# Java & Training

Vilas Varghese
Corporate Trainer
vilas_varghese@yahoo.com

# Rules.. Can we?

- Decide to be punctual?
- Decide to communicative?

# Agenda

- Interface static and default methods
- Functional paradigm
- Lambda expressions
- Stream API
- Collectors
- Optional
- Date and Time API

# Prerequisite

- Oracle Java 8 installed on your machine
  - http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html
  - Exact Version -> Java SE 8u60
- IntelliJ Idea Community Edition (or Eclipse Mars)
  - https://www.jetbrains.com/idea/download/
  - http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/marsr
- Git

# Why Languages evolve?

# Why Languages evolve?

- To meet developers expectations
- To remain relevant
- To keep up-to-date with hardware advancements
- Security fixes
- Better approaches to perform certain task

# Why you should learn Java 8?

- To embrace functional programming paradigm
  - Lambdas
  - Declarative data processing
- New and improved API's
  - Date and Time API
  - Stream API
  - Concurrency utilities
- Improved support for clean API design
  - Optional
  - Interface default and static methods

# Section 01

# Interface API Design

# Three main components of API Design

- Interfaces
- Abstract Classes
- Classes

# Let's write a simple Calculator

- Calculator >> Interface
- BasicCalculator >> Implementation

# Static methods

- Allow API designer to define static methods inside an interface
- for defining static utility methods

# Static method example

```
public interface Calculator {

    static Calculator getInstance() {
        return new BasicCalculator();
    }
```

Interfaces can now have static and default(instance) methods

# Why we need this?

- Ability to evolve API with time
- Remain source compatible

# Demo >>

# Evolving API's

# Default Methods

- Interfaces can supply non-abstract methods
- Enables interfaces to evolve with time
- Eliminate need for companion or utility classes

# Example of default method

Iterable's forEach method

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

# Multiple inheritance

- With interfaces having the capability to define methods, we have behaviour level multiple interfaces

# Interface Class Hierarchy

```java
public class App1 implements A {
    @Override
    public void doSth() {
        System.out.println("inside App1");
    }

    public static void main(String[] args) {
        new App1().doSth();
    }
}

interface A {
    default void doSth() {
        System.out.println("inside A");
    }
}
```

# Multiple Interface Hierarchy

```java
public class App2 implements B, D {
    public static void main(String[] args) {
        new App2().doSth();
    }
}

interface B {
    default void doSth() { System.out.println("inside B"); }
}

interface D extends B {
    default void doSth() { System.out.println("inside D"); }
}
```

# Diamond problem

```java
public class App3 implements E, F {
    @Override
    public void doSth() {
        F.super.doSth();
    }
    public static void main(String[] args) {
        new App3().doSth();
    }
}

interface E {
    default void doSth() {
        System.out.println("inside E");
    }
}

interface F {
    default void doSth() {
        System.out.println("inside F");
    }
}
```

# Section 02

# Functional Paradigm

# Programming paradigm

- A programming paradigm is a fundamental style of computer programming.
  - Imperative, declarative, functional, object-oriented, procedural, logical, etc.
- A programming language can follow one or more paradigms.

# Imperative vs Functional Paradigm

- Imperative programming paradigm

  - Computation: Uses statements that change a program's state.

  - Program: Consists of sequence of commands that tell a program how it should achieve the result.

- Functional programming paradigm

  - Computation: Evaluation of expressions

  - Expression: Formed by using functions to combine basic values.

  - Focusses on what the program should accomplish rather than how

Examples on next page

# Canonical example - Factorial

Imperative style >>

```java
public static int factorial(int number) {
    int factorial = 1;
    if (number == 0) {
        return factorial;
    }
    for (int i = 1; i <= number; i++) {
        factorial *= i;
    }
    return factorial;
}
```

Functional style >>

```java
public static int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    return number * factorial(number-1);
}
```

Scala >>

```
def fact(n: Int): Int = n match {
case 0 => 1
case _ => n * fact(n-1)
}
```

# Functional Programming

- Functional programming was invented in 1957
  - Before Object Oriented programming
  - Before structured programming
- Memory was too expensive to make functional programming practical

# Functional programming

"Functional programming is so called because a program consists entirely of functions."

– John Hughes, Why Functional Programming Matters

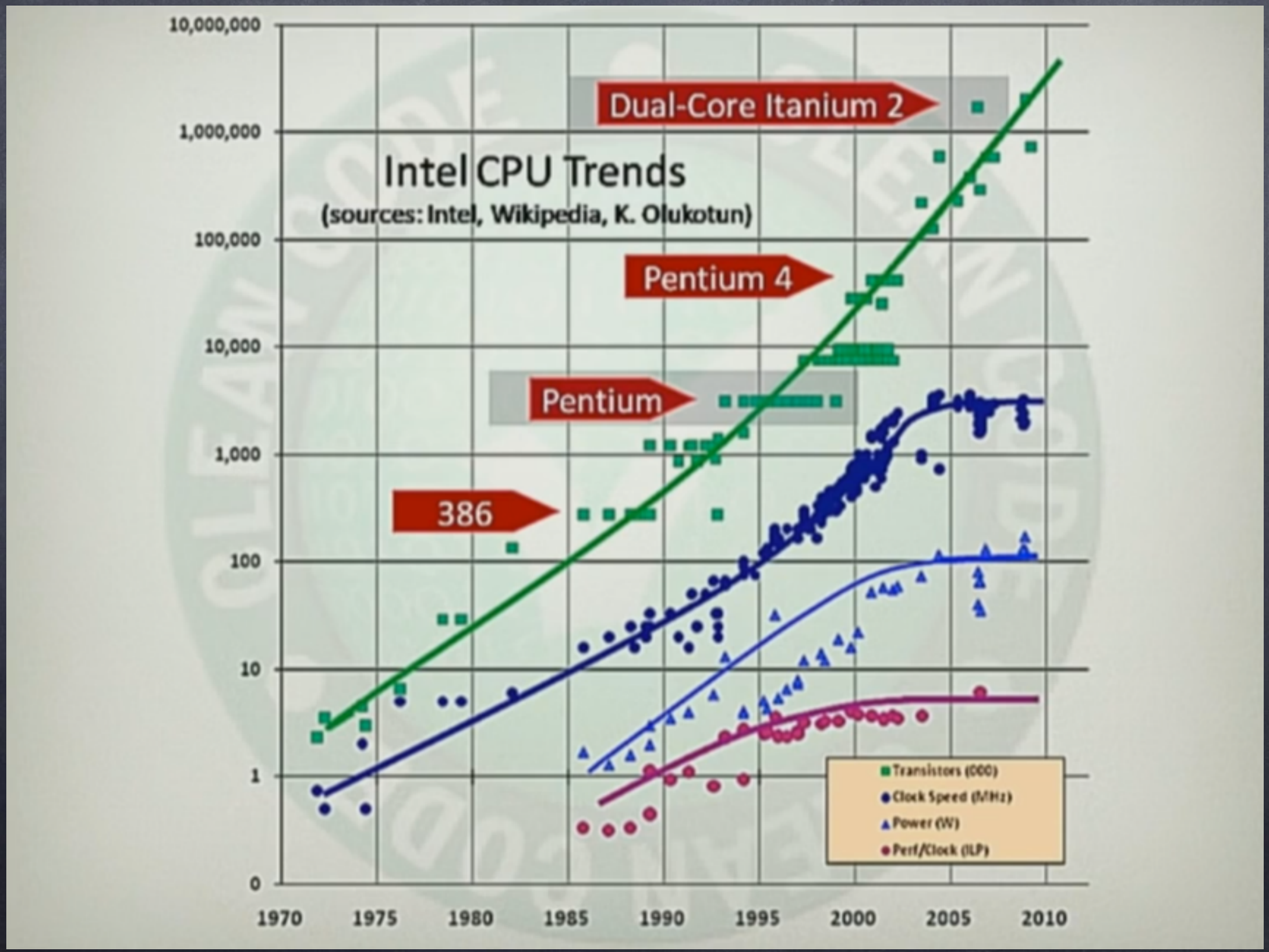
# What is a function?

- A side effect free computation that always result in same value when passed with same argument.

# Why should we embrace functional programming?

- Functional programs are simpler
- No side effects
- Fewer concurrency issues

# Moore's Law ran out!!



# Functional Languages

- Haskell
- ML
- Erlang
- F# - (Hybrid)
- Scala - (Hybrid)
- Clojure
- Java 8 - (Hybrid)

# Key Functional Programming Concepts

- Function
  - A side effect free computation that always result in same value when passed with same argument.
- Higher order function
  - function that takes function as argument or return functions.
- Referential transparency
  - allows you to replace function with its value
- Recursion
  - function calls itself
- Lazy evaluation
  - function is not evaluated until required

# Introduce Example Domain

- Task management application
  - Task
  - TaskType

# SECTION 03

# LAMBDAS

# Lambda expression

- A new feature introduced in Java 8
- A representation of anonymous function that can be passed around.
- Allows you to pass behaviour as code that can be executed later
- Allows you to encapsulate changing behaviour in a lambda expression
- Earlier this was achieved via the use of anonymous inner classes

# Demo >> Lambda Example 1

# Lambda expression

```
(first, second) -> first.length() - second.length();
```

- The (first, second) are parameters of the compare method
- first.length() - second.length() is the function body
- -> is the lambda operator

# Lambda syntax

- (String a, String b) -> {return a + b;}
- (a,b) -> a + b
- a -> a*a
- () -> a
- () -> {System.out.println("hello world");}

# Type Inference

The act of inferring a type from context is called Type Inference.

In most cases, javac will infer the type from context.

If it can't resolve then compiler will throw an error.

```
Comparator comparator = (first, second) -> first.length() - second.length();
```

Cannot resolve method 'length()'

Demo >> Example2_Lambda

# Lambdas are Typed

```
Comparator<String> nameComparator = (first, second) -> second.length() - first.length();
```

- Type of a lambda expression is an interface
- Only interfaces with single abstract method can be used
- These interfaces are called functional interfaces
- You can use @FunctionalInterface annotation to mark your interface a functional interface

# Example @FuntionalInterface

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

Demo >> Example3_FunctionalInterfaces.java

# Invalid @FunctionalInterface

```
@FunctionalInterface
public interface InvalidFunctionInterface {
```

Multiple non-overriding abstract methods found in interface io.shekhar.trainings.java8.lambdas.examples.InvalidFunctionInterface

```
    public boolean test();

    public boolean test1();


}
```

# Look at few @FunctionalInterface

- Most of the time you don't have to write your own @FunctionalInterface
- You can leverage interfaces defined in java.util.function package

- Some shown below:

```
Predicate<String> nameStartWithS = name -> name.startsWith("s");

Consumer<String> sendEmail = message -> System.out.println("Sending email >> " + message);

Function<String, Integer> stringToLength = name -> name.length();

Supplier<String> uuidSupplier = () -> UUID.randomUUID().toString();
```

# Test your lambda knowledge

>> Exercise1_Lambda.java

# Where can I use lambda expression?

- You can use lambdas wherever you use an interface with single method
- Some examples callbacks, comparisons, filtering, transformation, actions,etc.

Spring templates are example where lambda could be used

# Using local variables >> Effective final

```
public class Example4_EffectiveFinal {

    public static void main(String[] args) {
        int number = 0;
        number++;
        IntConsumer consumer = i -> System.out.println(i + number);
    }

}
```

Variable used in lambda expression should be effectively final

# Method References

# Method reference

- Lets you create lambda expression from existing method implementation
- Shorthand for lambdas calling only a specific method

```
String str -> str.length()
String::length
```

# Demo >>

# Method Reference

```java
public class Exercise_Lambdas {

    public static void main(String[] args) {
        List<Task> tasks = getTasks();
        List<String> titles = taskTitles(tasks);
        titles.forEach(new Consumer<String>() {
            @Override
            public void accept(String s) {
                System.out.println(s);
            }
        });
    }

    public static List<String> taskTitles(List<Task> tasks) {
        List<String> readingTitles = new ArrayList<>();
        for (Task task : tasks) {
            if (task.getType() == TaskType.READING) {
                readingTitles.add(task.getTitle());
            }
        }
        return readingTitles;
    }

}
```

Exercise >> Let's apply what we have learnt so far

# SECTION 04

# STREAM PROCESSING

# Stream Processing

- Why we need a new data processing abstraction?
- Stream vs Collection
- Using Stream API

# Why?

- Collection API is too low level
- Developers needed a higher level declarative data processing API
- Processing data in parallel

# Data processing before Java 8

```
public static void main(String[] args) {
    List<Task> tasks = getTasks();

    List<Task> readingTasks = new ArrayList<>();
    for (Task task : tasks) {
        if(task.getType() == TaskType.READING){
            readingTasks.add(task);
        }
    }

    Collections.sort(readingTasks, (t1,t2) -> t1.getCreatedAt().compareTo(t2.getCreatedAt()));
    for (Task readingTask : readingTasks) {
        System.out.println(readingTask.getTitle());
    }


}
```

What does this code do?

# Data processing in Java 8

```
public static void main(String[] args) {
    List<Task> tasks = getTasks();
    tasks.stream().
            filter(task -> task.getType() == TaskType.READING).
            sorted().
            map(Task::getTitle).
            forEach(System.out::println);
}
```

Isn't this code beautiful?

# Understanding the code

```
public static void main(String[] args) {
    List<Task> tasks = getTasks();
    tasks.stream().
            filter(task -> task.getType() == TaskType.READING).
            sorted().
            map(Task::getTitle).
            forEach(System.out::println);
}
```

Creates a stream on the source collection

perform data processing operations. Each produce a stream

finally stream is evaluated and each element is printed on the console

# Why Java 8 code is better?

- Developer's intent is clear
- Declarative >> What over How
- Reusable chainable higher level construct
- Unified language for data processing
- No boilerplate code.!! Yay :)

# Stream

- Stream is a sequence of elements from a source supporting data processing operations
- source -> collection with elements
- data processing operations -> filter, map, etc.

# Collection vs Stream

| Collection | Stream |
|---|---|
| Read and write operation | Only read operations. You can't add or remove elements. |
| Eagerly evaluated | Lazily evaluated |
| Collections are about data | Streams are for performing computations on data |
| Client has to iterate over collection >> External iteration | internal iteration |
| You can iterate over collection multiple times | You can only process a stream once |

# Stream API

- Stream
  - Builder
  - allMatch(Predicate<? super T>): boolean
  - anyMatch(Predicate<? super T>): boolean
  - builder(): Builder<T>
  - collect(Collector<? super T, A, R>): R
  - collect(Supplier<R>, BiConsumer<R, ? super T>, BiConsumer<R, R>): R
  - concat(Stream<? extends T>, Stream<? extends T>): Stream<T>
  - count(): long
  - distinct(): Stream<T>
  - empty(): Stream<T>
  - filter(Predicate<? super T>): Stream<T>
  - findAny(): Optional<T>
  - findFirst(): Optional<T>
  - flatMap(Function<? super T, ? extends Stream<? extends R>>): Stream<R>
  - flatMapToDouble(Function<? super T, ? extends DoubleStream>): DoubleStream
  - flatMapToInt(Function<? super T, ? extends IntStream>): IntStream
  - flatMapToLong(Function<? super T, ? extends LongStream>): LongStream
  - forEach(Consumer<? super T>): void
  - forEachOrdered(Consumer<? super T>): void
  - generate(Supplier<T>): Stream<T>
  - iterate(T, UnaryOperator<T>): Stream<T>
  - limit(long): Stream<T>
  - map(Function<? super T, ? extends R>): Stream<R>
  - mapToDouble(ToDoubleFunction<? super T>): DoubleStream
  - mapToInt(ToIntFunction<? super T>): IntStream
  - mapToLong(ToLongFunction<? super T>): LongStream
  - max(Comparator<? super T>): Optional<T>
  - min(Comparator<? super T>): Optional<T>
  - noneMatch(Predicate<? super T>): boolean
  - of(T): Stream<T>
  - of(T...): Stream<T>
  - peek(Consumer<? super T>): Stream<T>
  - reduce(BinaryOperator<T>): Optional<T>
  - reduce(T, BinaryOperator<T>): T
  - reduce(U, BiFunction<U, ? super T, U>, BinaryOperator<U>): U
  - skip(long): Stream<T>
  - sorted(): Stream<T>
  - sorted(Comparator<? super T>): Stream<T>
  - toArray(): Object[]
  - toArray(IntFunction<A[]>): A[]

# Stream operations

- Intermediate operations >> which results in another stream
  - map, filter, sort, distinct, etc
- Terminal operations >> which produce a non-stream result
  - collect, foreach,

# filtering


```
filter(x => x > 10)
```

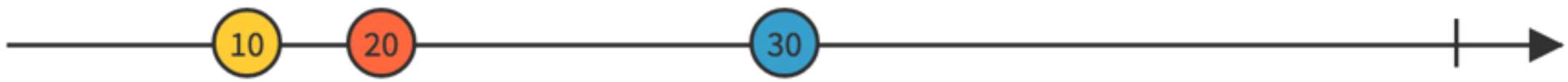

Demo >> Example2_FilterStream

# Filtering operations

- Find all the reading tasks sorted by their creation date
- Find all distinct tasks and print them
- Find top 5 reading tasks sorted by their creation date
  - Pagination with Skip and Limit
- Count all reading tasks

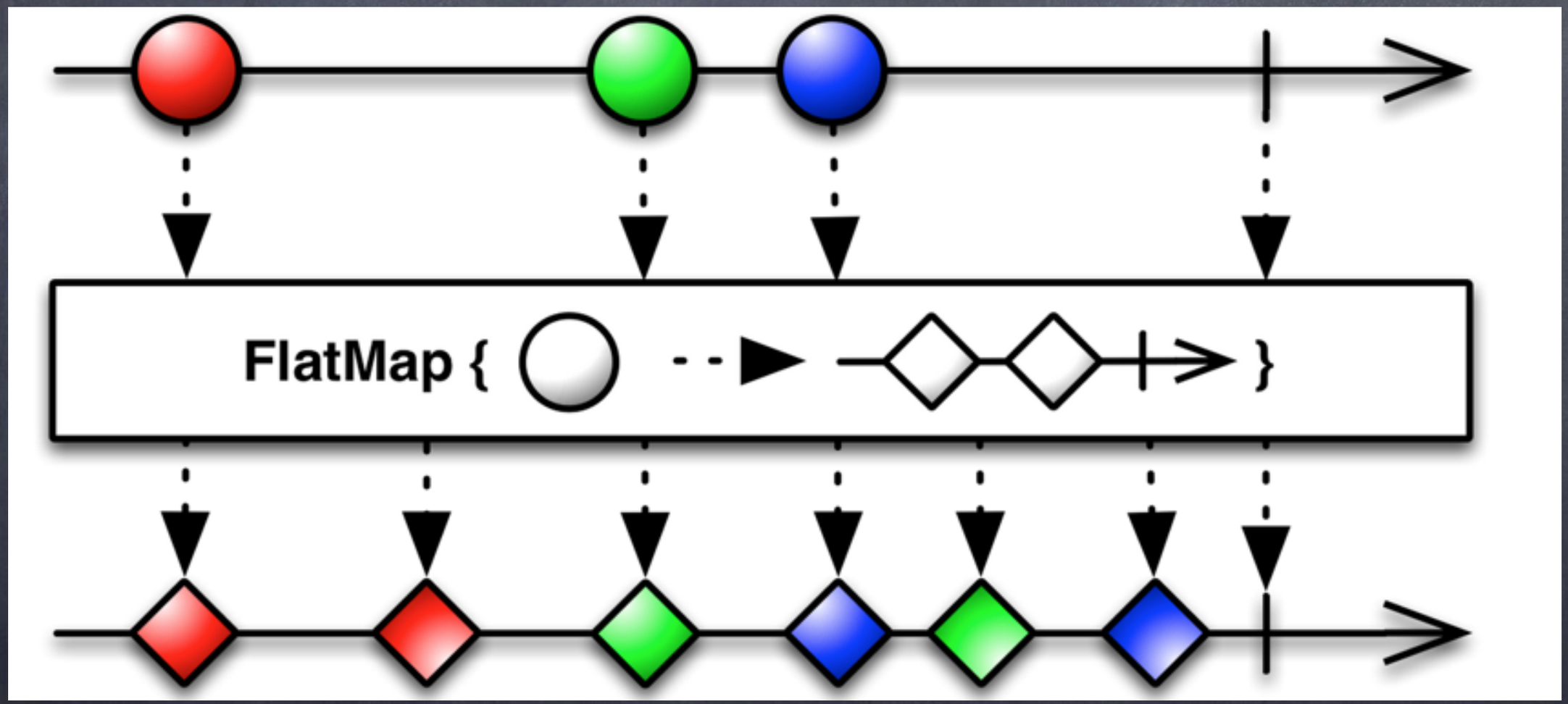
# map operation


```
map(x => 10 * x)
```




Example3_MapStreams

# flatmap operation



# reduce operation


```
reduce((x, y) => x + y)
```

# Duplicate Elements in a List

```
public static void main(String[] args) {
    duplicate(Arrays.asList("a", "b", "c", "d", "e"));
}
```

Output >>

```
a
a
b
b
c
c
d
d
e
e
```

# Duplicate Elements n times in a List

```java
public static void main(String[] args) {
    duplicate(Arrays.asList("a", "b", "c", "d", "e"), 3);
}
```

Output >>

```
a
a
a
b
b
b
c
c
c
d
d
d
e
e
e
```

# Working with numeric streams

- IntStream
- LongStream
- DoubleStream

# Parallel Stream

- You can create an instance of parallel stream by calling the parallel operator

# SECTION 05

# COLLECTORS

# Collector

```
List<String> readingTasks = tasks.stream().
        filter(task -> task.getType() == TaskType.READING).
        map(Task::getTitle).
        collect(Collectors.toList());
```

- Reduce a stream to a single value
  - collect is a reduction operation
- They are used with collect terminal method

# Demo >>
# Power of Collector

Group tasks by Type >> Java 7 and Java 8

# What you can do with collect?

- Reducing stream to a single value
- Group elements in a stream
- Partition elements in a stream

# Collectors class

- A utility class that contains static factory methods for most common collectors
- that collect to a Collection
- grouping
- partitioning

# Reducing to a single value

- Single value could be
  - numeric type
  - domain object
  - a collection

Demo >> Example2_ReduceValue

# Grouping elements

- Group elements by key
- You can do both single level and multilevel grouping

Demo >> Example3

# Partitioning

- It is a special case of grouping
- Groups source collection into at most two partitioned by a predicate
- Returned map has following syntax
  - Map<Boolean, List<Task>>

Demo >> Example4

# Sec 06

# Optional<T>

# How many of you have experienced NullPointerException?

# Null History

- It was designed by Sir Tony Hoare in 1965
  - creator of Quicksort
- He was designing Algol W language
- null was designed to signify absence of value
- Most programming languages like Java, C++, C#, Scala, etc all have Null
- He called it was a Billion Dollar Mistake

http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

# What could possibly go wrong?

```
public String taskAssignedTo(String taskId) {
  return taskRepository.find(taskId).getAssignedTo().getUsername();
}
```

# NullPointerException

```
public String taskAssignedTo(String taskId) {
  return taskRepository.find(taskId).getAssignedTo().getUsername();
}
```

Task could be null

```
public String taskAssignedTo(String taskId) {
  return taskRepository.find(taskId).getAssignedTo().getUsername();
}
```

User could be null

# We end up writing ...

```
public String taskAssignedTo(String taskId) {
  Task task = taskRepository.find(taskId)
  if(task != null){
    User user = task.getAssignedTo() ;
    if(user != null){
        return user.getUsername();
    }
  }
  return null;
}
```

# Biggest problem with code

Absence of value is not visible in the API

# Possible solutions

- Null Object Pattern << Before Java 8
- Optional << From Java 8

# Optional

- A container that may or may not contain a value
- If a function returns Optional then the client would know that value might not be present
- Common concept is functional languages
  - MayBe, Nothing >> Haskell
  - Option, Some, None >> Scala
  - Optional<T> >> Java 8

# Demo >>

# Optional TaskRepository

# Another Example

```java
@Override
public boolean equals(Object o) {
    return Optional.ofNullable(o)
            .filter(that -> that instanceof Task)
            .map(that -> (Task) that)
            .filter(that -> Objects.equals(this.title, that.title))
            .filter(that -> Objects.equals(this.type, that.type))
            .isPresent();
}
```

# SECTION 07

# DATE AND TIME API

# What does the below program prints?

```
public class DatePain {

    public static void main(String[] args) {
        Date date = new Date(12, 12, 12);
        System.out.println(date);
    }
}
```

# Existing Date API Problems

```
public class DatePain {

    public static void main(String[] args) {
        Date date = new Date(12, 12, 12);
        System.out.println(date);
    }
}
```

Sun Jan 12 00:00:00 IST 1913

- Which 12 is for date field?
- 12 is for December right?? No. It's January
- Year 12 is 12 CE?? Wrong 1913.. starts from 1900
- Hmmm. why there is time in date??
- There is timezone as well?? Who asked??

# Existing Date API

- Date API was introduced in JDK in 1996
- There are many issues with
  - Mutability
  - Date is not date but date with time
  - Separate Date class hierarchy for SQL
  - No concept of timezone
  - Boilerplate friendly

# Calendar API

- Still mutable
- Can't format a date directly
- You can't perform arithmetic operations on date.
- Calendar instance does not work with formatter

# New API – Getting Started

- Developed as part of JSR 310
- Heavily inspired by Joda-Time library
- New package – java.time

# New types for humans

- LocalDate - a date with no time or timezone

- LocalTime - a time with no date or timezone

- LocalDateTime - LocaDate + LocalTime

All types are immutable

# Demo >>
# LocalDate, LocalTime, and LocalDateTime

# New Type >> Instant

- A machine friendly way to describe date and time
- Number of seconds passed since epoch time
- Nanosecond precision
- Useful to represent event timestamp

# Demo >> Instant

```
public static void main(String[] args) {
    Instant instant = Instant.ofEpochSecond(3);
    System.out.println(instant);//1970-01-01T00:00:03Z

    Instant now = Instant.now();
    System.out.println(now);//2015-09-19T17:18:18.425Z

    System.out.println(Instant.parse("2015-09-12T10:15:30.00Z"));//2015-09-12T10:15:30Z
}
```

# Duration and Period

- Duration represents quantity or amount of time in seconds or nano-seconds like 10 seconds
  - Duration d = Duration.between(dt1,dt2)
- Period represents amount or quantity of time in years, months, and days
  - Period p = Period.between(ld1,ld2)

# Demo >>

# Temporal Adjuster

- When you need to do advance date-time manipulation then you will use it
- Examples
  - adjust to next sunday
  - adjust to first day of next month
  - adjust to next working day

# Demo >>