Aprameyah
Technologies Pvt. Ltd.

Java 8 Primary Enhancements

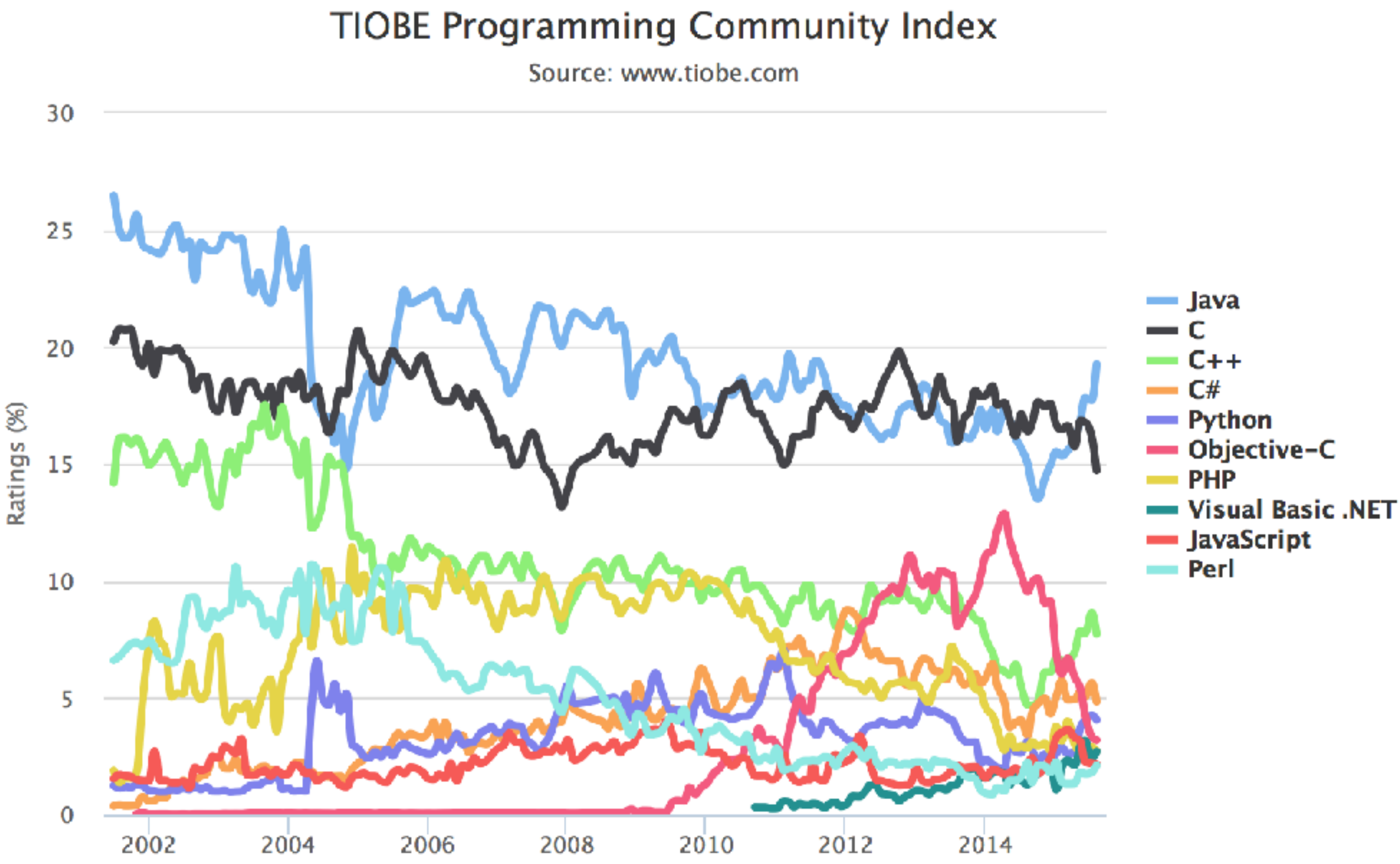
Aprameyah
Technologies Pvt. Ltd.

About Me

- Maruthi Janardhan
- Consulting Chief Architect Trilogy Group
- Electronics Engineer now a semi active hobbyist and a farm entrepreneur
- Started career with corporates like ANZ, HP, IBM and then spent time at 2 startups plus one of my own - total 18 years in the industry
- I train occasionally
- Expertise has been enterprise systems, aws, python/java

Aprameyah
Technologies Pvt. Ltd.

Over the Years



ForEach

- Study java.awt.button.addActionListeners

- Extend the ArrayList and in the child class create a “forEach” function on similar lines to register an iteration handler

- Create another method called “iterate” that calls this handler for each element in the list

Some More…

- Create a class whose objects know their position in an array they are inserted into
- Send an email to 100 customers present in a list
- Do not send emails to customers with Yahoo mail id
- Now eliminate any embedded links in the email content
- Count successfully sent emails

Aprameyah
Technologies Pvt. Ltd.

Passing Function References

- Javascript, Python, Ruby, Scala, etc

```
var strs = ["one", "two", "three"];
        strs.sort(function(s1,s2){
            return (s1.length-s2.length);
        });
```

```
-----------------------------
```

```
def myfunction(data):
    print(data)

printer = myfunction
printer("Hello there")
```

Consider This

```
for(Integer x: intcollection){
     //do something with x
}
----------------------
interface Processor<T>{
   public void process(T x);
}
intcollection.forEach(new Processor(){
   public void process(Integer x){
      //do something with x
   }
});
----------------------
intcollection.forEach((Integer x)->{
   //do something with x
});
```

Aprameyah
Technologies Pvt. Ltd.

Lambda

- Concise syntax

```
button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            //button action code
        }
});
```

```
button.addActionListener((ActionEvent e)->{
        //button action code
});
```

- Inner classes handle “this” differently. No access to non final local vars.

- Programmers in many languages are used to callbacks, closures, map/reduce programming.

Biggest Gain: new way of thinking

- Encourages Functional thinking
  - Solves many classes of problems in a clean way
  - Encourages Liskov's open-close principle
- Supports Streams
  - Map/Filter/Reduce, Lazy Evaluation, Easy Parallel processing

```
List<User> users;
for(User u: users){
      u.retire();
}

users.parallel().forEach(u->{u.retire();});
```

Filter Interface

- Create a SAM interface which has a method that accepts a generic type and returns boolean
- Create a method in the extended arraylist to register this filter interface
- Change the implementation of forEach to use this filter
    - Now implement Yahoo mail filter using this

Mapping Interface

- Create a SAM interface with a method that transforms an object of one type to another
- In the extended arraylist, create a method that registers a map function called “transform”
- Change the implementation of forEach to use this map
    - Now implement the link removal using a map interface

Reduce Interface

- Create a reduction interface that has an SAM that takes initial value, prev value and new value
- Create a reduce method on our extended arraylist class that takes an initial value and reduction interface - returns the reduced value
    - Use this to compute the count

Type Inferencing and Parentheses

- We dont have to specify the type of arguments in a lambda expression.

```
(Float m)->{
   //method impl
}

(m)->{
   //method impl
}

m->{
   //method impl
}
```

Expression For Body

- If the interface method we are making a lambda has a ret val, the lambda need not return it directly
    - An expression in lambda is treated as a return value

```
interface SomeInterface{
    public Integer methodOne(Float m);
}

class SomeClass {

    public void someMethod(SomeInterface i){
        //Do something
    }

    public static void main(String[] args) {
        SomeClass x = new SomeClass();
        x.someMethod((m)-> (int)(m*m));
    }
}
```

Single statements are always not expressions

```
x.someMethod((m)-> if(m>5) return 3; else return 5;);
```

Thinking

```
Arrays.sort(strings, (a,b)->a.length-b.length);
```

- Whats Really Happening
  - A shortcut way of representing an instance of a class that implements Comparator<T>. Body of compare method is provided after ->
- How we usually think of it
  - We passed a comparison function or expression

Aprameyah
Technologies Pvt. Ltd.

java.util.function

- Many reusable interfaces
    - Set of ordinary interfaces treated as a set of functions
- Simple Typed Interfaces
    - IntPredicate, DoubleBinaryOperator
- Predicate<T> - T in, Boolean out **(FILTER)**
- Function<T,R> - T in, R out **(MAP)**
- Consumer<T> - T in, nothing out
- Supplier<T> - nothing in, T out
- BinaryOperator<T> - Two Ts in, T out

Standard Functions

- java.util.function package has standard interfaces for our situation

- Replace the Transformer and Eliminator interfaces with Function and Predicate interfaces from java.util.function

Aprameyah
Technologies Pvt. Ltd.

Streams

- Streams are a way to sequentially access a list with lot of useful functions
- Streams are faster and memory efficient ways of processing lists
    - Lazy Evaluation
    - Automatic Parallelization
    - infinite unbounded streams
- Streams do not store data. Just programmatic wrappers on data structures

Aprameyah
Technologies Pvt. Ltd.

Streams

- Support many high performance operations expressed with lambdas expressed sequentially or in parallel

- Streams do not modify the underlying data structure

```
Stream s = Stream.of(empids)
                          .map(dao::findEmpById)
                          .filter(e->e.getSalary()>10000);

Employee richEmplpyee = s.findFirst()
                          .orElse(null);
```

Streams

- Designed for lambdas
- Do not support indexed access
- However can generate arrays
- Lazy evaluation - only needed amount of data is analyzed
- Parallel execution without creating threads
- Tied with a generator function, streams can be infinite

Important Thing about Streams

- When you make or transform a Stream, it does not copy the underlying data. Instead it just builds a pipeline of operations. How many times this pipeline is invoked will depend on what we do with the stream in the end. Such as find first element or sort etc.

- Ways to create stream
  - list.stream()
  - Stream.of(array)
  - Stream.of(val1, val2,..)

Re-do

- Send an email to 100 customers present in a list
    - Do not send emails to customers with Yahoo mail id
    - Now eliminate any embedded links in the email content
    - Count successfully sent emails

Default Interfaces

- Interfaces with only one method or a SAM (Single Abstract method) interface

- Default interfaces (Use only for keeping interfaces backward compatible - violates the spirit of interfaces):

```
interface SomeInterface{
    public void methodOne();
    public default void methodTwo(){
        //Default impl of method 2
    }
}

class SomeImplementation implements SomeInterface{
    @Override
    public void methodOne() {
        // Method 1 impl

    }
}
```

Scope

- Can access outer class variables as long as they are effectively final. Try this:

```
public static void main(String[] args) {
    SomeClass x = new SomeClass();
    String[] strs = {"ab","abc","a"};
    Arrays.sort(strs, (a,b)->{
        System.out.println(x.toString());
        return a.length()-b.length();
    });
}
```

- Now change x

@Override annotation

- What are the uses of the @Override annotation. Write some code to understand.

- Catches errors at compile time

- Expresses design intent clearly

- But not really required

Aprameyah
Technologies Pvt. Ltd.

@FunctionalInterface

- This annotation can decorate an interface meant to be treated as a functional interface

- Compiler gives you an error on this:

```
@FunctionalInterface
interface SomeInterface {
   public Integer methodOne(Float m);
   public void secondMethod();

}
```

Method References

```
@FunctionalInterface
interface SomeInterface {
    public void methodOne(Float m);

}

class SomeClass {

    public void someMethod(SomeInterface i) {
        i.methodOne(4.5f);
    }

    public static void main(String[] args) {
        SomeClass s = new SomeClass();
        s.someMethod(f -> {
            System.out.println(f);
        });

        s.someMethod(System.out::println);

    }
}
```

Method References

- Signature should match the functional interface
- ClassName::staticMethod
    - Arrays::sort, String::valueOf
- variable::instanceMethod
    - System.out::println

Type of Method Reference

- type of System.out::println is undefined without a context.
  - All these are syntactical sugars. Java does not have method reference. here it is of type of the param

```
s.someMethod(System.out::println);
```

- We can even use constructors as method references: Customer::new

Consuming Streams

- stream.collect(Collectors.toList())
- stream.toArray(EntryType[]::new)
- stream.toArray(n->buildEmptyArray(n))
    - The lambda is an IntFunction that takes int(size) as an argument and returns an empty array that can be filled in.
- stream.findFirst()

Important Stream Methods

- forEach(Consumer)
  - designed for lambdas
  - reusable - save the lambda and use again
  - can be made parallel with minimum effort
  - Cannot loop twice on the same stream
  - Cannot change variables in enclosing closure
- map(Function)
- filter(Predicate)
- findFirst()

Try with resources

```
try (InputStream alinePropStream =
                    getResourceAsStream(“aline.properties");
        InputStream configPropStream =
                    getResourceAsStream("configuration.properties")) {
    //Use both prop streams

} catch (IOException e) {
        throw new RuntimeException(e);
}
```

Paths

- Path is a flexible replacement for file
- Path p = Paths.get(“/somefile”);
- Files.lines(p) - yields a stream

Collectors

- Collectors are terminal operations that produce mutable reduction
    - They apply on copies of data
- stream.collect()
- Collectors is a class full of collector algorithms
    - Collectors.toList
    - Collectors.partitionBy
    - Collectors.groupBy

Lets try it

- Read a csv file and assemble customers using a mapping function
    - Use try-resource construct
- Using collectors, get hold of a list of customers
- Using collectors.groupBy classify customers by their mail domain
- Using collectors.partitionBy to divide customers who have a mail id and who dont

Use the map merge function

- Merge a new customer into the grouped by data using the map merge and bifunction

- Merge function: If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.

Merge

```
CustomerWithEmail cnew = new CustomerWithEmail("x@abc.com", "vinay", "Hello
there");
List<CustomerWithEmail> custs = new ArrayList<>();
custs.add(cnew);
```

```
groupedData.merge(cnew.email.split("@")[1], custs,
        new BiFunction<List<CustomerWithEmail>, List<CustomerWithEmail>,
                        List<CustomerWithEmail>>() {
                public java.util.List<CustomerWithEmail>
                    apply(List<CustomerWithEmail> t, List<CustomerWithEmail> u) {
                        t.addAll(u);
                        return t;
                };
});
```

Parallel Reduction

- Take a stream of customers, parallel reduce it using an accumulater and combiner function. Print thread ids to see threading

- Reduce first param - identity

- Second param - bifunction to accumulate identity and stream type to return identity type

- Third param - binary operator to combine return values of accumulators

```
Integer count = s.parallel().reduce(0,
    (num,cust)->{
                    S.O.P(Thread.currentThread().getId()+": Reducing: "+num+"
"+cust.email);
                    return num+1;
    },
    (num1,num2)->{
                    S.O.P(Thread.currentThread().getId()+": Combining: "+num1+"
"+num2);
                    return num1+num2;
    });
```

Aprameyah
Technologies Pvt. Ltd.

Get a list of email ids via parallel reduction

```
List<String> emailIds = s.parallel().reduce(new ArrayList<String>(),
                ( t, u) -> {
                        ArrayList<String> ret = new ArrayList<>();
                        ret.addAll(t);
                        ret.add(u.email);
                        return ret;
                },
                ( t, u) ->{
                        ArrayList<String> ret = new ArrayList<>();
                        ret.addAll(t);
                        ret.addAll(u);
                        return ret;
                });
```

Aprameyah
Technologies Pvt. Ltd.

Flatmap

- Take a list of customers with multiple email ids, Say emails are separated with a -
- Produce an expanded list where each customer has only one email id

```
custStream.flatMap(cust ->{
    String[] emails = cust.email.split("-");
    List<CustomerWithEmail> customers = new ArrayList<>();
    for(String email: emails){
        customers.add(new CustomerWithEmail(email, cust.name,
                                            cust.emailContent));
    }
    return customers.stream();
})
.distinct().forEach(System.out::println);
```

Aprameyah
Technologies Pvt. Ltd.

File walking

- Walking a tree is a library feature in java 8

```
Files.walk(Paths.get(“/Users/maruthir“))
```

- Build a stream based logic that deletes all .jpg files in a directory tree

```
Files.walk(Paths.get(".")).filter(path ->{
    return path.toString().contains(".jpg");
}).forEach(path->{
    try {
        Files.delete(path);
    } catch (Exception e) {}
});
```

Aprameyah
Technologies Pvt. Ltd.

Static Methods In Interfaces

- Java 8 supports static methods on interfaces
- Generic types can now have methods related to the type.. not just class.
- Default methods bring back the diamond problem of conflicting implementations.

functions that return predicates - Static methods on predicate

- AND - given a predicate as an argument, returns a new predicate whose test method is true if both original predicate and argument predicate is true for an argument
- OR - guess
- NEGATE - Takes no arguments, returns a predicate which returns the opposite of the original predicate

Lets Try It

- Using collectors and AND operation of predicates, get hold of customers who have a mail ID and have an email address

```
        Predicate<CustomerWithEmail> nameAndEmailFilter = new
Predicate<CustomerWithEmail>() {
            public boolean test(CustomerWithEmail t) {
                return t.email!=null && !t.email.equals("");
            }
}.and(cust->{
    return cust.name!=null && !cust.name.equals("");
});

linesFromFile.map(line->{
    CustomerWithEmail c = new CustomerWithEmail(line.split(",")[0],
                                    line.split(",")[1],
                                    line.split(",")[2]);
    return c;
}).filter(nameAndEmailFilter).forEach(System.out::println);
```

Try this

- Perform a filter and reduction operation on customers to get any customer who is under 18 years of age. use findAny operator of stream

- What happens if there is none

Optional

- Methods like findFirst return an Optional
- Optional<SomeClass> val = Optional.of(x)
- Optional<SomeClass> val = Optional.empty()
  - value.get(), value.orElse(otherVal), value.orElseGet(supplier), value.isPresent()

Other stream operations

- limit(n)
- skip(n)
- sorted()
- min()
- max()
- distinct()

- Filter out duplicate entries in the customer list with distinct stream operation

- What happens when the stream is a parallel stream

Matches

- allMatch
- anyMatch
- noneMatch
- count
- Number Specialised Streams

- Validate the CSV file data to make sure that there are no customers who do not have an email address

- use one of the stream match functions

Stream Issues

- Try this

```
IntStream.iterate(0, i -> ( i + 1 ) % 2)
    .distinct()
    .limit(10)
    .forEach(System.out::println);
```

- And this

```
IntStream.iterate(0, i -> ( i + 1 ) % 2)
    .parallel()
    .distinct()
    .limit(10)
    .forEach(System.out::println);
```

- And this

```
IntStream.range(1, 5)
    .peek(System.out::println)
    .peek(i -> {
         if (i == 5)
             throw new RuntimeException("bang");
     });
```

Aprameyah
Technologies Pvt. Ltd.

Samples

```
public ProductivityReport computeProductivity(long projectVersionId,
LocalDateTime startDate) {
    List<SvnCheckin> projectCheckins = _svnCheckinRepository
            .findByProductVersionIdAndCommitTimeGreaterThanAndJobResultIn(proje
ctVersionId, startDate,
                    ServiceStatus.COMPLETED);

    Map<String, List<SvnCheckin>> checkinsGroupedByDev =
projectCheckins.stream()
            .collect(Collectors.groupingBy(SvnCheckin::getAuthor));

    List<DeveloperProductivityReport> devReports =
checkinsGroupedByDev.entrySet().stream()
            .map(this::createDeveloperProductivityReport).collect(Collectors.to
List());

    return new ProductivityReport(devReports);

}
```

Aprameyah
Technologies Pvt. Ltd.

Samples

```
private DeveloperProductivityReport createDeveloperProductivityReport(
        Map.Entry<String, List<SvnCheckin>> developerCheckins) {
    long linesAccepted = developerCheckins.getValue().stream()
            .filter(checkin -> checkin.getJobResult() == JobResult.ACCEPTED)
            .mapToLong(e -> e.getLinesAdded() + e.getLinesRemoved() +
e.getLinesModified()).sum();

    return new
DeveloperProductivityReport.DeveloperReportBuilder().withAuthor(developerChecki
ns.getKey())
            .withCheckins(developerCheckins.getValue()).withLinesAcceptedByFire
wall(linesAccepted)
            .createAuthorReport();
}
```

Samples

```
List<String> usernames =
authorAggregateSummaryList.stream().map(AuthorAggregateSummary::getAuthor)
        .collect(Collectors.toList());
```

Samples

```
Map<String, AuthorDetails> authorDetailsMap = authorDetailsList.stream()
        .collect(Collectors.toMap(AuthorDetails::getUserName, details ->
details));
```

```
List<DeveloperSummary> devSums =
authorAggregateSummaryList.stream().map(summary -> {
    AuthorDetails details = authorDetailsMap.get(summary.getAuthor());
    List<SvnCheckin> svnCheckinsList =
svnCheckensMap.getOrDefault(summary.getAuthor(), new ArrayList<>());
    DeveloperSummary developerDetails = new DeveloperSummary(summary, details,
svnCheckinsList,
            getImageURIByUser(summary.getAuthor()), authorRankingsCalculator,
            _activeDirectory.getDisplayNameByUsername(summary.getAuthor()));
    return developerDetails;
}).collect(Collectors.toList());
```

Java8 Memory Model

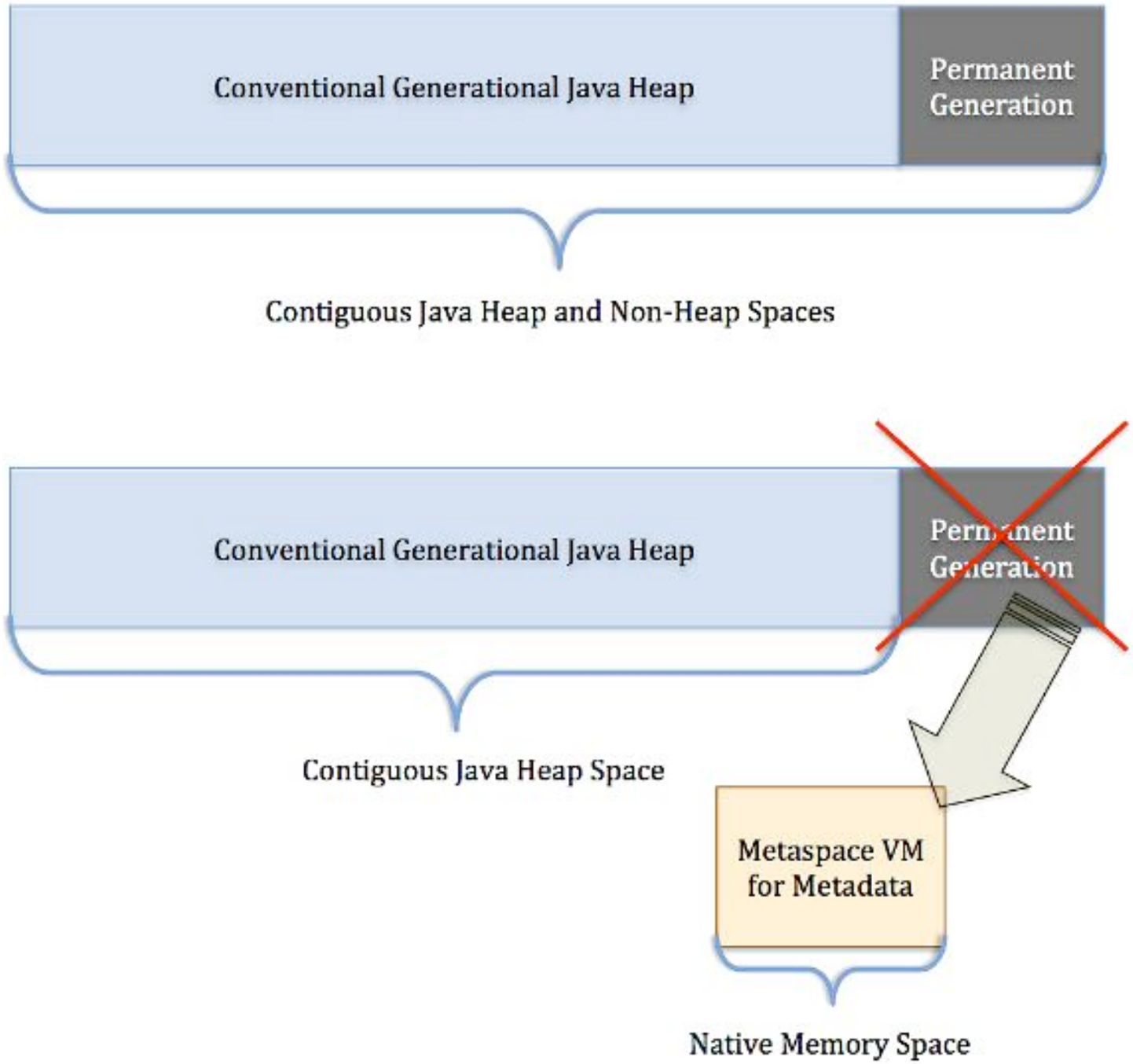
- JVM uses an internal representation of its classes containing per-class metadata such as class hierarchy information, method data and information (such as bytecodes, stack and variable sizes)
- This is stored in whats known as PermGen space
- Originally PermGen was designed to be permanent non-garbage collected store
- Defaults to 64 MB and tuned with XX:MaxPermSize

Java8 Memory Model

- The metadata has now moved to native memory to an area known as the “Metaspace”.
- PermGen was really hard to tune
- possibility that the metadata could move with every full garbage collection
- garbage collector in HotSpot needed specialized code for dealing with metadata

Aprameyah
Technologies Pvt. Ltd.

Java 8 Memory Model



Aprameyah
Technologies Pvt. Ltd.

What does this mean

- the max available space is the total available system memory
- Lambdas and such other dynamic class loading mechanisms are more efficient

Metaspace VM

- Metaspace VM now employs memory management techniques to manage Metaspace
- Moves the work from the different garbage collectors to just the one Metaspace VM
  - as long as the classloader is alive, the metadata remains alive in the Metaspace and can’t be freed
- There is no relocation or compaction in these metaspaces

Tuning

- -XX:MaxMetaspaceSize to set an upper limit.
  Default is no limit

- Its always good to set it to a valid value because it initially sets it to a pushable limit of 21MB and then starts garbage collecting if that limit is reached