

JAVA 8

Evgeny Borisov
evgeny@inwite.pro

About myself

Big data technical leader in Naya
CTO in “democra.c”

Partner of JFrog

Consulting

Lecturing

Writing courses

Writing code



Терминология

- Аппликация = приложение
- **Айбернет** = хибернет
- Штрудель = Собака
- Компонент – использую с любым ударением
- Параметр = Параметр
- Список пополняется...

Agenda

- Callback method pattern
- Closures
- Lambda Expression & Functional Interface
- Existing Functional Interfaces
- Streams
- Map reduce & Predicates
- Concurrency and parallelism
- Performance Benchmark
- G1
- Default methods in interfaces
- JSR-310
- What new in Reflection
- Repeatable annotations

Callback method pattern

- What can you pass to the method, as its args?
 - primitive, references to objects
 - What about some algorithm or method, function?



Callback method pattern

- Write method which will calculate duplicates of object in list
- `public int countDuplicates(T, List<T>...)`

Just implement equal method



```
public static <T> int countDuplicates(T t, List<T> list, Equalator<T> equalator) {  
    int counter=0;  
    for (T o : list) {  
        if (equalator.equals(o, t)) {  
            counter++;  
        }  
    }  
    return counter;  
}
```

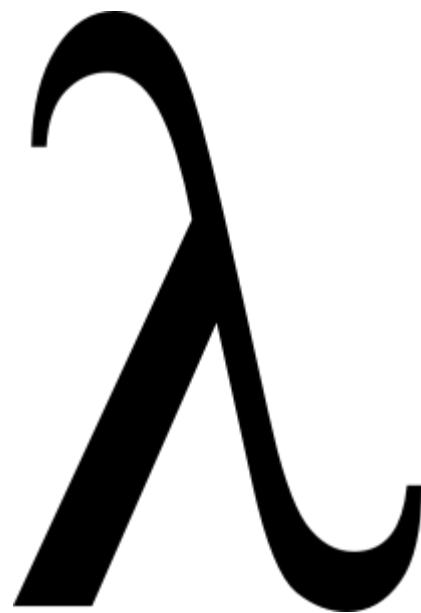
```
countDuplicates("java", strings, new Equalator<String>() {  
    @Override  
    public boolean equals(String t1, String t2) {  
        return t1.length() - t2.length()>0;  
    }  
});
```



```
countDuplicates("java", strings, new Equalator<String>() {  
    @Override  
    public boolean equals(String t1, String t2) {  
        return t1.length() - t2.length() > 0;  
    }  
});
```

```
countDuplicates("java", strings, (t1, t2) -> t1.length() == t2.length());
```

Lambda - גימל



What is lambda?

- ~~Anonymous function~~
- ~~Expression, which describes anonymous function~~
- Expression, which describes anonymous function, when invoked will return some object, which class implements required functional interface

```
countDuplicates("java", strings, (t1, t2) -> t1.length() == t2.length());
```

What is lambda?

- ~~Anonymous function~~
- ~~Expression, which describes anonymous function~~
- Expression, which describes anonymous function, when invoked will return some object (of unknown nature), which class implements required functional interface

```
countDuplicates("java", strings, (t1, t2) -> t1.length() == t2.length() > 0);
```

```
countDuplicates("java", strings, (t1, t2) -> t1.length() == t2.length());
```

Syntax

- `() ->`
- `() -> {}`
- `someName - > someName.someMethod()`
- `someName - > {someName.someMethod();}`
- `(a,b,c) - > {// some code using a,b,c is here;}`
- `(Type1 a, Type2 b, Type3 c) -> ...`

syntax of lambda without arguments

() -> 42

```
public class TaxService {  
    public double afterMaam(double price, Supplier<Double> maamSupplier) {  
        return maamSupplier.get() * price + price;  
    }  
}
```

```
Supplier<Double> maamSupplier = () -> 0.18;  
double afterMaam = taxService.afterMaam(100, maamSupplier);
```

syntax of lambda without arguments

() -> 42

```
public class TaxService {  
    public double afterMaam(double price, Supplier<Double> maamSupplier) {  
        return maamSupplier.get() * price + price;  
    }  
}
```

```
taxService.afterMaam(priceBeforeMaam, () -> 0.18);
```


syntax of lambda without arguments

() -> 42

```
public class TaxService {  
    public double afterMaam(double price, Supplier<Double> maamSupplier) {  
        return maamSupplier.get() * price + price;  
    }  
}
```

```
double afterMaam = taxService.afterMaam(100, () -> {  
    double maam=0;  
    //some code which calculates maam  
    return maam;  
});
```

syntax of lambda with one arguments

(Type t) -> t... (t)->t... t->t...

```
List<String> names = Arrays.asList("Gena", "Jenia", "Hadas", "Eldad");
```

```
names.forEach((String name) -> System.out.println(name));
```

syntax of lambda with one arguments

(Type t) -> t... **(t)->t...** t->t...

```
List<String> names = Arrays.asList("Gena", "Jenia", "Hadas", "Eldad");
```

```
names.forEach((name) -> System.out.println(name));
```

syntax of lambda with one arguments

(Type t) -> t... (t)->t... **t->t...**

```
List<String> names = Arrays.asList("Gena", "Jenia", "Hadas", "Eldad");
```

```
names.forEach(name->System.out.println(name));
```

syntax of lambda with one arguments

```
List<String> names = Arrays.asList("Gena", "Jenia", "Hadas", "Eldad");  
  
names.forEach(System.out::println);
```

syntax of lambda with several arguments

`(Type1 t1, Type2 t2) -> t1.someMethod(t2)`

or

`(t1, t2) -> t1.someMethod(t2)`

syntax of lambda with several arguments

(Type1 t1, Type2 t2) -> t1.someMethod(t2)

or

(t1, t2) -> t1.someMethod(t2)

```
List<String> names = Arrays.asList("Gena", "Jenia", "Hadas", "Eldad");
```

```
names.sort((String s1, String s2) -> s1.length()-s2.length());
```

syntax of lambda with several arguments

`(Type1 t1, Type2 t2) -> t1.someMethod(t2)`

or

`(t1, t2) -> t1.someMethod(t2)`

```
List<String> names = Arrays.asList("Gena", "Jenia", "Hadas", "Eldad");
```

```
names.sort((s1, s2) -> s1.length()-s2.length());
```


syntax of lambda with several arguments

It can be a block of code

```
(t1, t2) -> {  
    //some code  
}
```

```
List<String> names = Arrays.asList("Gena", "Jenia", "Hadas", "Eldad");  
names.sort((s1, s2) -> {  
    int diff=0;  
    //some logic here  
    return diff;  
});
```

Lab

- Write method which will receive a list, lambda and delay(int)
- Method will invoke lambda on each object from the list with delay between each invocation
- `public static <T> void forEachWithDelay(List<T> list, int delay,...)`

```
LambdaUtils.forEachWithDelay(strings, 5, s -> System.out.println(s));
```

What is inside lambda?





- **this**, inside lambda will point on instance of outer class
- Lambda is not serializable by default (security reason)
- Lambda can't declare throws (if your lambda code throws checked exception, you must handle it inside lambda)
- It's not only syntactic sugar
- It's not implemented with anonymous classes (actually almost yes, but...)

Functional interface

- Only one method to implement
 - ActionListener
 - Comparator
 - Supplier
 - Consumer
- @FunctionalInterface above interface type –WHY?
 - In order to mark them (code readability)
 - In case there will be more one method to implement you will not be allowed by compiler to mark it with @FunctionalInterface
- Can replace such interface with lambda

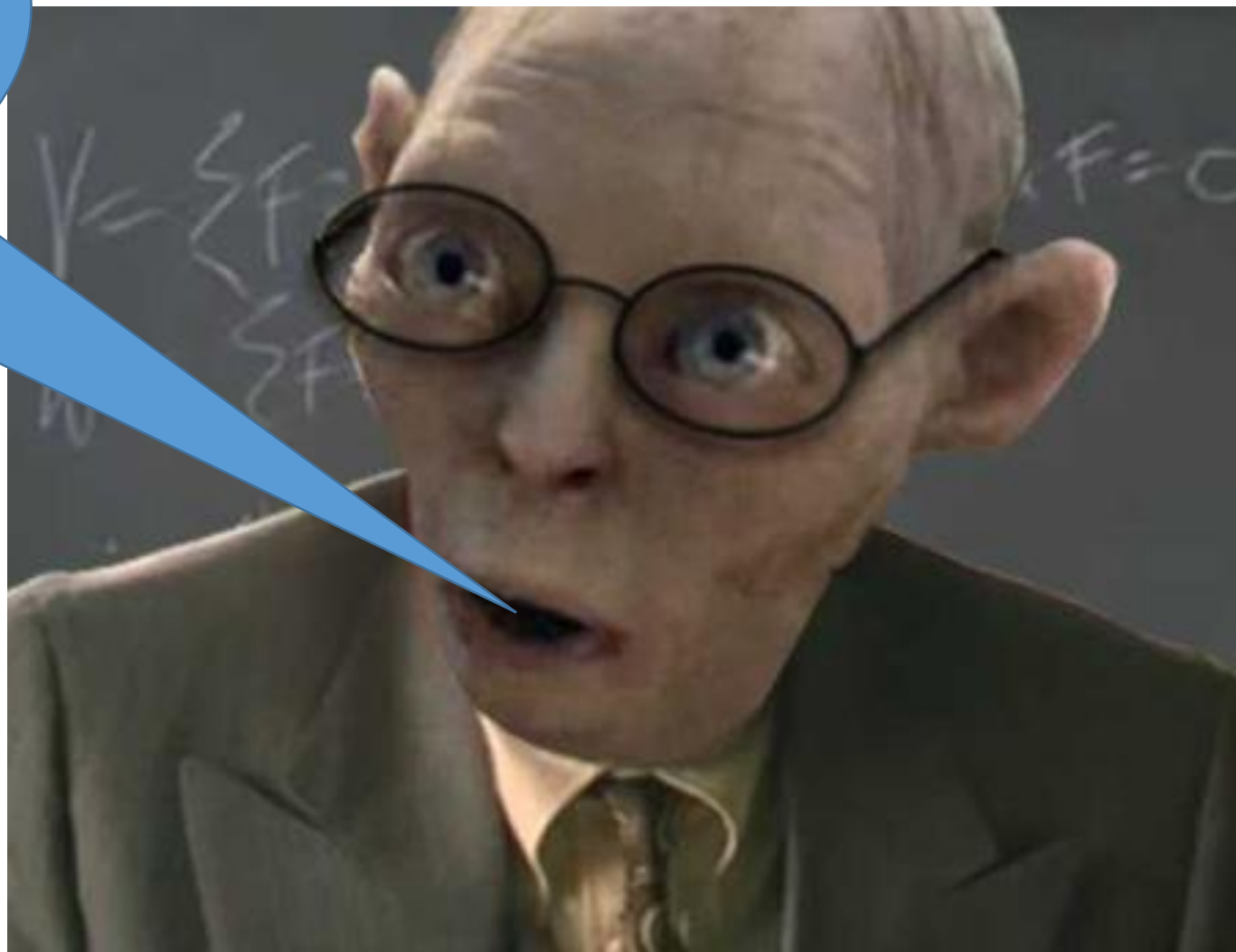
```
button.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        repaint();  
    }  
});
```

```
button.addActionListener(e -> repaint());
```

Comparator can not be
functional interface.
It has many methods
with code!!!



Default methods –
Не считаются



Now we have multiple
inheritance...



Default method

- Do we have multiple inheritance?
- Why do we need defaults methods?
- Lets talk about it...

Comparator can not be
functional interface.
It still has 2 abstract
methods: compare, equals



Riddle

What's wrong here?

```
public void tapelTapel(int num, List<Integer> nums) {  
    nums.forEach(num -> {  
        //some code here  
    });  
  
}
```

Riddle

Already defined in this scope

```
public void tapelTapel(int num List<Integer> nums) {  
    nums.forEach(num -> {  
        //some code here  
    });  
  
}
```

What will not compile?

```
public void tapelTapel(int x, List<Integer> nums) {  
    nums.forEach(num -> {  
        x++;  
    });  
}
```

```
public int tapelTapel(int x, List<Integer> nums) {  
    nums.forEach(System.out::print);  
    return ++x;  
}
```

```
public void tapelTapel(int x, List<Integer> nums) {  
    for (Integer num : nums) {  
        x++;  
    }  
}
```

```
public void tapelTapel(int x, List<Integer> nums) {  
    nums.forEach(num->num=x);  
}
```

What will not compile?

```
public void tapelTapel(int x, List<Integer> nums) {  
    nums.forEach(num -> {  
        x++;  
    });  
}
```

```
public int tapelTapel(int x, List<Integer> nums) {  
    nums.forEach(System.out::print);  
    return ++x;  
}
```

```
public void tapelTapel(int x, List<Integer> nums) {  
    for (Integer num : nums) {  
        x++;  
    }  
}
```

```
public void tapelTapel(int x, List<Integer> nums) {  
    nums.forEach(num->num=x);  
}
```

Riddle

What's wrong here?

```
public void tapelTapel(int x, List<Integer> nums) {  
    nums.forEach(num -> {  
        x++;  
    });  
  
}
```


Effective final

Variable used inside lambda automatically become final

```
public void tapelTapel(int x, List<Integer> nums) {  
    nums.forEach(num -> {  
        x++  
    });  
}
```

What will not compile?

```
public void tapelTapel(int x, List<Integer> nums) {  
    nums.forEach(num -> {  
        x++;  
    });  
}
```

```
public int tapelTapel(int x, List<Integer> nums) {  
    nums.forEach(System.out::print);  
    return ++x;  
}
```

```
public void tapelTapel(int x, List<Integer> nums) {  
    for (Integer num : nums) {  
        x++;  
    }  
}
```

```
public void tapelTapel(int x, List<Integer> nums) {  
    nums.forEach(num->num=x);  
}
```

When lambda can be used?

- assignment: `FunctionalInterface fi = () -> 73;`



When lambda can be used?

- assignment: `FunctionalInterface fi = () -> 73;`
- array initializer: `FI[] fis = { () -> 73, () -> 42 };`
- return: `() -> 73;`
- method argument: `foo(42, () -> 73)`
- constructor argument: `new Person("Avishay", () -> 37)`
- cast: `(FI) () -> 73`

Write sort method using lambda

- Method will receive list of objects and will print them sorted.
- Do not create Anonymous class for Comparator

Exercise

- Write Equalator functional interface
- Write DuplicateCounter service with method
- `int count(List<T> list, T t, Equalator lambda)`
- Test it

How does Lambda work?




```
@FunctionalInterface // this will be Functional Interface
                        // even without this annotation
```

```
public interface Rattleable {
    void rattle();
}
```

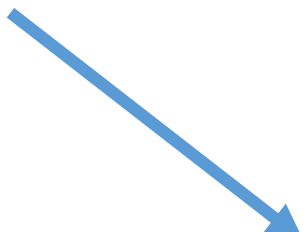
```
Rattleable r = () -> System.out.println("Trr trr");
r.rattle();
```

javac



```
Rattleable r = new R$1();
r.rattle();
```

```
public class R$1 implements Rattleable {
    @Override
    public void rattle() {
        System.out.println("Trr trr");
    }
}
```



A man in a dark suit, white shirt, and dark tie is shown from the chest up. He has a determined, almost angry expression on his face, with his mouth open as if shouting or speaking forcefully. His eyes are squinted, and his eyebrows are furrowed. The background is a blurred office setting with glass partitions and a person in the background.

**You think you can rattle me?
I'm un-rattleable.**

USA

```
@FunctionalInterface // this will be Functional Interface
                        // even without this annotation
```

```
public interface Rattleable {
    void rattle();
}
```

```
Rattleable r = () -> System.out.println("Trr trr");
r.rattle();
```

javac

```
Rattleable r = new R$1();
r.rattle();
```

```
public class R$1 implements Rattleable {
    @Override
    public void rattle() {
        System.out.println("Trr trr");
    }
}
```

```
@FunctionalInterface // this will be Functional Interface  
                        // even without this annotation
```

```
public interface Rattleable {  
    void rattle();  
}
```

```
Rattleable r = () -> System.out.println("Trr trr");  
r.rattle();
```

javac

```
Rattleable r = LambdaFactory;  
r.rattle();
```



Recursive Lambdas



IntUnaryOperator

- Functional interface

```
int applyAsInt (int operand) ;
```

Is it ok?

```
public static IntUnaryOperator getFactorialFunc() {  
    IntUnaryOperator factorial = x -> {  
        if (x == 1) return x;  
        else return x * factorial.applyAsInt(x - 1);  
    };  
    return factorial;  
}
```

May not have been initialized

```
public static IntUnaryOperator getFactorialFunc() {  
    IntUnaryOperator factorial = x -> {  
        if (x == 1) return x;  
        else return x * factorial.applyAsInt(x - 1);  
    };  
    return factorial;  
}
```


Will it help?

```
public static IntUnaryOperator getFactorialFunc() {  
    IntUnaryOperator factorial = null;  
    factorial = x -> {  
        if (x == 1) return x;  
        else return x * factorial.applyAsInt(x - 1);  
    };  
    return factorial;  
}
```

Effectively final

```
public static IntUnaryOperator getFactorialFunc() {  
    IntUnaryOperator factorial = null;  
    factorial = x -> {  
        if (x == 1) return x;  
        else return x * factorial.applyAsInt(x - 1);  
    };  
    return factorial;  
}
```

Solution

```
public static IntUnaryOperator getFactorialFunc() {  
    IntUnaryOperator factorial = x -> {  
        if (x == 1) return x;  
        else return x * getFactorialFunc().applyAsInt(x - 1);  
    };  
    return factorial;  
}
```

Even shorter

```
public static IntUnaryOperator getFactorialFunc() {  
    return x -> {  
        if (x == 1) return x;  
        else return x * getFactorialFunc().applyAsInt(x - 1);  
    };  
}  
  
int i = getFactorialFunc().applyAsInt(5);
```

Exercise

- Write Utility class with getSumLambda method
- This method should return IntUnaryOperator
- applyAsInt method should return sum till the number
- $\text{applyAsInt}(5) = 5+4+3+2+1$

Exercise

- Write IntegerComparator and test it
- How do you compare Integers?
- Integer.compare(x,y)
- Can you do it shorter, without lambda?

Method reference



Static method reference

```
public static void printSQRT(double x) {  
    System.out.println(Math.sqrt(x));  
}
```

```
numbers.forEach(MyMath::printSQRT);
```


Non static method reference

```
PrintStream out = System.out;  
Consumer<Integer> consumer = out::println;  
numbers.forEach(consumer);
```



```
Consumer<Integer> consumer = System.out::println;  
numbers.forEach(consumer);
```



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

Non static method reference

```
SomeService service = new SomeService();  
numbers.forEach(service::tapelTapel);
```

Predicate

- Functional Interface

```
boolean test(T t);
```

```
default Predicate<T> or(Predicate<? super T> other) {  
    Objects.requireNonNull(other);  
    return (t) -> test(t) || other.test(t);  
}
```

```
default Predicate<T> and(Predicate<? super T> other) {  
    Objects.requireNonNull(other);  
    return (t) -> test(t) && other.test(t);  
}
```

Non static method reference

```
Predicate<String> isTrue = "true"::equalsIgnoreCase;
```

```
assertTrue(isTrue.test("TRUE"));
```

```
assertTrue(isTrue.test("true"));
```

```
assertTrue(isTrue.test("TrUe"));
```

```
assertFalse(isTrue.test("bla"));
```

```
Predicate<String> matcher(String str) {
```

```
    return str::equalsIgnoreCase;
```

```
}
```

```
assertTrue(matcher("Eldad Dor").test("ELDAD DOR"));
```

Lambda Expressions and Method Pointers

- JdbcTemplate, JmsTemplate, TransactionTemplate...

Before

After



Looks familiar?

- `jt.query("SELECT * FROM person WHERE dep = ?",`
- What's the second parameter?

Spring 3

```
@Component
public class Example1 {
    @Autowired
    private DataSource dataSource;

    public void testJDBC() {
        JdbcTemplate jt = new JdbcTemplate(dataSource);
        List<Person> persons = jt.query("SELECT * FROM person WHERE dep = ?",
            new PreparedStatementSetter() {
                @Override
                public void setValues(PreparedStatement ps) throws SQLException {
                    ps.setString(1, "Sales");
                }
            }, new RowMapper<Person>() {
                @Override
                public Person mapRow(ResultSet rs, int rowNum) throws SQLException {
                    return new Person(rs.getString(1), rs.getString(2));
                }
            }
        );
    }
}
```

Spring 4

```
@Component
public class Example2 {
    @Autowired
    private DataSource dataSource;

    public List<Person> getPersonList(String department) {
        JdbcTemplate jt = new JdbcTemplate(this.dataSource);
        return jt.query("SELECT name, age FROM person WHERE dep = ?",
            ps -> {
                ps.setString(1, "Sales");
            },
            this::mapPerson);
    }
}
```


What if we need non-static method, but we don't have an instance?



Integer.compareTo is non-static method

```
public int compareTo(Integer anotherInteger) {  
    return compare(this.value, anotherInteger.value);  
}
```

Method reference: unbounded

```
Comparator<Integer> comparator = Integer::compareTo;  
  
numbers.sort(comparator);
```

```
numbers.sort(Integer::compareTo);
```

Function<T,R>

Functional Interface

```
R apply(T t);
```

```
default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {  
    Objects.requireNonNull(after);  
    return (T t) -> after.apply(apply(t));  
}
```

```
default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {  
    Objects.requireNonNull(before);  
    return (V v) -> apply(before.apply(v));  
}
```

Method reference to constructor

```
1. Function<String,Integer> f = Integer::new;  
2. Integer one = f.apply("1");  
3. Supplier<Integer> supplier = Integer::new;  
4. Integer zero = supplier.get();
```

Is there any compilation problem?

Method reference to constructor

```
1. Function<String,Integer> f = Integer::new;  
2. Integer one = f.apply("1");  
3. Supplier<Integer> supplier = Integer::new;  
4. Integer zero = supplier.get();
```

Is there any compilation problem?

Method reference to constructor

- Supplier Interface requires method without parameter, so we need an empty constructor. But Integer doesn't have it.
- Function Interface requires method with one parameter, and Integer constructor has it.

Method reference to constructor

```
1. Function<String,String> f = String::new;  
2. String one = f.apply("1");  
3. Supplier<String> supplier = String::new;  
4. String zero = supplier.get();
```

Is there any compilation problem?

Everything is ok

- String class has both constructors

Lambda Serialization problem

```
NavigableSet<Integer> set = new TreeSet<>((a,b)->Integer.compareUnsigned(a,b));
```

Lambda Serialization problem

Can you see problem here?

```
NavigableSet<Integer> set = new TreeSet<>(Integer::compareUnsigned);

ObjectOutputStream oos; // some code here
try {
    oos.writeObject(set);
} catch (NotSerializableException e) {
    // we here
}
```



Lambda Serialization solution – type intersection

```
NavigableSet<Integer> set =  
    new TreeSet<>((Comparator<Integer> & Serializable) Integer::compareUnsigned);  
ObjectOutputStream oos; // some code here  
try {  
    oos.writeObject(set);  
} catch (NotSerializableException e) {  
    // we are not here
```



**After intersection it must be only one method:
Comparator (1 method) + Serializable (0 methods) = 1 method**

```
}
```

Порошок

- Восьмая джава это круто
- Она продвинута весьма
- Аж даже final добавляет
- Сама

New Task

- You have employee class {Name, Company, Insurance}
- Write method which will receive List of Employees
- And print out company names after sorting and making distinct operations only of IDI clients
- Sort must be done by company revenue

Java 7 solution

```
public void printIDIClients(List<Employee> employees) {  
    Set<Company> companies = new HashSet<>();  
    for (Employee employee : employees) {  
        if (employee.getInsurance() == Insurance.IDI) {  
            companies.add(employee.getCompany());  
        }  
    }  
    ArrayList<Company> sorted = new ArrayList<>(companies);  
    Collections.sort(sorted, new Comparator<Company>() {  
        @Override  
        public int compare(Company o1, Company o2) {  
            return Integer.compare(o1.getRevenue(), o2.getRevenue());  
        }  
    });  
    for (Company company : sorted) {  
        System.out.println(company.getName());  
    }  
}
```

Java 8 solution

```
public void printIDIClients(List<Employee> employees) {  
    employees.stream().  
        filter(employee -> employee.getInsurance() == Insurance.IDI).  
        map(Employee::getCompany).  
        sorted(comparing(Company::getRevenue)).  
        map(Company::getName).  
        distinct().  
        forEach(System.out::println);  
}
```


Streams



What's wrong with Java 7?

Why do we need streams?

- Shorter
- Syntax sugar
- More readable
- Laziness
- Code optimization as a reason of laziness
- Parallelism

Parallelism

- Why compiler can't do it?

```
Collection<Employee> data;  
...  
for (Employee employee : data) {  
    processEmployee(employee);  
}
```

Parallelism

```
data.parallelStream()  
    .forEach(employee -> processEmployee(employee)) ;
```

Streams design

source → op → op → op → gangamstyle

Streams design

source → op → op → op → Terminal operation

- Sources: collections, iterators, channels, files...
- Operations: filter, map, reduce...
- Terminal operation: collections, variable, lambda...

Stream

- «Multiplicity of values»
- It's not data structure (no storage)
- Laziness
- Can be infinite
- Doesn't change the source
- Can be used only once
- Ordered / Unordered
- Parallel / Sequential
- There are primitive specializations: IntStream, LongStream, DoubleStream

Stream pipeline

1 - A source: Source → Stream

∞ - Intermediate operation: Stream → Stream

1 - Terminal operation: Stream → Profit

```
public void printIDIClients(List<Employee> employees) {  
    employees.stream().  
        filter(employee -> employee.getInsurance() == Insurance.IDI).  
        map(Employee::getCompany).  
        sorted(comparing(Company::getRevenue)).  
        map(Company::getName).  
        distinct().  
        forEach(System.out::println);  
}
```


Stream pipeline

1 - A source: Source → Stream

∞ - Intermediate operation: Stream → Stream

1 - Terminal operation: Stream → Profit

```
public void printIDIClients2(List<Employee> employees) {  
    employees.stream().  
        filter(employee -> employee.getInsurance() == Insurance.IDI).  
        map(Employee::getCompany).  
        sorted(comparing(Company::getRevenue)).  
        map(Company::getName).  
        distinct().  
        forEach(System.out::println);  
}
```

Stream pipeline

1 - A source: Source → Stream

∞ - **Intermediate operation**: Stream → Stream

1 - Terminal operation: Stream → Profit

```
public void printIDIClients2(List<Employee> employees) {  
    employees.stream().  
        filter(employee -> employee.getInsurance() == Insurance.IDI).  
        map(Employee::getCompany).  
        sorted(comparing(Company::getRevenue)).  
        map(Company::getName).  
        distinct().  
        forEach(System.out::println);  
}
```

Stream pipeline

1 - A source: Source → Stream

∞ - Intermediate operation: Stream → Stream

1 - Terminal operation: Stream → Profit

```
public void printIDIClients2(List<Employee> employees) {  
    employees.stream().  
        filter(employee -> employee.getInsurance() == Insurance.IDI).  
        map(Employee::getCompany).  
        sorted(comparing(Company::getRevenue)).  
        map(Company::getName).  
        distinct().  
        forEach(System.out::println);  
}
```

Riddle – what will happen?

```
employees.stream().  
    filter(employee -> employee.getInsurance() == Insurance.IDI).  
    map(Employee::getCompany).  
    sorted(comparing(Company::getRevenue)).  
    map(company -> {  
        throw new RuntimeException("PROBLEM");  
    }).  
    distinct();
```



Riddle – what will happen?

```
Stream<Employee> s1 = employees.stream();
Stream<Employee> s2 = s1.filter(employee -> employee.getInsurance() == Insurance.IDI);
Stream<Company> s3 = s2.map(Employee::getCompany);
Stream<Company> s4 = s3.sorted(comparing(Company::getRevenue));
Stream<Object> s5 = s4.map(company -> {
    throw new RuntimeException("PROBLEM");
});
Stream<Object> s6 = s5.distinct();
```



Only terminal operation will throw an exception

```
Stream<Employee> s1 = employees.stream();
Stream<Employee> s2 = s1.filter(employee -> employee.getInsurance() == Insurance.IDI);
Stream<Company> s3 = s2.map(Employee::getCompany);
Stream<Company> s4 = s3.sorted(comparing(Company::getRevenue));
Stream<Object> s5 = s4.map(company -> {
    throw new RuntimeException("PROBLEM");
});
Stream<Object> s6 = s5.distinct();
s6.forEach(System.out::println);
```


Declaring Stream variable. Is it good idea?

- Stream is not reusable!!!
- You can't use the same stream twice.

```
Stream<Employee> s1 = employees.stream();  
Stream<Employee> s2 = s1.filter(employee -> employee.getInsurance() == Insurance.IDI);  
Stream<Company> s3 = s2.map(Employee::getCompany);  
Stream<Company> s4 = s3.sorted(comparing(Company::getRevenue));  
Stream<Object> s5 = s4.distinct();
```

How can I know if it is intermediate stream method or not?

Intermediate methods return stream 😊

- Otherwise it is terminal operation
- After terminal operation was called this stream can't be used anymore

Stream sources: collections

```
ArrayList<Employee> list;  
Stream<Employee> stream = list.stream();    // sized, ordered
```

```
HashSet<Employee> set;  
Stream<Employee> stream = set.stream();    // sized, distinct
```

[illegible]

Stream sources: factories, builders

```
Employee[] arr;  
Stream<Employee> stream = Arrays.stream(arr);
```

```
Stream<Integer> stream = Stream.of(1, 10, 20);  
Stream<Employee> stream = Stream.of(new Employee("Nisan"), new Employee("Eldad"));
```

```
Stream<Object> stream = Stream.builder().add(emp1).add(emp2).build();
```

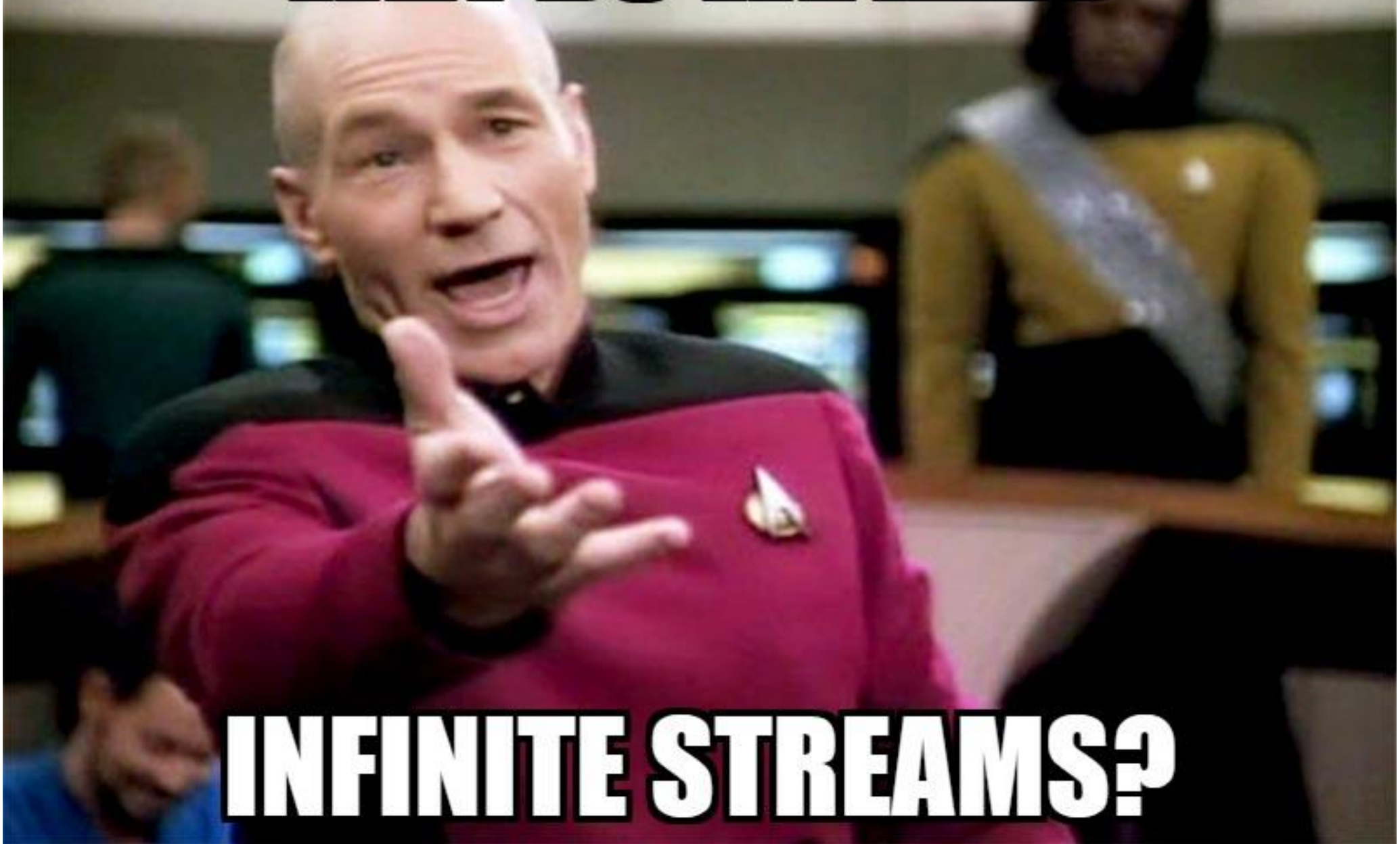
```
IntStream range = IntStream.range(1, 666);
```

Stream Sources: generators

```
Random random = new Random();  
DoubleStream doubleStream = DoubleStream.generate(random::nextDouble);
```

```
Stream<Integer> intStream = Stream.iterate(10, i -> i + 2);
```

WHY DO WE NEED



INFINITE STREAMS?

We can throw them away or limit them

```
stream.findFirst()...
```

```
OptionalDouble max = doubleStream.limit(10000).max();  
System.out.println(max.getAsDouble());
```

Stream sources: others

```
Stream<String> stream = bufferedReader.lines();
```

```
Stream<String> stream = Pattern.compile(regex).splitAsStream(str);
```

```
DoubleStream doubleStream = new SplittableRandom().doubles();
```


Intermediate Operations

- `Stream<T> stream.filter(Predicate<T>);`
- `Stream<T> stream.map(Function<S,T>);`
- `Stream<T> stream.flatMap(Function<S,Stream<T>>);`
- `Stream<T> stream.peek(Function<S,T>);`
- `Stream<T> stream.sorted();`
- `Stream<T> stream.distinct();`
- `Stream<T> stream.limit(long);`
- `Stream<T> stream.skip(long);`

What???

- `Stream<T> stream.filter(Predicate<T>);`
- **`Stream<T> stream.map(Function<S,T>);`**
- **`Stream<T> stream.flatMap(Function<S,Stream<T>>);`**
- **`Stream<T> stream.peek(Function<S,T>);`**
- `Stream<T> stream.sorted();`
- `Stream<T> stream.distinct();`
- `Stream<T> stream.limit(long);`
- `Stream<T> stream.skip(long);`

map

- Returns a stream consisting of the results of applying the given function to the elements of this stream.

```
Stream<Employee> stream = employees.stream();  
Stream<String> stringStream = stream.map(employee -> employee.getName());
```

map

- Returns a stream consisting of the results of applying the given function to the elements of this stream.

```
Stream<Employee> stream = employees.stream();  
Stream<String> stringStream = stream.map(Employee::getName);
```

Lab

- Write method which receives list of employees and returns sum of their salaries

flatMap

- has the effect of applying a one-to-many transformation to the elements of the stream, and then flattening the resulting elements into a new stream

```
Stream<String> stringStream =  
    employeeStream.flatMap(emp -> emp.getChildrenNames().stream());
```

Lab

Class employee has salaries (array property for salary of each month)

```
Employee{  
    private int[] salary = new int[12]  
    ...  
}
```

Write method which will calculate salaries of all employees per year

lab

- Write method which will receive a file and return number of words
- Write method which will receive a file and return average length of the word

Peek

- Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.
- **Will not affect original stream**
- **Good for debug**

Intermediate Operations

- `Stream<T> stream.filter(Predicate<T>);`
 - `Stream<T> stream.map(Function<S,T>);`
 - `Stream<T> stream.flatMap(Function<S,Stream<T>>);`
 - `Stream<T> stream.peek(Function<S,T>);`
 - `Stream<T> stream.sorted();`
 - `Stream<T> stream.distinct();`
 - `Stream<T> stream.limit(long);`
 - `Stream<T> stream.skip(long);`
-
- `Stream<T> stream.unordered();` //some performance benefits (it's not shuffle)
 - `Stream<T> stream.parallel();`
 - `Stream<T> stream.sequential();`

Terminal operations

- Terminal operations – give a result
- Parallel or sequential
- Types
 - Iteration: `forEach`, `forEachOrdered`, `iterator`
 - Searching: `findFirst`, `findAny`
 - Check: `anyMatch`, `allMatch`, `noneMatch`
 - Aggregators:
 - Reducers – reduce all stream to one value (e.g. sum)
 - Collectors – put stream data to some collection

Short circuiting

- Some operations can stop working with stream
- Can be useful, when working with infinite streams
- `find*()`, `limit()`, `*Match()`

```
boolean chuckNumberExists = Stream.iterate(1, i -> i + 1).  
    filter(i -> i % 2 == 0).  
    anyMatch(MyMath::isPrime);
```

Peek + allMatch = takeWhile 😊

```
IntStream.iterate(1, n -> n + 1)
    .peek(n -> {
        // doActionOn(n)
        System.out.println(n);
    })
    .allMatch(n -> n < 9);
```

Short circuiting

- Some operations can stop working with stream
- Can be useful, when working with infinite streams
- `find*()`, `limit()`, `*Match()`

```
Optional<Integer> superNumber = Stream.iterate(1, i -> i + 1) .  
    filter(MyMath::isPrime) .  
    filter(i -> i % 2 == 0) .  
    findAny() ;  
System.out.println(superNumber.get()) ;
```

What will happen if there is no Chuck Norris number in infinite stream?

- OutOfMemory – is the best scenario for you

There only 6 Short circuiting operations

- anyMatch
- allMatch
- noneMatch
- findFirst
- findAny
- limit

Iteration – only two methods

- `forEach`

```
IntStream.range(1, 100).forEach(System.out::println);
```

- `iterator` – this can be used for backward compatibility with legacy code

```
Iterator<Integer> iterator = IntStream.range(1, 100).iterator();
```

Riddle

```
int sum=0;  
IntStream.range(1, 100).forEach(i->sum+=i);  
System.out.println(sum);
```

- A. Compile Error
- B. 4950
- C. RuntimeException
- D. 0

Riddle

```
int sum=0;  
IntStream.range(1, 100).forEach(i->sum+=i);  
System.out.println(sum);
```

A. Compile Error

B. 4950

C. RuntimeException

D. 0



Riddle

```
int sum=0;  
IntStream.range(1, 100).forEach(i->sum+=i);  
System.out.println(sum);
```



Effectively final

A. Compile Error

B. 4950

C. RuntimeException

D. 0

I know how to solve it...

```
int[] sum=new int[1];  
IntStream.range(1, 100).forEach(i->sum[0]+=i);  
System.out.println(sum[0]);
```

```
private static int getSum(Stream<Integer> stream) {  
    int[] sum=new int[1];  
    stream.forEach(i -> sum[0] += i);  
    return sum[0];  
}
```

```
Stream<Integer> stream = Stream.iterate(1, i-> 1).limit(100);  
int sum = getSum(stream);  
System.out.println(sum);
```

- A. 100
- B. OutOfMemory
- C. RuntimeException
- D. Compile error

```
private static int getSum(Stream<Integer> stream) {  
    int[] sum=new int[1];  
    stream.forEach(i -> sum[0] += i);  
    return sum[0];  
}
```

```
Stream<Integer> stream = Stream.iterate(1, i-> 1).limit(100);  
int sum = getSum(stream);  
System.out.println(sum);
```

A. 100

B. OutOfMemory

C. RuntimeException

D. Compile error


```
private static int getSum(Stream<Integer> stream) {  
    int[] sum=new int[1];  
    stream.forEach(i -> sum[0] += i);  
    return sum[0];  
}
```

```
Stream<Integer> stream = Stream.iterate(1, i-> 1).limit(100).parallel();  
int sum = getSum(stream);  
System.out.println(sum);
```

- A. 100
- B. 72
- C. 95
- D. 92
- E. Runtime Exception

```
private static int getSum(Stream<Integer> stream) {  
    int[] sum=new int[1];  
    stream.forEach(i -> sum[0] += i);  
    return sum[0];  
}
```

```
Stream<Integer> stream = Stream.iterate(1, i-> 1).limit(100).parallel();  
int sum = getSum(stream);  
System.out.println(sum);
```

A. 100

B. 72

C. 95

D. 92

E. Runtime Exception



You need to synchronize

```
private static int getSum(Stream<Integer> stream) {  
    int[] sum=new int[1];  
    stream.forEach(i -> {  
        synchronized (MyMath.class) {  
            sum[0] += i;  
        }  
    });  
    return sum[0];  
}
```

Atomic Integer???



Or undo parallelism 😊

```
private static int getSum(Stream<Integer> stream) {  
    int[] sum=new int[1];  
    stream.sequential().forEach(i -> sum[0] += i);  
    return sum[0];  
}
```

All this are wrong solutions

- forEach sucks...



Reduce function

```
T reduce(T identity, BinaryOperator<T> accumulator);
```

```
public static int getSum(Stream<Integer> stream) {  
    return stream.reduce(0, (x, y) -> x + y);  
}
```



This is not default value.

Try parallel stream and enjoy

BinaryOperator

- Sub Interface of BiFunction

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply(T t, U u);

    default <V> BiFunction<T, U, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t, U u) -> after.apply(apply(t, u));
    }
}
```

Reduce function

```
T reduce(T identity, BinaryOperator<T> accumulator);
```

```
public static int getSum(Stream<Integer> stream) {  
    return stream.reduce(0, (x, y) -> x + y);  
}
```



This is not default value.

Try parallel stream and enjoy

Reduce with optional

```
Optional<T> reduce(BinaryOperator<T> accumulator);
```

```
public static int getSum(Stream<Integer> stream) {  
    Optional<Integer> optional = stream.reduce((x, y) -> x + y);  
    return optional.get();  
}
```

optional.isPresent()

What else should you know about reduce?

**It is not constrained to
execute sequentially.**

Lab

- Write method which will receive list of employees and will return string with their names separated by comma:
- Input:

```
ArrayList<Employee> employees = new ArrayList<>();  
employees.add(new Employee("Hirsh"));  
employees.add(new Employee("Avishay"));  
employees.add(new Employee("Hadas"));
```
- Output: String which contains text: "Hirsh,Avishay,Hadas"

```
Optional<Integer> optional = stream.reduce((x, y) -> x + y);
```

Lab

- Write method which will receive list of employees and will return List of their names (only the uppercased ones) sorted by length.

Riddle

```
ArrayList<Integer> nums = new ArrayList<>();  
IntStream.range(1, 10000).forEach(nums::add);  
System.out.println(nums.size());
```

(1) ArrayIndexOutOfBoundsException

(3) 9999

(2) 10000

(4) Compilation Error

Riddle

```
ArrayList<Integer> nums = new ArrayList<>();  
IntStream.range(1, 10000).parallel().forEach(nums::add);  
System.out.println(nums.size());
```

(1) ☒ ArrayIndexOutOfBoundsException

(3) 9999

(2) Any number less than 10000

(4) Compilation Error

Collectors

- Mutable reduction operation that accumulates input elements into a mutable result container
- optionally transforming the accumulated result into a final representation after all input elements have been processed
- can be performed either sequentially or in parallel.

How to collect objects from stream

- `List<Employee> list = stream.collect(Collectors.toList());`
- `Object[] objects = stream.toArray();`
- `Employee[] employees = stream.toArray(Employee[]::new);`



















Labs

Fix it with Collectors

```
ArrayList<Integer> nums = new ArrayList<>();  
IntStream.range(1, 10000).parallel().forEach(nums::add);  
System.out.println(nums.size());
```

- Write method which will receive list of employees and will return List of their names sorted by salary of employee, starting from most expensive employee

Collectors

```
  toConcurrentMap Collector<T, ...  
  toList() Collector<T, ?, List<  
  toSet Collector<T, ?, Set<T>>  
  toCollection Collector<T, ?, ...  
  toConcurrentMap Collector<T, ...  
  toConcurrentMap Collector<T, ...  
  toMap Collector<T, ?, Map<K, ...  
  toMap Collector<T, ?, Map<K, ...  
  toMap (Fun... Collector<T, ?, M>
```

Collectors examples

```
Map<String, Integer> concurrentMap = employees.stream().  
    collect(Collectors.toMap(Employee::getName, Employee::getSalary));
```

```
ConcurrentMap<String, Integer> concurrentMap = employees.stream().  
    collect(Collectors.toConcurrentMap(Employee::getName, Employee::getSalary));
```

Collectors methods

- `partitioningBy(Predicate<T>)` - returns `Map<Boolean, List<T>>`
- `groupingBy(Function<T,K>)` – returns `Map<K, List<T>>`
- `toMap(Function<T,K>, Function<T,U>)` – returns `Map<K,U>`

Lab

- Write method which will return list of employees and will calculate sum of man salaries against sum of woman salaries
- `public boolean isManEmployeesMoreExpensive(List<Employee>)`

Lab

- Write method which will receive List of Employees and will return `map<CompanyName, List<Employee>>`
- Yes each employee has property: `String companyName`
- Write method which receive map from previous method and returns `Map<CompanyName,NumberOfWorkers>`
- Test it: Print name of each company against number of workers

Collectors more...

```
maxBy (Comparator<? super T> comparator)
```

```
minBy (Comparator<? super T> comparator)
```

- counting
- summarizingInt
- summarizingDouble
- averagingDouble

Lab

- There are three categories of employees
- Juniors – salary <14000
- Middle – salary < 21000
- Seniors - >21000
- Create appropriate enum
- Write method which will receive list of companies and will categorize it by setting companyProfile property according to the most employee's seniorities

iterable.forEach() vs stream.forEach()

- Iterable forEach() – ordered not parallel (like old for)
- Stream for each – you can decide
- Stream forEachOrdered (relevant only if stream was ordered)

Performance



How to do benchmark?



Students think that benchmark is:



Junior Software Engineer

```
public static void main(String[] args) throws Exception {  
    ApplicationContext context = new AnnotationConfigApplicationContext("com");  
    long before = System.currentTimeMillis();  
    Dao dao = context.getBean(Dao.class);  
    long after = System.currentTimeMillis();  
    System.out.println(after-before);  
}
```

Middle Software Engineer

```
public static void main(String[] args) throws Exception {  
    ApplicationContext context = new AnnotationConfigApplicationContext("com");  
    long before = System.nanoTime();  
    for (int i=0;i<1000000;i++) {  
        Dao dao = context.getBean(Dao.class);  
    }  
    long after = System.nanoTime();  
    System.out.println((after-before)/1000000);  
}
```


Senior Software Engineer

```
public static void main(String[] args) throws Exception {  
    ApplicationContext context = new AnnotationConfigApplicationContext("com");  
    Dao dao=null;  
    long before = System.nanoTime();  
    for (int i=0;i<1000000;i++) {  
        dao = context.getBean(Dao.class);  
    }  
    long after = System.nanoTime();  
    System.out.println((after-before)/1000000);  
    System.out.println(dao);  
}
```

The architect

You drink,
I'll benchmark



JMH

- <http://openjdk.java.net/projects/code-tools/jmh/>

Simple benchmark of void code or method

```
@BenchmarkMode (Mode.AverageTime)  
@OutputTimeUnit (TimeUnit.MILLISECONDS)  
@Benchmark  
public void test1() throws Exception {  
  
    // code you want to test  
  
}
```

Simple benchmark of which return something

```
@BenchmarkMode (Mode.AverageTime)  
@OutputTimeUnit (TimeUnit.MILLISECONDS)  
@Benchmark  
public List<Integer> testGetNums () throws Exception {  
  
    return getNums ();  
}
```

State

```
@State (Scope.Benchmark)
public static class BenchmarkState {
    List<Integer> primes;

    public BenchmarkState() {
        primes = getNums();
    }
}
```

```
@BenchmarkMode (Mode.AverageTime)
@OutputTimeUnit (TimeUnit.MILLISECONDS)
@Benchmark
public Optional<Integer> testGetPrimeNumbersWithIterate (BenchmarkState state) throws Exception {
    return state.primes.stream().parallel().reduce((integer, integer2) -> {
        return integer + integer2;
    });
}
```

Running benchmark

```
public static void main(String[] args) throws RunnerException {  
    Options opt = new OptionsBuilder()  
        .include(".*" + YourBenchmarkClassName.class.getSimpleName() + ".*")  
        .forks(1).warmupIterations(15)  
        .build();  
  
    new Runner(opt).run();  
}
```

Same operation on 1 000 000 elements

Code	Style	Operations per time
for(Element e: ArrayList){...}	Old style	13
list.stream().filter(..).collect(toList)	Sequential & Ordered	10
list.stream().unordered()	Sequential & Unordered	10
list.parallelStream()...	Parallel & Ordered	20
list.parallelStream().unordered()..	Parallel & Unordered	26



So when should I use
parallel streams?

Parallel useful when

- > 100 000 nanos for all work
- Ordered collections (like ArrayList) more influenced from parallelism
- Sure, number of cores influence

How can I generate stream?

- `Stream.generate?`
- `Stream.iterate?`

Lab

- Write method which will return a Stream of primary numbers
- Develop service which will return a list of prime numbers
- `public List<Integer> getPrimeNumbers(int amount)`

Lets benchmark

Old school: 467 millis

	Stream.generate	Stream.iterate
Sequential	493	487
Parallel & Ordered	61	677
Parallel & Unordered	43	648

Generate vs Iterate

Generate	Iterate
Paralleled better	Don't need state (can calculate next element, from previous)
Provokes state – can't be used in parallel	

Splitterator

```
long estimateSize();
```

```
boolean tryAdvance(Consumer<? super T> action);
```

```
Splitterator<T> trySplit();
```

```
int characteristics();
```

Spliterators

```
Splitter<Integer> splitter =  
    Spliterators.splitter(iterator,  
        Integer.MAX_VALUE,  
        Splitter.NONNULL |  
        Splitter.IMMUTABLE |  
        Splitter.ORDERED );  
  
return StreamSupport.stream(splitter, true);
```


Lab

- You have Employee class
- You have product class (name, price)
- You have sale class (employee, product)
- Write method which will receive list of sales and will return best employee (max total income to company)
- What if there are more than one best employee?

Lab

- Write method which will receive a text and will return the most popular words

New methods of existing classes

- **Default methods in interfaces**
- **Static methods in interfaces**
- Wrapper classes
- String
- Iterable / Collection / List
- Map

Integer

```
static int sum(int a, int b)
```

```
static int max/min(int a, int b)
```

```
static int compareUnsigned(int x, int y)
```

Additional unsigned methods (toUnsignedString, toUnsignedLong...)

String

```
static String join(CharSequence delimiter, CharSequence... elements)
```

Iterable / Collections / List

- forEach
- spliterator

default boolean removeIf(Predicate<? **super** E> filter)

- stream
- parallelStream

default void replaceAll(UnaryOperator<E> operator)

- sort

Map

static Comparator comparingByKey() / comparingByValue()

default **V** getOrDefault(Object key, **V** defaultValue)

default void forEach(BiConsumer<? **super** **K**, ? **super** **V**> action)

default void replaceAll(BiFunction function)

default **V** putIfAbsent(**K** key, **V** value)

default boolean remove(Object key, Object value)

Map – more new methods

default boolean replace(**K** key, **V** oldValue, **V** newValue)

default V replace(**K** key, **V** value)

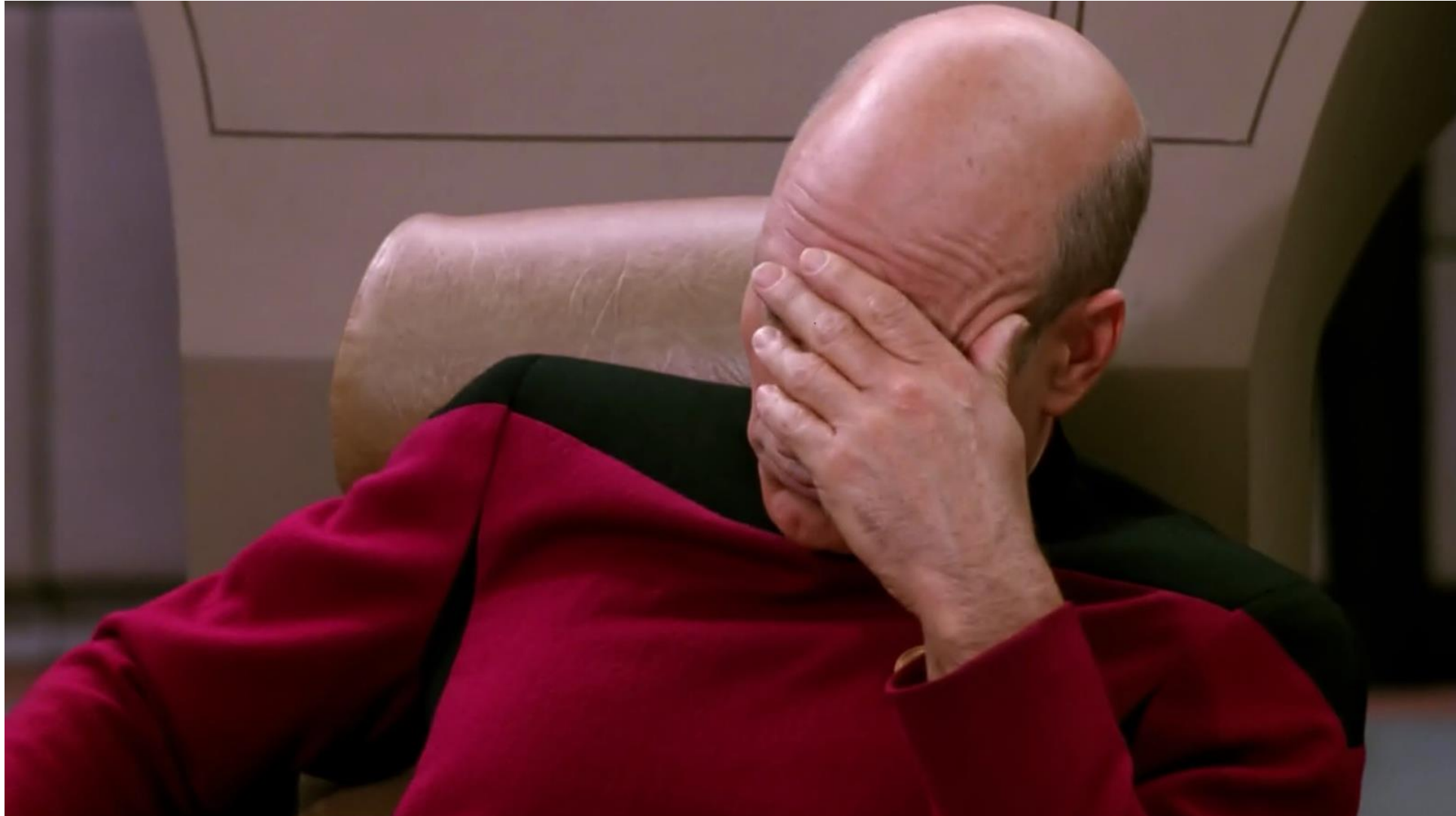
default V compute(**K** key, BiFunction remappingFunction)

default V merge(**K** key, **V** value, BiFunction remappingFunction)

JSR 310

NEW DATA TIME API

What's wrong again with old api?



What's wrong

- The only actual get method from calendar class was:
public Date getTime()
- public deprecated **int** getYear()
- public deprecated int getMonth()
- public deprecated int getDay ()
- public deprecated int getHours()
- public deprecated int getMinutes()
- public deprecated int getSeconds()

The only NOT deprecated method in class Date is:

- **public long getTime()**
- **Returns the number of milliseconds 😊**
- **since January 1, 1970, 00:00:00 ☹️**

JDK Date

`java.util.Date`

- Wasn't updated (from JDK1.0)
- Uses two digit years (from 1900)
- January is 0, December is 11
- Not immutable.
- Not Threadsafe.
- Most methods deprecated in JDK1.1
- Uses milliseconds from 1970 representation

JSR 310

- The new Java Date-Time API (JSR 310) tries to address these issues.
- It is based on ISO 8601 standard, and uses the Gregorian calendar as the default one.
- The `java.time` package is the main API for date and time in Java 8.

JSR 310

- We have different classes for different purposes:
- Specialized classes for date.
- Specialized classes for time.
- Specialized classes for date and time.
- Specialized classes for date, time and time zone.
- We have classes that represent human readable time and classes that represent machine time.
- As opposed to Date class method calls can be chained together.
- As opposed to Calendar class all classes are thread safe.

Immutable

- `LocalDateTime now = LocalDateTime.now();`
- `LocalDateTime minusSecond = now.minusSeconds(1);`

Most important classes

- LocalDate
- LocalTime
- LocalDateTime
- Duration
- Period
- ChronoUnit

Repeatable annotations

How it works?

```
@Component
public class Alarm {
    @Scheduled(cron = "0/3 * * * * ?")
    @Scheduled(cron = "0/10 * * * * ?")
    @Scheduled(cron = "0 0 12 * * ?")
    public void wakeUp() {
        LocalDateTime dateTime = LocalDateTime.now();
        System.out.println("Pik Pik! " + dateTime.getSecond());
    }
}
```

Repeatable Annotations

- Java 8 introduces new meta annotation: `@Repeatable`
- Any `@Repeatable` annotation can be placed multiple times on the same element



Wait a second, what
`getAnnotation()` will
return?

NULL

Repeatable Annotations

- Each repeatable annotation points to another annotation, which holds the list of the former annotations
- In runtime, repeatable annotation will create a handler, which will hold the list of those annotations



How it works?

```
@Component
public class Alarm {
    @Scheduled(cron = "0/3 * * * * ?")
    @Scheduled(cron = "0/10 * * * * ?")
    @Scheduled(cron = "0 0 12 * * ?")
    public void wakeUp() {
        LocalDateTime dateTime = LocalDateTime.now();
        System.out.println("Pik Pik! " + dateTime.getSecond());
    }
}

@Repeatable(Schedules.class)
public @interface Scheduled {

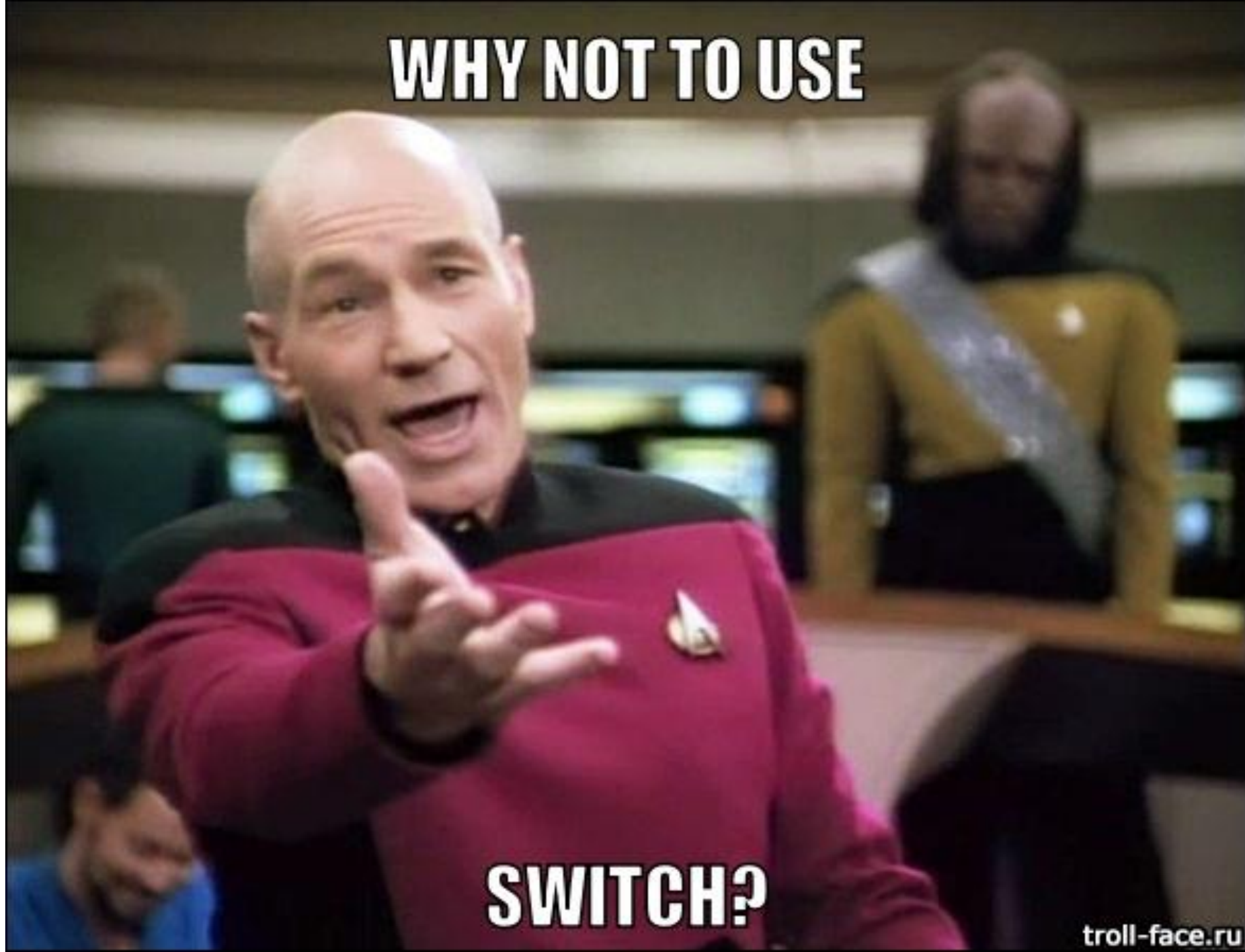
    public @interface Schedules {
        Scheduled[] value();
    }
}
```


Write your own framework

- Your application need to take some int parameter and run appropriative business logic.
- Lets start from 3 different options for this int.
- The code you will write now will become internal infrastructure.
- All your descendants will continue doing the same



WHY NOT TO USE



SWITCH?

Switch pattern 😊

```
public void doWork() {  
    int code = DBUtils.getWorkCode();  
  
    switch (code) {  
        case 1:  
            // welcome handling (25 lines of code)  
            break;  
        case 2:  
        case 3:  
            // bye bye handling (40 lines of code)  
    }  
}
```

We love you, Switch ...

```
public DistribHandler resolve(Integer valueOfAccess) {
    switch (Integer.valueOfAccess) {
        case PDF_STORAGE:
            fileContainer = new PdfRecordFileContainer();
            getPdfFromStorage(fileContainer, documentObject.getDocument());
            break;
        case PDF_SRC:
            fileContainer = new PdfRecordFileContainer();
            fillObjectsForPdf(fileContainer, documentObject.getDocument());
            break;
        case PDF_WS:
            fileContainer = new WsPdfRecordFileContainer();
            getPdfFromPdfWs(fileContainer, documentObject.getDocument());
            break;
        case LIS:
            fileContainer = new PdfRecordFileContainer();
            getPdfFromLisDocument(j, fileContainer);
            break;
        case IMAGE:
            fileContainer = new PdfRecordFileContainer();
            getPdfFromImageDocument(j, fileContainer);
            break;
        case FORM:
            fileContainer = new PdfRecordFileContainer();
            getPdfFromFormDocument(fileContainer, documentObject.getDocument());
            break;
    }
}

switch (value.getNumericValue()) {
    case 1:
        text = MessageFormat.format("{0} הצעת פוליטת", emailRequest.getSubject());
        break;
    case 2:
        text = MessageFormat.format("{0} רבישת פוליטת", emailRequest.getSubject());
        break;
    case 3:
        text = MessageFormat.format("{0} חידוש פוליטת", emailRequest.getSubject());
        break;
    case 4:
        text = MessageFormat.format("{0} שינויים בפוליטת", emailRequest.getSubject());
        break;
    case 5:
        text = MessageFormat.format("{0} מכתב מ", brandHebName);
        break;
    case 6:
        text = MessageFormat.format("{0} מכתב חשוב עבור מ", brandHebName);
        break;
    case 7:
        text = MessageFormat.format("{0} רבישת ביטוח נסיעות", brandHebName);
        break;
    case 9:
        text = MessageFormat.format("{0} - {1}", emailRequest.getSubject(), brandHebName);
        break;
    case 10:
        text = MessageFormat.format("{0} - {1}", emailRequest.getSubject(), brandHebName);
        break;
    case 11:
        text = emailRequest.getSubject();
        break;
    case 12:
        if (!resources.getBrandKey().isBituhYashir()) {
            text = format("{0} - תודה מ", brandHebName);
        } else {
            text = format("{0} - תודה מ", brandHebName);
        }
        break;
    case 14:
        text = MessageFormat.format("{0} - השקט הנפשו שלך - ביטוח האוננות אישיות", brandHebName);
        break;
    case 15:
        text = MessageFormat.format("{0} בקשה ליצירת קשר לחידוש ביטוח", emailRequest.getSubject());
        break;
    case 16:
```

```
icyDatFileConsumer.class);
if (item.getAddressCode().length() >= 1
    de()) ? item.getAddressCode().trim() : "0";
dresser(address);
currentPath, basket, dataConsumer);
```

```
icyDetailsConfConsumer.class);
currentPath, basket, dataConsumer);
```

```
icyLetterConsumer.class);
lr(basket.getMetaDBasket().getPrtNr());
currentPath, basket, dataConsumer);
```

```
icyCompulsoryConsumer.class);
setSeqNr(item.getCompSeqNr());
currentPath, basket, dataConsumer);
```

;

Solve it with repeatable annotations

```
@TemplateCode(2)
@TemplateCode(3)
public class GoodByeHandler implements Handler {
    @Override
    public void handle() {
        System.out.println("By bye");
    }
}
```