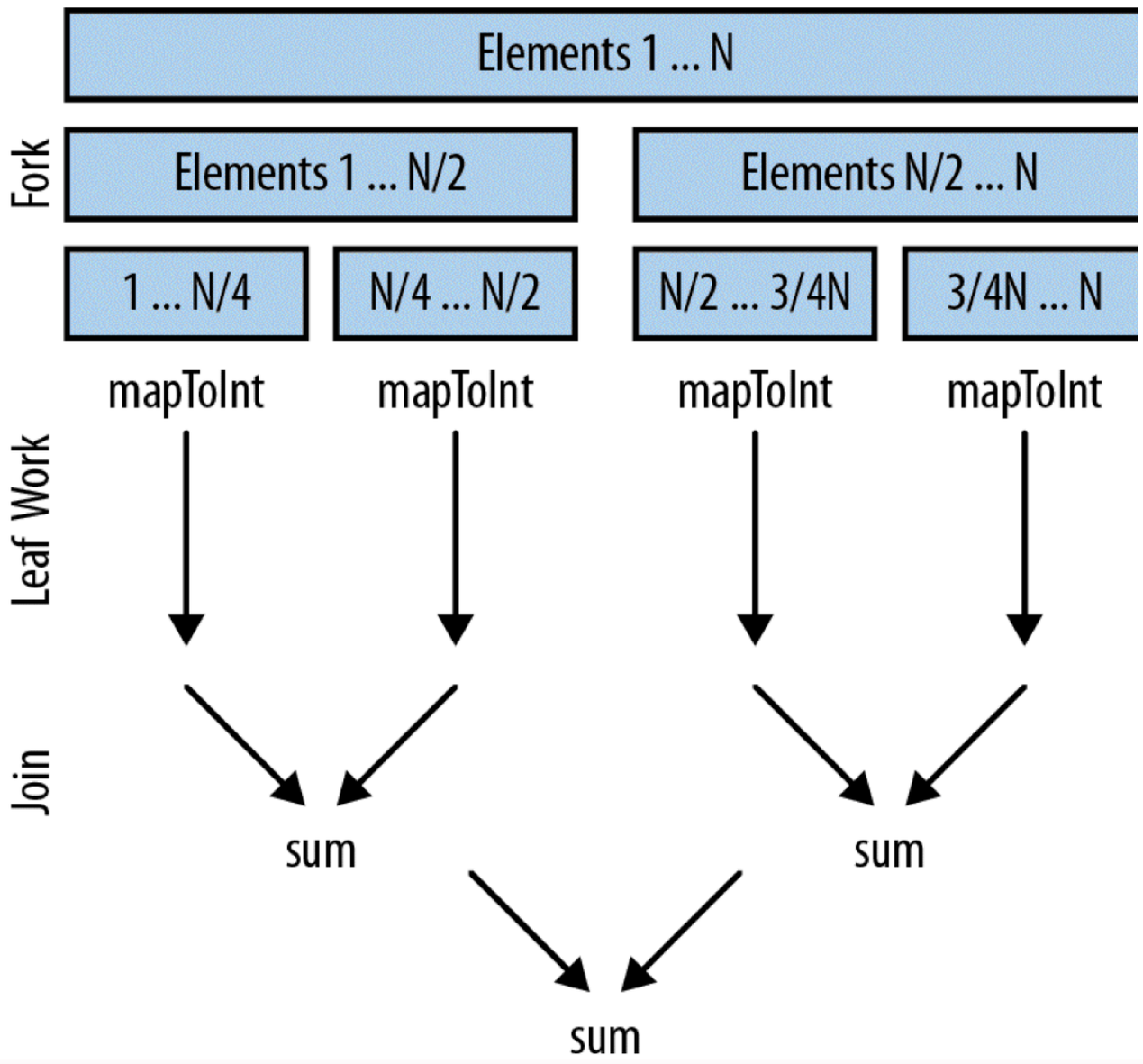


- *Collection.parallelStream()*
- *Stream.parallel()*
- *Uses Forkjoin.commonPool()*
- *The effects of the parallelStream must be as same as sequential*

3 important factors that influence parallel streams

- *Data : small data sets wont yield much of benefit. There is overhead, split, merge, context switch etc.*
- *Packing : Primitives are faster to operate upon with much less overhead*
- *Number of cores : Usually you do have many cores on each system*

Under the hood, parallel streams back onto the fork/ join framework. The fork stage recursively splits up a problem. Then each chunk is operated upon in parallel. Finally, the join stage merges the results back together.



Pretty Simple

```
double[] values = new double[5];

Arrays.parallelSetAll(values, i -> values.length - i);
DoubleStream.of(values).forEach(System.out::println);

System.out.println(" **** ");
Arrays.parallelSort(values);
DoubleStream.of(values).forEach(System.out::println);
```

- *With threading model, every time you want to write some data to socket, you would block the thread that you're running on.*
- *This approach to I/ O, called blocking I/ O, is fairly common but you will have to start a significant number of threads on your server in order to service them*
- *Nonblocking I/ O — or, as it's sometimes called, asynchronous I/ O — the methods to read and write data to your chat clients return immediately.*
- ***The actual I/ O processing is happening in a separate thread, and you are free to perform some useful work in the meantime.***

- *To build up complex sequences of concurrent operations is to use what's known as a Future.*
- *Futures cannot be explicitly completed by another thread*
- *Futures cannot be composed elegantly into chain of dependent operations*
    - *For e.g : How will you pass the result of one future to the other without the pyramid doom*

- *runSequentially : uses Stream*
- *useParallelStream : uses parallelStream*
- *useCompletableFuture : uses CompletableFuture with Forkjoin.commonPool()*
- *useCompletableFutureWithExecutor : uses custom executor service with 10 threads for 10 tasks. Completes faster*

- *Create one CF with simple supplier task which takes 3 seconds*
- *Create another CF that would take 5 seconds*
- *Create stream of CFs and execute them such that the results are joined and one does not wait for the other. There can be more than 2 CFs*
- *The total time take will be well within 5.5 seconds*

- *Create one CF called cf1 with simple supplier task which takes 3 seconds*
  - *Create a CF called cf3 that = cf1.thenCompose(x -> CompletableFuture.supplyAsync(() -> {x + " ~ " + " cf2"});*
  - *cf3.join to get the results. Take the time stamp and see how much time it takes*
- *thenCompose function executes the CFs sequentially because cf2 may be dependent on the results of cf1.*
- *How will the result of cf1 will be used (composed) by cf2, is upto the implementation of cf2 Supplier*
  - ***cf2 does not kick off till the time cf1 is not completed***
  - *cf2 and cf1 may run in different threads but still cf2 will start only after cf1 is completed with its results*
  - *So essentially this is sequential as cf2 is dependent on the value of cf1*
  - *The total time taken for execution is 8+ seconds*

- *Create one CF called cf1 which takes 3 seconds*
- *Create another CF called cf2 which takes 5 seconds*
- *thenCombine as follows : cf3 = cf1.thenCombine(cf2, (x, y) -> x + " ~ " + y)*
- *Note that you are combining the results*
  - ***cf2 and cf1 executes in parallel***
  - *cf2 and cf2 will run in different threads*
  - *The total time taken for execution is ~5 seconds*

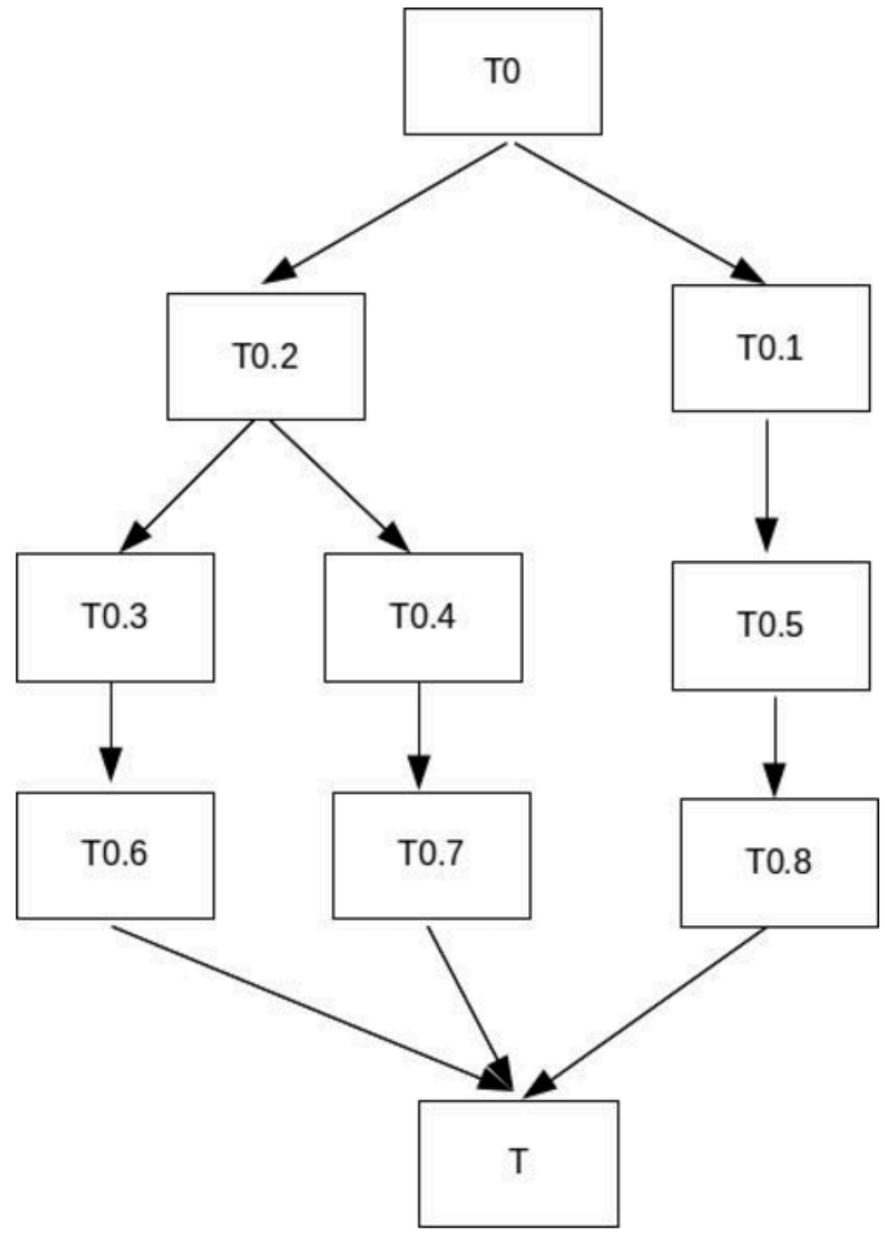
Using CompletableFuture.allOf

- *The concept behind a CompletableFuture can be generalized from single values to complete streams of data using reactive programming.*
- *Reactive programming is actually a form of declarative programming that lets us program in terms of changes and data flows that get automatically propagated.*
- *You can think of a spreadsheet as a commonly used example of reactive programming. If you enter = B1 + 5 in cell C1, it tells the spreadsheet to add 5 to the contents of cell B1 and put the result in C1.*
- *In addition, the spreadsheet reacts to any future changes in B1 and updates the value in C1.*
- *The RxJava library is a port of these reactive ideas onto the JVM.*

- *ForkJoinPool is the framework introduced in 1.7*
- *Works with its own thread pool with number of threads equal to cores*
- *Work-Stealing algorithm at the heart of it*
- *There are 2 types of Tasks - RecursiveTask and RecursiveAction*
- *RecursiveTask returns a value - public T compute()*
- *RecursiveAction does not return a value - public void compute()*

Ch7App0FolkJoinPoolRecurisveTask.java

- *An array of String to be concatenated while converting each element to uppercase*
- *Split the tasks till the time the length of array is greater than THRESHOLD and invoke them*
- *If the length is lesser than actually compute (don't split)*



- RecursiveAction does not return a value
- A good opportunity to see memory visibility related issues
- As such don't use synchronized in compute. It defeats the purpose of parallelism - contention will be very high

- *More often then not parallelism is to work with data*
- *1.8 provides splitting of data using a new interface added to java.util is the Spliterator*
- *Collection interface has been updated to include a new spliterator()*
- *Allow parallel processing of different parts of a Collection*
- *To support **parallel** traversal of the suitably partitioned parts of a Collection*
- *FJP can be used with SplitIterator*

- The trySplit method provides the required behavior. try because it may fail to split and return null
- estimateSize returns the estimated elements that will be available when traversing

- After splitting the elements may be processed in different threads.
  - The source must not be changed
  - To help - trySplit() returns a 'late-binding' and 'fast-fail' Spliterator
    - late-binding binds the elements to the *Spliterator* on first split, first traversal or first query for estimated size
    - NOT at the time of the creation of the *Spliterator*

Run the code and see the results

- *LongAdder uses very clever technique to be lock-free and still remain thread-safe.*
- *AtomicLong can be a bottleneck. This uses a compare-and-swap operation, which – under heavy contention – can lead to a lot of wasted CPU cycles*
- *With increment() method, the implementation keeps an array of counters that can grow on demand.*
- *When more threads are calling increment(), the array will be longer*
- *LongAdder is a very efficient way to increment a counter from multiple threads*

```
LongAdder counter = new LongAdder();
    ExecutorService executorService = Executors.newFixedThreadPool(8);

    int numberOfThreads = 4;
    int numberOfIncrements = 100;

    //when
    Runnable incrementAction = () -> IntStream
      .range(0, numberOfIncrements)
      .forEach((i) -> counter.increment());

    for (int i = 0; i < numberOfThreads; i++) {
        executorService.execute(incrementAction);
    }

    //then
    executorService.awaitTermination(500, TimeUnit.MILLISECONDS);
    executorService.shutdown();

    System.out.println(counter.sum() + " -- " + (numberOfIncrements * numberOfThreads));
```

- *Allows us to implement a lock-free algorithm in a number of scenarios*
- *LongAccumulator can be created by supplying the LongBinaryOperator and the initial value to its constructor*
- *LongAccumulator uses the **compare-and-swap** implementation*
- *It executes an action defined as a LongBinaryOperator, and then it checks if the **previousValue changed**. If it was changed, the action is **executed again with the new value**. If not, it succeeds changing the value that is stored in the accumulator*
- *LongAccumulator will work correctly if we supply it with a commutative function where the **order of accumulation does not matter***

```
ExecutorService executorService = Executors.newFixedThreadPool(8);
LongBinaryOperator sum = Long::sum;
LongAccumulator accumulator = new LongAccumulator(sum, 0L);
int numberOfThreads = 4;
int numberOfIncrements = 100;

// when
Runnable accumulateAction = () ->
        IntStream.rangeClosed(0, numberOfIncrements)
                .forEach(accumulator::accumulate);

for (int i = 0; i < numberOfThreads; i++) {
    executorService.execute(accumulateAction);
}

// then
executorService.awaitTermination(500, TimeUnit.MILLISECONDS);
executorService.shutdown();
```

Only the data types are different. Double has add()method

- *The **Files** class is one of the primary entry points of the java.nio.file package.*
- *Rich set of APIs for reading, writing, and manipulating files and directories*
- *The Files class methods work on instances of **Path** objects*
- *Checking a file or a directory.*

- *Path instance represents a file or a directory on the file system.*
- *Whether that file or directory it's pointing to exists or not, is accessible or not can be confirmed by a file operation.*
- *The file system API provides single line operations for creating files*
- *To create a regular file, we use the createFile API and pass to it a Path object representing the file we want to create.*
- *All the name elements in the path must exist, apart from the file name, otherwise, we will get an IOException:*

- *Code Snippet 1 : Basic file & directory operations*
- *Code Snippet 2 : Creating files - Files.createFile*
- *Code Snippet 3 : Creating directories - Files.createDirectory*
- *Code Snippet 4: Checking if directory exists - Files.exists*
- *Code Snippet 5: Creating hierarchy of directories - dir.resolve*

- *Code Snippet 6: Creating temp files - Files.createTempFile*
- *Code Snippet 7: Tempfiles without prefix & suffix. Extension will be tmp*
- *Code Snippet 8: Temp file in default temp folder*
- *Code Snippet 9: Deleting a file. IOException if does not exist - Files.delete*
- *Code Snippet 10 : Copying files - Files.copy*
- *Code Snippet 11: Moving files - Files.move*

- *File change notification : One way to do this is to poll the file system looking for changes but this approach is inefficient — it does not scale to applications that may have hundreds of open files or directories to monitor*
- *Welcome to java.nio.file.WatchService : register a directory (or directories) with the watch service.*
- *Register types of events you are interested in: file creation, file deletion, or file modification*
- *The registered process has a thread (or a pool of threads) dedicated to watching for any events it has registered for*

- *Create a WatchService (watcher) for the file system*
- *Register a directory with the watcher for the events (create, delete, modify). A WatchKey is returned*
- *Use a loop or (may be in a different thread) to wait for incoming events*
- *On event, WatchKey is signaled and placed in the Watcher's queue*
- *Get the WatchKey from the queue and. The WatchKey provides the filename*
- *Retrieve each pending event(s), reset the key and resume waiting for the events*

Write code print the events (create, modify delete) happening in your home folder

- *The `java.nio.file.attribute` package provides access to file attributes or metadata, an area that is **highly file-system specific.***
- *The package groups together the related attributes and defines a view of the commonly used ones.*
- *An implementation is required to support a basic view that defines attributes that are common to most file systems, such as file type, size, and time stamps.*
- *An implementation may also support additional views.*

- *The basic attributes that are common to all file systems is provided by the* ***BasicFileAttributeView*** *which stores all mandatory and optional visible file attributes.*
- *The attributes are kind of meta-data of the file. You can get*
  - *the size of the file*
  - *whether it is a directory*
  - *is it regular file*
  - *whether it is a symbolic link ?*
  - *the creation time*
  - *modified time*
  - *last access time*
  - *and a lot more*

- *Operating systems and several third party applications have a file search function where a user defines search criteria*
- *Should you need to search for all .class files, find and delete .mov files or find all files that haven't been accessed in the last month, then FileVisitor is the solution*
- *With FileVisitor, you can traverse the file tree to any depth and perform any action on the files or directories found on any branch*

- ***FileVisitor** interface has 4 methods that you need to implement*
  - *FileVisitResult **preVisitDirectory** (Path dir, BasicFileAttributes attrs)*
  - *FileVisitResult **visitFile**(Path file, BasicFileAttributes attrs)*
  - *FileVisitResult **visitFileFailed**(Path file, IOException exc)*
  - *FileVisitResult **postVisitDirectory**(Path dir, IOException exc)*
- *The return value at each stage is of type **FileVisitResult** and controls the flow of the traversal*

FileVisitResult is an enum of four possible return values for the FileVisitor interface methods

- ***FileVisitResult.CONTINUE*** *– indicates that the file tree traversal should continue after the method returning it exits*
- ***FileVisitResult.TERMINATE*** *– stops the file tree traversal and no further directories or files are visited*
- ***FileVisitResult.SKIP_SUBTREE*** *– this result is only meaningful when returned from the preVisitDirectory API, elsewhere, it works like CONTINUE. It indicates that the current directory and all its subdirectories should be skipped*
- ***FileVisitResult.SKIP_SIBLINGS*** *– indicates that traversal should continue without visiting the siblings of the current file or directory. If called in the preVisitDirectory phase, then even the current directory is skipped and the postVisitDirectory is not invoked*

Checkout the class in your exercise code base

- *Basic I/O in Java is blocking -- meaning that it waits until it can complete an operation*
- *NIO's non-blocking I/O is event-based*
- *A selector (or callback or listener) is defined for an I/O channel, then processing continues*
- *When an event occurs on the selector -- when a line of input arrives, for instance -- the selector "wakes up" and executes.*
- *All of this is achieved within a single thread, which is a significant contrast to typical Java I/O*

- *Scholarly articles on multiplexing*
  - *Asynchronous Java NIO*
  - *High Performance with I/O with Java NIO*
  - *How to build a Scalable Multiplexed Server with NIO*
  - *Five ways to maximize Java NIO and NIO.2*
- *All in all it is about using fewer threads to serve high number of concurrent requests*

AsynchronousServerSocketChannel and AsynchronousSocketChannel classes are the key classes used in implementing the server and client respectively.

```
AsynchronousServerSocketChannel server  = AsynchronousServerSocketChannel.open();
server.bind(new InetSocketAddress("127.0.0.1", 7001));
server.bind(null);
Future<AsynchronousSocketChannel> future = server.accept();
AsynchronousSocketChannel worker = future.get();
```

AsynchronousFileChannel is the key classes used

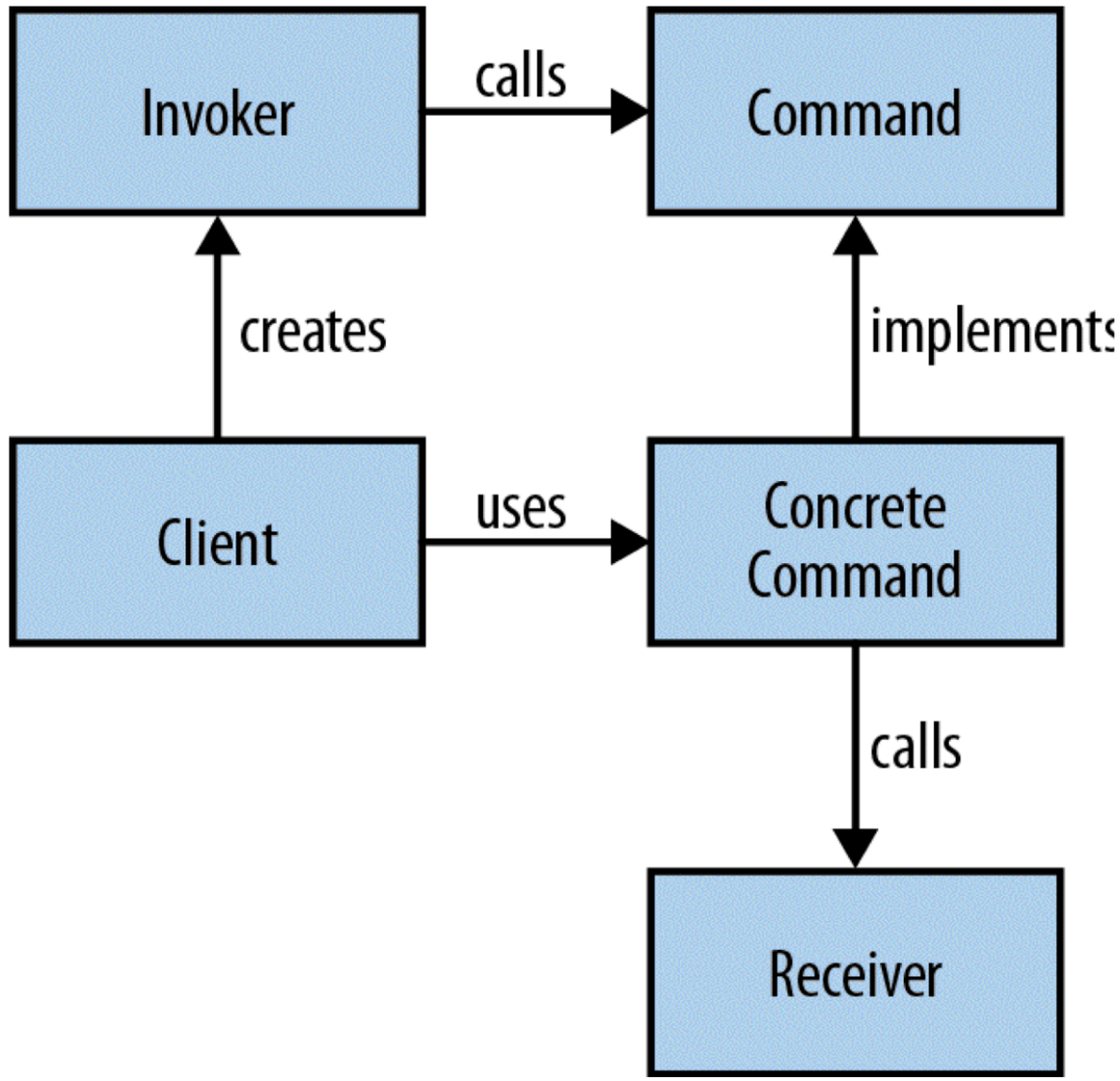
```
Path filePath = Paths.get("/path/to/file");
AsynchronousFileChannel fileChannel = AsynchronousFileChannel.open(filePath, READ,
WRITE, CREATE, DELETE_ON_CLOSE);
AsynchronousFileChannel fileChannel = AsynchronousFileChannel.open(filePath,
StandardOpenOption.READ)
```

- *Patterns document reusable templates that solve common problems in software architecture.*
- *If you spot a problem and you're familiar with an appropriate pattern, then you can take the pattern and apply it to your situation.*
- *In a sense, patterns codify what people consider to be a best-practice approach to a given problem.*

- *A command object is an object that encapsulates all the information required to call another method later*
- *The command pattern is a way of using this object in order to write generic code that sequences and executes methods based on runtime decisions.*

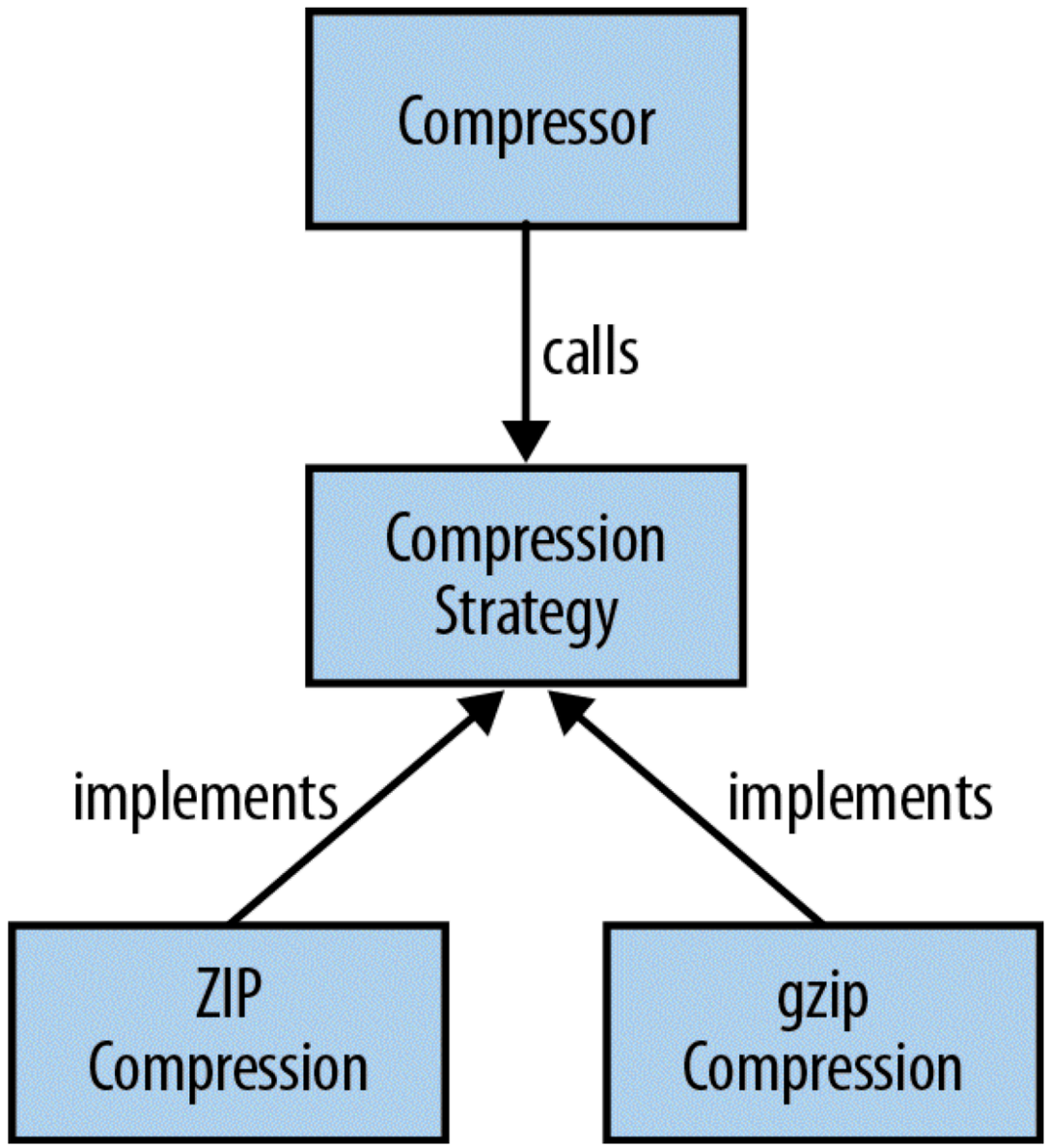
There are four classes that take part in the command pattern

- *Receiver Performs the actual work*
- *Command Encapsulates all the information required to call the receiver*
- *Invoker Controls the sequencing and execution of one or more commands*
- *Client Creates concrete command instances*



- *The strategy pattern is a way of changing the algorithmic behavior of software based upon a runtime decision.*
- *The idea is to be able to define a common problem that is solved by different algorithms and then*
- *Encapsulate all the algorithms behind the same programming interface.*

- *The strategy pattern is a way of changing the algorithmic behavior of software based upon a runtime decision.*
- *The idea is to be able to define a common problem that is solved by different algorithms and then*
- *Encapsulate all the algorithms behind the same programming interface.*



- *The observer pattern is a behavioral pattern that can be improved and simplified through the use of lambda expressions.*
- *In the observer pattern, an object, called the subject, maintains a list of other objects, which are its observers.*
- *When the state of the subject changes, its observers are notified.*

- *The subject has a function called addObserver which takes an argument of type Observer*
- *The client creates instances of concrete Observers (SimpleWeatherForeCastDisplay, ConsolidatedDisplay) which invokes addObserver*
- *Note that the Observer can be a functional interface*
- *Using lambdas, you can do away with Concrete Observer classes and pass lambda expression for e.g.*
    - *weatherInfo.addObserver(name -> { implementation of observer });*

- *A pretty common situation when developing software is having a common algorithm with a set of differing specifics.*
- *We want different implementations to have a common pattern in order to ensure that they're following the same algorithm*
- *Your overall algorithm design is represented by an abstract class which has a series of abstract methods that represent customized steps in the algorithm*
- *Any common code can be kept in this class.*
- *Each variant of the algorithm is implemented by a concrete class that overrides the abstract methods and provides the relevant implementation.*

- *A domain-specific language (DSL) is a programming language focused on a particular part of a software system.*
- *Less expressive than a general-purpose language.*
- *Highly specialized: by trading off being good at everything, they get to be good at something.*
- *Cascading Style Sheets (CSS) and regular expressions are commonly used external DSLs*
- *Makes your code more succinct and easier to read.*
- *Ideally, code written in a DSL reads like statements made within the problem domain that it is reflecting.*
- *Check Streams, CompletableFuture, IntegrationFlow in Spring Integration*

- *Java SE Embedded 8 introduces a new concept called, Compact Profiles, which enable reduced memory footprint for applications that do not require the entire Java platform.*
- *The Java SE 8 `javac` compiler has a new `-profile` option, which allows the application to be compiled using one of the new supported profiles*
- *There are three supported profiles: `compact1`, `compact2` and `compact3`*
- *Usage*
  - *The compact profiles feature is useful in small devices.*
  - *Smaller storage footprint, compact profiles*
  - *Can enable many Java applications to run on resource-constrained devices*

- *A smaller Java environment would require less compute resources, thus opening up a new domain of devices previously thought to be too humble for Java.*
- *A smaller runtime environment could be better optimized for performance and start up time.*
- *Elimination of unused code is always a good idea from a security perspective.*
- *If the environment could be pared down significantly, there may be tremendous benefit to bundling runtimes with each individual Java application.*
- *These bundled applications could be downloaded more quickly.*

Each profile includes the APIs of the lower-numbered profiles (compact2 is a superset of compact1)

Full SE API	Beans	Input Methods	IDL
	Preferences	Accessibility	Print Service
	RMI-IIOP	CORBA	Java 2D
	Sound	Swing	
	AWT	Drag and Drop	
	Image I/O	JAX-WS	
compact3	Security[1]	JMX	JNDI
	XML JAXP[2]	Management	Instrumentation
compact2	JDBC	RMI	XML JAXP
compact1	Core (`java.lang.*`)	Security	Serialization
	Networking	Ref Objects	Regular Expressions
	Date and Time	Input/Output	Collections
	Logging	Concurrency	Reflection
	JAR	ZIP	Versioning
	Internationalization	JNI	Override Mechanism
	Extension Mechanism	Scripting	

- *When developing applications for Compact Profiles, you must remember to only use the APIs found in the specifications for compact1, compact2, or compact3 profiles.*

- *To ease in the programming of embedded apps for Compact Profiles, you can use the "jdeps" tool to see which Java packages are being used inside your source code after you have written your app.*

- *The result of running jdeps on your Java app will allow you to see which Compact Profile runtime you will need to be able to execute your application*

```
% jdeps -P HelloWorld.class

HelloWorld.class -> /net/test11.us.example.com/export/java-
re/jdk/8/ea/b124/binaries/linux-i586/jre/lib/rt.jar
   <unnamed> (HelloWorld.class)
      -> java.io                                            compact1
-> java.lang                                               compact1
```

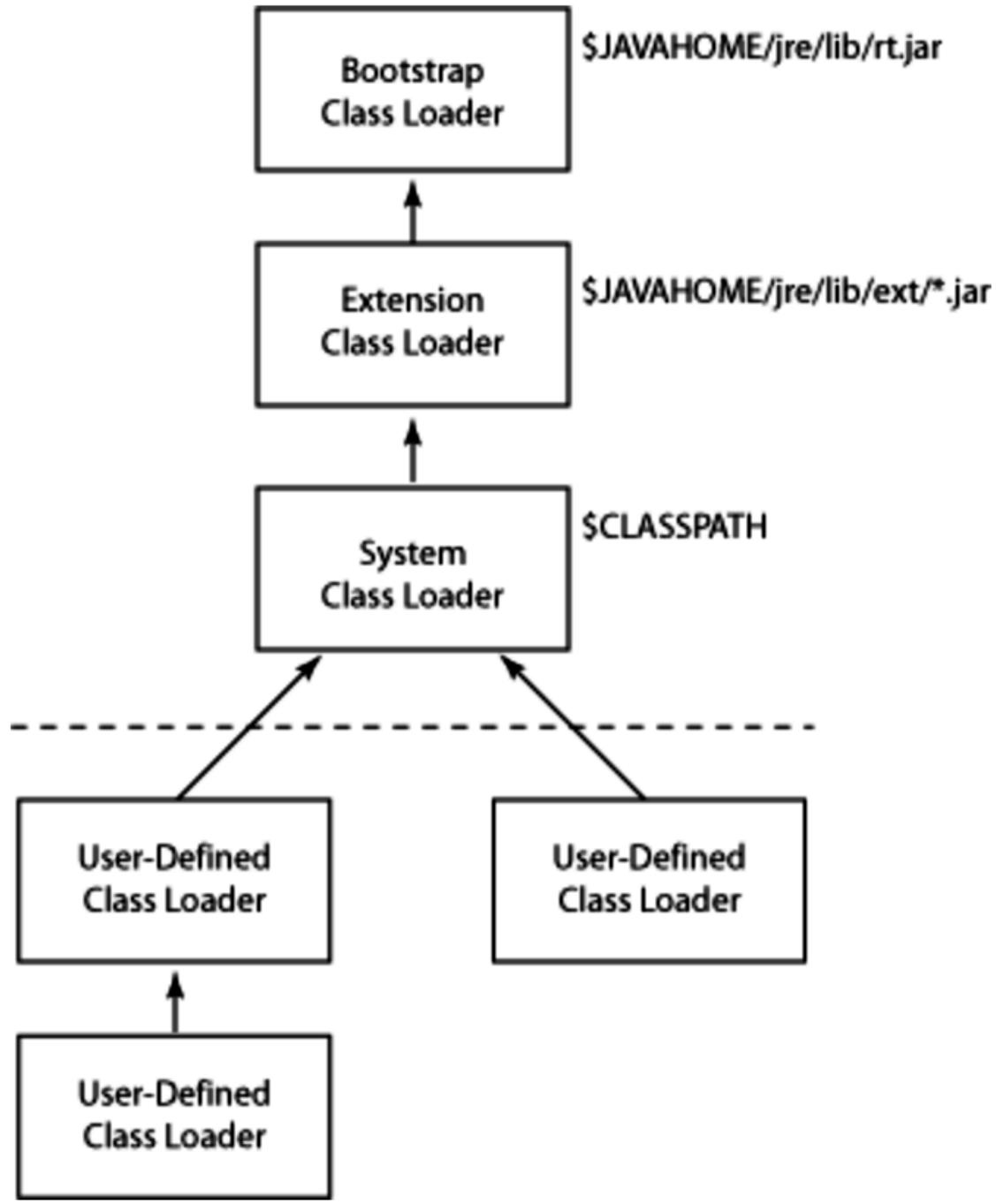
```
javac -profile compact1 Counter.java
```

- *A formal effort has been underway having the stated goal of providing a much more modular Java platform.*
- *Called Project Jigsaw, when complete, Java SE will be composed of a set of finer-grained modules and will include tools to enable developers to identify and isolate only those modules needed for their application.*
- *However, implementing this massive internal change and yet maintaining compatibility has proven to be a considerable challenge.*
- *Consequently full implementation of the modular Java platform has been delayed until* ***Java 9.***

- *The class loader delegation model is the graph of class loaders that pass loading requests to each other.*
- *The bootstrap class loader is at the root of this graph.*
- *Class loaders are created with a single delegation parent and looks for a class in the following places:*
  - *Cache*
  - *Parent*
  - *Self*

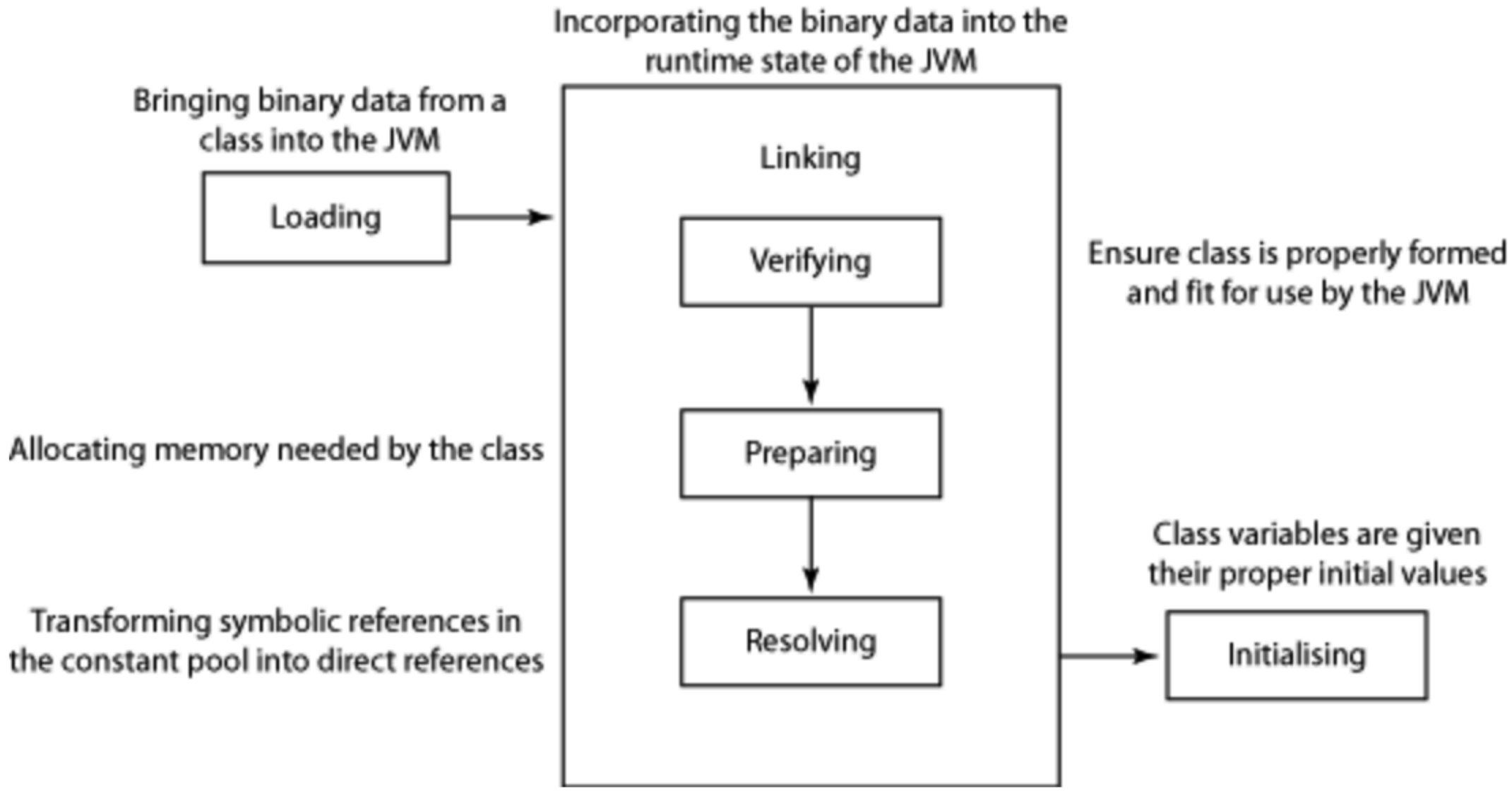
- *A class loader first determines if it has been asked to load this same class in the past.*
- *If so, it returns the same class it returned last time (that is, the class stored in the cache).*
- *If not, it gives its parent a chance to load the class.*
- *These two steps repeat recursively and depth first.*
- *If the parent returns null (or throws a ClassNotFoundException), then the class loader searches its own path for the source of the class.*

- *Because the parent class loader is always given the opportunity to load a class first, the class is loaded by the class loader nearest the root.*
- *This means that all core bootstrap classes are loaded by the bootstrap loader, which makes sure that the correct versions of classes such as java.lang.Object are loaded.*
- *A class loader to see classes loaded by itself or its parent or ancestors*
- *The classes loaded by its children are* ***not visible*** *to the parent class loader*



- *The **bootstrap** class loader (also known as the primordial class loader) cannot be instantiated by Java code.*
- *BootstrapClassLoader class loader loads the core system classes from the boot classpath, which is normally the JAR files located in the jre/lib directory.*
- *The **extension** class loader (also known as the standard extensions class loader) is a child of the bootstrap class loader. Its primary responsibility is to load classes from the extension directories, normally located the jre/lib/ext directory. This provides the ability to simply drop in new extensions, such as various security extensions, without requiring modification to the user's classpath.*
- *The **system class loader** (also known as the application class loader) is the class loader responsible for loading code from the path specified by the CLASSPATH environment variable. By default, this class loader is the parent of any class loader created by the user. This is also the class loader returned by the ClassLoader.getSystemClassLoader() method.*

Problems relating to class loading can be tracked down to a problem occurring in one of these phases. Hence, understanding these phases is important



- ***ClassNotFoundException*** : *is the most common type of class loading exception. It occurs during the loading phase.*
- *Thrown when an application tries to load in a class through its string name using:*
  - *The forName method in class Class*
  - *The findSystemClass method() in class ClassLoader.*
  - *The loadClass() method in class ClassLoader.*
- *So a ClassNotFoundException is thrown if an attempt to load a class fail*
- *Resolution : These exceptions are usually simple to fix. You can ensure that the classpath being used is set as expected*

***NoClassDefFoundError** : is thrown as a result of a unsuccessful implicit class load.*

```
1  public class NoClassDefFoundErrorTest {
2      public static void main(String[] args) {
3          A a = new A();
4      }
5  }
```

Listing 3. A.java

```
1  public class A extends B {
2  }
```

Listing 4. B.java

```
1  public class B {
2  }
```

At the time of compilation class B exists but then removed before execution

UnsatisfiedLinkError : *Occurs during the resolving stage of the linking phase when a program tries to load an absent or misplaced* ***native library***

```
public class UnsatisfiedLinkErrorTest {

    public native void call_A_Native_Method();

    static {
        System.loadLibrary("myNativeLibrary");
    }

    public static void main(String[] args) {
        new UnsatisfiedLinkErrorTest().call_A_Native_Method();
    }
}
```

Once you understand the class loaders involved in the loading of the library, you can resolve these types of problems by placing the library in an appropriate location.

ClassFormatError : *The binary data that purports to specify a requested compiled class or interface is malformed.*

This exception is thrown during the ***verification stage of the linking phase*** *of class loading.*

The binary data can be malformed if the bytecodes have been changed -- if the major or minor number has been changed.

This could occur if the bytecodes had been deliberately hacked, for example, or if an error had occurred when transferring the class file over a network

The only way to ***fix*** *this problem is to obtain a corrected copy of the bytecodes, possibly by recompiling.*

- *Classloader leaks can cause OOME - Perm Space*
- *IllegalAccessExeption : when you try to load a class that is not accessible out of the package it is defined in*
- *ClassCastException : every one knows this :)*

Custom ClassLoader & LinkageError

- *loadClass method implements the delegation model*
- *The method delegates to its parent class loader before attempting to find the class itself. This means that it now finds java.lang.Object loaded by the bootstrap class loader.*

```
public Class loadClass(String name) throws ClassNotFoundException {
    Class c = null;
    try {
        c = getParent().loadClass(name);
    } catch (ClassNotFoundException e) {
    }
    if(c == null)
        c = findClass(name);
    return c;
}
```

The right way of doing it

- *The above may still lead to* ***LinkageError*** *because because the application is asking the class loader to load the same class twice, and loadClass() tries to reload it from scratch.*
- *First check in cache*

```
public Class loadClass(String name) throws ClassNotFoundException {
    Class c = findLoadedClass(name);
    if(c == null) {
        try {
            c = getParent().loadClass(name);
        } catch (ClassNotFoundException e) {
        }
        if(c == null)
            c = findClass(name);
    }
    return c;
}
```

- *About hot loading : Updating the byte code in the perm gen directly using java.lang.instrumentation*
- *JRebel from ZeroTurnAround does the same*
- *Check demo project : **instrumentation***

- *Loading of class over the network on nodes in the cluster.*
- *Very powerful mechanism as you do not have to redeploy on all the nodes*
- *This is how it fundamentally works*
  - *Your product / framework with a custom class loader will check if class is available on local classpath **and if it was**, it will be returned. No class loading from a peer node will take place in this case.*
  - *If class is not locally available, then a request will be sent to the originating node to provide class definition.*
  - *Originating node will send class byte-code definition and the class will be loaded on the worker node.*
  - *This happens only once per class - once class definition is loaded on a node, it will never have to be loaded again.*

- *The **in-memory** computing platform*
- *The closures and tasks that you use for your computations may be of any custom class, including anonymous classes.*
- *In Ignite, the remote nodes will automatically become aware of those classes, and you won't need to explicitly deploy or move any .jar files to any remote nodes.*
- *Such behavior is possible due to peer class loading (P2P class loading), a special **distributed ClassLoader** in Ignite for inter-node byte-code exchange.*
- *With peer-class-loading enabled, you don't have to manually deploy your Java or Scala code on each node in the grid and re-deploy it each time it changes*

- *Under the covers, Java 7 version of the platform shipped the invokedynamic instruction*
- *It laid the foundation for implementing lambda expressions in Java 8,*
- *It also was a game changer for translating dynamic languages into the Java byte code format*

- *A method handle is "a typed, directly executable reference to an underlying method, constructor, field, or similar low-level operation, with optional transformations of arguments or return values."*
- *java.lang.invoke.MethodHandles.Lookup object is a factory for creating method handles and is used to search for targets such as virtual methods, static methods, special methods, constructors, and field accessors.*

- *It is dependent on a call site's invocation context and enforces method handle access restrictions each time a method handle is created.*
- *In other words, a call site (for e.g. if lookup is called from main() then main method is acting as a call site)*
- *The lookup object can only access those targets that are accessible to the call site.*
- *The lookup object is obtained by invoking MethodHandles.lookup*

- *Other than* `Method` *instances of the reflection API, method handles can equally reference fields or constructors.*
- *Effectively, it does not matter what class member is referenced via a method handle at runtime as long as its* `MethodType`

- *Method handles are often described as being a more performant as the Java reflection API*
- *Reflection is too much of overhead as it takes objects and has to determine the right types runtime.*

- *Call site is the method from where the Method Handle is created.*
- *At runtime, an invokedynamic call site is bound to a method handle by way of a bootstrap method.*
- *This bootstrap method is executed the first time the JVM encounters this call site during execution.*
- *The invokedynamic instruction is followed by an operand that serves as an index into the classfile's constant pool. The value in the constant pool at this index is a CONSTANT_InvokeDynamic structure. This structure includes an index into the classfile's BootstrapMethods attribute, which ultimately identifies the bootstrap method.*

- *Whenever the Java compiler translates a lambda expression into byte code, it copies the lambda's body into a private method inside of the class in which the expression is defined.*
- *These methods are named* `lambda$X$Y` *with* `X` *being the name of the method that contains the lambda expression and with* `Y` *being a zero-based sequence number.*
- *The parameters of such a method are those of the functional interface that the lambda expression implements.*
- *The lambda expression is itself substituted by an invokedynamic call site.*
- *On its invocation, this call site requests the binding of a factory for an instance of the functional interface.*

- *As arguments to this factory, the call site supplies any values of the lambda expression's enclosing method which are used inside of the expression and a reference to the enclosing instance, if required.*

- *As a return type, the factory is required to provide an instance of the functional interface.*

- *For bootstrapping a call site, invokedynamic instruction currently delegates to the* `LambdaMetafactory` *class which is included in the Java class library.*

- *This factory is then responsible for creating a class that implements the functional interface and which invokes the appropriate method that contains the lambda's body which, as described before, is stored in the original class.*

- *In the future, this bootstrapping process might however change which is one of the major advantages of using invokedynamic for implementing lambda expressions.*

- *If one day, a better suited language feature was available for implementing lambda expressions, the current implementation could simply be swapped out.*

Why this matters for dynamic languages

- *Any language that should be executed on the Java virtual machine needs to be translated to Java byte code.*
- *And as the name suggests, Java byte code aligns rather close to the Java programming language.*
- *This includes the requirement to define a strict type for any value and before invokedynamic was introduced, a method call required to specify an explicit target class for dispatching a method.*
- *Looking at the following JavaScript code, specifying either information is however not possible when translating the method into byte code*

```
function (foo)
    foo.bar();
}
```

- *Using an invokedynamic call site, it has become possible to delay the identification of the method's dispatcher until runtime and furthermore, to rebind the invocation target, in case that a previous decision needs to be corrected.*
- *Before, using the reflection API with all of its performance drawbacks was the only real alternative to implementing a dynamic language*