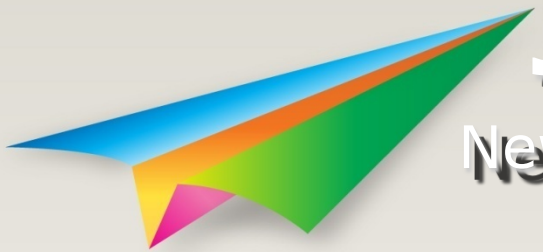


Java 8

New language features and Libraries

trainologic

What's new in Java 8

- Lambda Expressions.
- java.util.function.
- Default Methods.
- Filter/map/reduce for Java.
- Date & Time API.
- Concurrency Additions.
- Reflection And Annotations.

trainologic

Background

- Java is an Object Oriented language.
- Functions are not first class citizens of Java.
- You can't assign a function to a variable, use it as a method argument or return it from a method call.
- We usually use callback interface and anonymous inner classes to pass functionality to methods.
- The "Hello World" of anonymous inner classes:

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

Closures

- Can we write shorter, less verbose code?
- It Scala, Groovy and other programming languages we can. Using **closures**.
- A closure is just a piece of functional code that we can move around, assigned to a variable, return it from a method call, etc.
- A Groovy closure example:

```
["Java", "Scala", "Groovy"].collect {
    it + " is a Programming Language"
}
```

Lambda Expressions

- Lambda expressions, introduced in Java 8, are the Java alternative to closures.
- It let's us treat functions as first class citizens in Java.
- A Lambda expression is a functional expression without a formal declaration: name, modifiers, return value.
- You can write it where you use it, especially when you use it only once.
- A Lambda expression must be bound to a Functional Interface.
- We will cover functional interfaces shortly.

Lambda Expressions Syntax

- A Lambda expressions is composed of three parts:
  - Arguments list, comma separated.
  - Arrow Token (**->**).
  - The expression Body.
- It can have zero or more arguments.
- Argument types can be explicit or inferred.
- Arguments are enclosed in parentheses, unless there is a single argument whose type can be inferred.

Lambda Expressions Syntax

- The expression body can be a single statement or an expression block.
- For a single statement curly brackets are optional. The body of the expression is evaluated an returned.
- An expression block is evaluated just like any other Java statement block.
- The Mouse Listener revisited:

```
button.addActionListener( e -> System.out.println(e) );

button.addActionListener( e -> { System.out.println(e); } );

button.addActionListener
    ((ActionEvent e) -> System.out.println(e));
```

Functional Interfaces

- The *ActionListener* interface contains a single abstract method.
- In Java 8 such kind of an interface is known as a **Functional Interface**.
- Such interfaces are leveraged to use Lambdas in Java 8.
- Nothing special is needed to identify an interface as functional. The complier identify it as such.
- API authors may mark a functional interface using the @*FunctionalInterface* annotation, and the compiler will validate the interface indeed contains a single method.
- This enables existing libraries to be used with Lambdas.

Functional Interface Examples

- A *Comparator* example:

```
list.sort((a, b) -> 0);

list.sort(new Comparator<String>() {
    @Override
    public int compare(String o1, String o2)  { return 0; }
});
```

- A *Runnable* example:

```
Runnable r = () -> System.out.println( "I'm Lambda" );

new Thread(
    // The Runnable interface is deducted by the compiler,
    // when we don't specify the functional interface.
    () -> System.out.println( "I'm Lambda" )
).start();
```

Example

- A Lambda expression can be passed as a method argument:

```
// FileFilter is a functional interface.
// It's interface method is accept(File)
FileFilter textFilesFilter = f -> f.getName().endsWith(".txt");
File folder = new File(".");
File[] textFiles = folder.listFiles(textFilesFilter);
```

Target Typing

- The functional interface is not part of the Lambda expression syntax.
- It's type is inferred from the surrounding context.

```
// Different types in different contexts.
Callable<String> c = () -> "called" ;
Supplier<String> s = () -> "supplied" ;
```

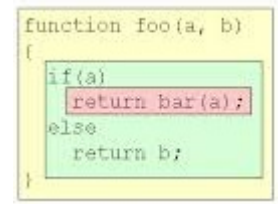
- The compiler is responsible to infer it's type. It uses the type expected in the context in which it appears.
- This type is called the **Target Type**.
- A lambda expression can only appear in a context whose target type is a functional interface.

Variables Capture

- Captured variables are local variables from an enclosing context referenced within a Lambda expression.
- The compiler checks such references to local variables.
- In Lambda expressions such a referenced variable can be *effectively final*: it's value is not mutated.
- This is also true for inner classes in Java 8, you don't have to explicitly declare such variables as **final**.
- Lambda expressions that do not capture members from the closing context, do not hold a reference to it. It has a beneficial impact on memory management.
- As opposed to inner class that always reference **this**.

Lexical Scoping

- Lambda expressions are lexically scoped.
- This means that variables and methods names in body are interpreted just as they are in the enclosing scope.
- As a consequence, the **this** keyword and it's references has the same meaning as outside of the Lambda expression.
- The parameters of a Lambda expression must not shadow those of the enclosing scope.

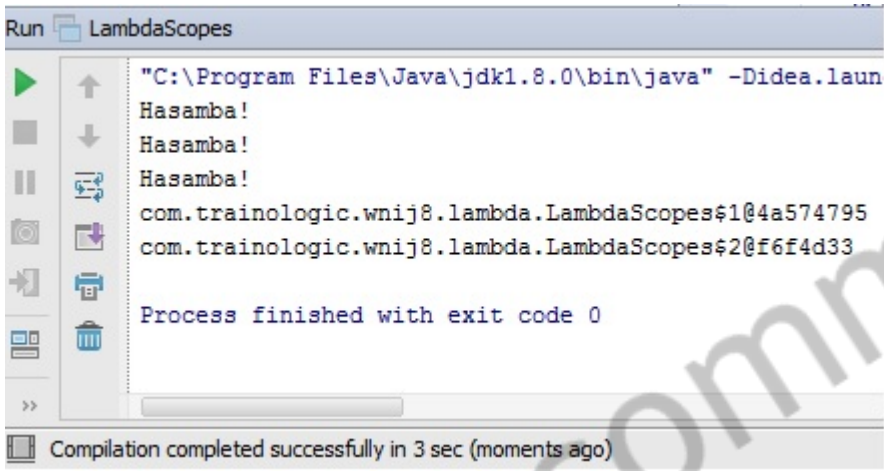


Example

```
public class LambdaScopes {
    public String toString() { return "Hasamba!" ; }

    Runnable r1 = () -> {System.out.println(this);};
    Runnable r2 = () -> {System.out.println(toString());};
    Runnable r3 = () -> {System.out.println(this.toString());};

    Runnable r4 = new Runnable() {
        public void run() {System.out.println(this);}
    };
    Runnable r5 = new Runnable() {
        public void run() {System.out.println(toString());}
    };

    public static void main(String[] args) {
        new LambdaScopes().r1.run();
        new LambdaScopes().r2.run();
        new LambdaScopes().r3.run();
        new LambdaScopes().r4.run();
        new LambdaScopes().r5.run();
    }
}
```

Example Output

- This is the output of the example:



- We can clearly see the difference between the anonymous inner classes and the Lambda expressions.

Method References

- Lambda expressions allow us to define an anonymous method and treat it as instance of a functional interface.
- Often, a Lambda expression does nothing but call an existing method.
- Method references are similar to Lambda expressions, but instead of providing an expression body, they refer to an existing method name.
- A method reference is denoted with a double colon delimiter.
- E.g.: `::`.
- An example will help to clarify this:

Method Reference Example

- The expression String::toLowerCase is a shorthand for a Lambda expression, which invokes the named method on a given argument.
- In this case, the type precedes the delimiter (::), the method follows it, and the receiver is the first parameter of the functional interface method.
- In short, The method reference will invoke the toLowerCase() method of a *String* type.

```
List<String> list = …

list.forEach(s -> s.toLowerCase());

list.forEach(String::toLowerCase);
```

Method Reference Types

- There are basically four method reference types:
  1. A static method.
  2. An instance method of a particular object.
  3. An instance method of an arbitrary object of a particular type.
  4. A class constructor reference.
- The example from the previous slide is of the third type. This is the one you will probably most often use.

Static Method Reference

- An example is given bellow.
- Don't worry if you don't know what *Consumer* is. We'll get to it soon.

```
// All consumers do the same.
Consumer<Integer> c1 = new Consumer<Integer>() {
    @Override
    public void accept(Integer integer) { System.exit(integer); }
};
Consumer<Integer> c2 = (integer) -> System.exit(integer);
Consumer<Integer> c3 = System::exit;

c2.accept(0);
c3.accept(1);
```

Instance Method Reference

- An example is given bellow.
- The lambda expression will capture the *String* object and will invoke contains(), using it as an argument.

```
// All predicates return the same results.
Collection<String> names = …
Predicate<String> p1 = new Predicate<String>() {
    @Override
    public boolean test(String s) {
        return names.contains(s);
    }
};
Predicate<String> p2 = (s) -> names.contains(s);
Predicate<String> p3 = names::contains;

p2.test("John");
p3.test("Jane");
```

Conclusion

- Lambda expressions are a valuable feature in Java 8.
- They eliminate the bulky syntax present in anonymous inner classes.
- They eliminate confusion regarding names by means of lexical scoping.
- They can capture effectively final variables.
- Lambda expressions open the door for some other new exciting features in Java 8, as we shall see in the next chapters.

What's new in Java 8

- Lambda Expressions.
- java.util.function.
- Default Methods.
- Filter/map/reduce for Java.
- Date & Time API.
- Concurrency Additions.
- Reflection And Annotations.

Introduction

- java.util.function is a new package in Java 8.
- It contains a bunch of functional interfaces.
- Hence, heavily used in Lambda expressions.
- Many JDK classes are now using these interfaces. We will cover some additions to the Collection API.
- We will cover the basic shapes.
- Some shapes also have different than natural arity. For example *Function* and *BiFunction*.
- Many of the interfaces also have primitive version, for example *IntFunction*.

Predicate

- Represents a predicate, a boolean evaluated function.
- It's functional method is test(T).
- Example:

```
class Person {
    String name;
    String password;
    String country;
    Integer age = 0;
    public void greet() {System.out.println ("Hello " + name)}
}

Predicate<Person> isVetern = new Predicate<Person>() {
    public boolean test(Person person) {
        return person.age > 65;
    }
};
```

Function

- Represents a function that accepts an argument of type T, and produces a result of type R.
- It's functional method is apply(T, R).
- Example:

```
Function<String, Person> toPerson =
    s -> new Person(s, (int)(random() * 100));

public Collection<Person> applyNames(Collection<String> names) {
    List<Person> persons = new ArrayList<>(names.size());
    names.forEach(s -> { persons.add(toPerson.apply(s)); });
    return persons;
}
```

Supplier

- Supplies a typed result.
- It's functional method is get(T).
- Example:

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

public void randomPassword(Person person) {
    person.password = passwordSupplier.get();
}
```

Consumer

- Consumes a typed argument and produces no result.
- It's functional method is accept(T).
- Example (in four flavors):

```
public void sayHi(Collection<Person> coll)  {
    for (Person p : coll) { p.greet(); }

    coll.forEach(new Consumer<Person>() {
        public void accept(Person person) {
            person.greet();
        }
    });

    coll.forEach(person -> person.greet());
    coll.forEach(Person::greet);
}
```

forEach

- We have seen few usages of the forEach() method.
- It is a new method introduced in the *Iterable* interface in Java 8.
- As you can see, it accepts a Consumer argument and call it's accept() method for every element.
- forEach (internal iteration) can be considered as a replacement of the enhanced for loop (external iteration).
- Although it's code is longer, the iteration logic and the execution logic are separated.
- Implementations can introduce a parallel forEach().

Other Interfaces

- *BiPredicate* – A two arity specialization of *Predicate*.
- *BiConsumer* – A two arity specialization of *Consumer*.
- *BiFunction* – A two arity specialization of *Function*.
- *UnaryOperator* – same as *Function*<T,T>.
- *BinaryOperator* – same as *BiFunction*<T,T,T>.

UnaryOperator Example

- The *List* interface contains a new replaceAll() method.
- It accepts an UnaryOperator as a parameter, and replaces each element with the result of applying the operator on the element.
- Example (in three flavors):

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

list.replaceAll(new UnaryOperator<String>() {
    public String apply(String s) { return s.toUpperCase(); }
});

list.replaceAll(s -> s.toUpperCase());

list.replaceAll(String::toUpperCase);
```

What's new in Java 8

- Lambda Expressions.
- Java.util.Function.
- Default Methods.
- Filter/map/reduce for Java.
- Date & Time API.
- Concurrency Additions.
- Reflection And Annotations.

Interface Methods – No Fun

- Adding functionality to an existing interface is not easy.
- Unless you can, and will update all it's implementations.
- Adding a new abstract method to an interface may cause implementations to break.
- Lambda expressions open up new possibilities, which we can use when re-designing our interfaces.

Interface Methods – No Fun

- For example, we would like to add a method **removeIf**(*Predicate*) to the *Collection* interface.
- But there are dozen of implementing classes just in the java.util package.
- Implementing it - even in just a few of these classes - is not an easy task.
- Well, in Java 8 it's easy. Using a **Default Method**.

Default Methods

- The purpose of default methods is to let interfaces evolve after publication, without breaking the API.
- They provide a concrete behavior to an interface.
- In Java 8, interface methods can be abstract or default - using the **default** keyword.
- Default methods have an implementation that is inherited by classes, unless overridden.
- When an interface extends another, it can add, re-implement or re-abstract a default method.

removeIf

- In fact, we do have a new default removeIf() method in the *Collection* interface:

```
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean removed = false;
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
        if (filter.test(each.next())) {
            each.remove();
            removed = true;
        }
    }
    return removed;
}
```

Functional Interfaces

- Default methods don't count in the - one abstract method per a functional interface - restriction.
- For example, we have a default negate() method in the *Predicate* interface.

```
default Predicate<T> negate() {
    return (t) -> !test(t);
}
```

Pseudo Multiple Inheritance

- In the unlikely case that you implement two interfaces with the same default method, you will need to override the method and choose one implementation.

```
public class MultiExample implements Foo, Bar {
    @Override
    public void done() { Foo.super.done(); }
}

interface Foo {
    default void done() { System.out.println("I'm fully done");}
}

interface Bar {
    default void done() { System.out.println("I'm barely done");}
}
```

Interface Static Methods

- In addition to default methods, in Java 8 we can code static methods in interfaces as well.
- This enables helper methods specific to an interface, to live in the interface rather then in a helper class.

What's new in Java 8

- Lambda Expressions.
- Java.util.Function.
- Default Methods.
- Filter/map/reduce for Java.
- Date & Time API.
- Concurrency Additions.
- Reflection And Annotations.

java.util.stream

- The java.util.stream package is new in Java 8.
- It provides classes that support functional operations on streams of elements.
- Such operations can be filters, transformations and map-reduce operations on a collection elements.
- The primary package member that we are going to work with is the *Stream* interface.

Streams

- The *Stream* interface contains a sequence of aggregate operations performed on elements.
- These element are consumed from a data source. A data source can be a collection object for example.
- Some stream operations are intermediate (distinct, filter, map) and some are terminal (reduce).
- The result of an intermediate operation is a new stream.
- Terminal operations are the last operations on a stream. They produce a result or a side-effect.
- A stream can have a single terminal operation.

Streams

- Streams are combined to perform a pipeline. A pipeline consists of a data source, zero or more intermediate operations and a terminal operation.
- Streams can execute in serial or parallel. This is a property of the stream.
- By default, the JDK implementations always return a serial (sequential) stream.
- A stream is disposable after it is used (consumed), and can't be reused. It is a sub-interface of *AutoCloseable*.

Streams Vs. Collections

- Streams differ from Collections in the following ways:
  - No storage – Streams just convey elements from it's source through a pipeline of operations.
  - Functional in nature – An operation on a stream provides a result but doesn't mutate it's source.
  - Laziness – Intermediate operations are lazy.
  - Possibly unbounded – Streams don't need to have a finite size.
  - Consumable – A new stream must be reopened each time you need to manipulate the elements of the same data source.

Streams in Collections

- The *Collection* interface in Java 8 contains a stream() method which returns a sequential *Stream* over it's elements.
- You can also return a new collection from the stream by calling it's collect() method.
- Note that collections are not the only source of streams. Other sources may be things like IO channels and *Stream*'s own static generate() method.
- The elements of the collection itself (as all stream source elements) are not being mutated by the stream.

Specialized Streams

- In addition to the *Stream* interface, we also have streams for primitive types:
  - *IntStream*.
  - *LongStream*.
  - *DoubleStream*.
- These interfaces introduce more reduce operations such as sum() and average().

Example

```
// The list is the pipeline data source.
Collection<String> names = Arrays.asList
    ("Vera", "Chuck", "Dave", "Dave");

// Create the stream.
Stream<String> stream = names.stream();

// Returns an array containing the elements of this stream.
// This is a terminal operation.
System.out.println(Arrays.toString(stream.toArray()));
// [Vera, Chuck, Dave]

// The filter argument is of type Predicate<String>
stream = names.stream().filter(p -> !p.startsWith("D"));
System.out.println(Arrays.toString(stream.toArray()));
// [Vera, Chuck]
```

Example

```
// Stream operations can be chained.
stream = names.stream().skip(1).limit(2);
System.out.println(Arrays.toString(stream.toArray()));
// [Chuck, Dave]

// mapToInt argument is of type ToIntFunction<String>.
// This is an int-producing specialization for Function.
int total = names.stream().mapToInt(String::hashCode).sum();

// Optional (new in Java8) is the equivalent for Scala Option.
Optional<Integer> reduce =
    names.stream().map(String::hashCode).reduce((x, y) -> x +
y);
assert (total == reduce.get());
```

Example

```
// We first map the String elements to lower case.
// Then we return a new collection with the lower case names.
// We will have a deeper look on collect() soon.
List<String> namesLowerCase = names.stream()
    .map(s -> s.toLowerCase())
    .collect(Collectors.toList());
System.out.println(namesLowerCase);
// [vera, chuck, dave, dave]

// Calculates the average length of the values in the names
list.
// flatMapToInt returns an IntStream after applying the
function.
OptionalDouble avg = names.stream()
    .flatMapToInt(s -> IntStream.of(s.length())).average();
```

Example

```
names.stream()
   // a mapping from String stream to Person stream.
   .flatMap(s -> {
        int age = (int)(random() * 100);
        return Stream.of(new Person(s, age));
    })
    // Sorted according to the provided Comparator (age).
    .sorted( (o1, o2) -> o1.age.compareTo(o2.age) )
    // Performs an action for each element of this stream.
    .forEach( person -> System.out.println(person.toString()) );
Person{age=5, name='Dave'}
Person{age=24, name='Dave'}
Person{age=74, name='Chuck'}
Person{age=94, name='Vera'}
```

Laziness

- Consider this example:

```
Collection<String> names = Arrays.asList
    ("Vera", "Chuck", "Dave", "Dave");
Optional<String> chuck = names.stream()
    .filter(s -> s.startsWith("C"))
    .findFirst();
```

- filter() is lazy.
- findFirst() will only pull from the upstream as long as it can't find a match.
- This means that we only apply the predicate until it evaluates to true.

Laziness

- Processing streams lazily has a significant benefit.
- Pipeline operations can be fused together into a single pass on the pipeline data.
- Plus, the state that we have to keep is minimal.
- Note: all terminal operations are eager, except for iterator() operation.

Parallelism

- Our example operations can execute in parallel:

```
Optional<String> chuck = names.parallelStream()
    .filter(s -> s.startsWith("C"))
    .findFirst();
```

- Except for non deterministic operations such as findAny, either method should produce the same computation.
- We can modify / query a stream serial / parallel mode using the following methods:
  - sequential()
  - parallel()
  - isParallel()

Parallelism

- When executed in parallel, the runtime splits the stream into sub-streams and afterwards combine the results.
- Parallel streams allows you to implement parallelism even with non thread safe collections, as long as the collection is not modified at the same time.
- Parallel operations don't necessarily perform faster.
- This is usually a function of how large the data set is and the number of cores.
- It is your responsibility to decide which way is better by experiment.

Mutable Reduction

- We have seen an example of reduce() operation usage.
- For efficiency, it is better to perform a **mutable reduction** on the elements of a stream.
- A mutable reduction is one in which the reduced value is a mutable result container.
- The elements are incorporated by mutating the state of the result rather than by replacing it.
- We can do it using the *Collector* interface.

The Collector Interface

- *Collector* is a mutable reduction interface that accumulates input elements into a result container.
- The interface signature is *Collector*<T,A,R>.
- T - the type of the input elements.
- A - the mutable accumulation type of the reduction operation.
- R – the result type of the reduction operation.

The Collector Interface

- The interface is specified by four functions that work together to accumulate entries into a result container.
- Optionally perform a final transformation of the result.
- These functions are:
  - supplier (*Supplier<A>*) : initially creates the result container .
  - accumulator (*BiConsumer<A,T>*) : incorporating a new data element into a result container.
  - combiner (*BinaryOperator<A,A>*) : combines two result containers into one.
  - finisher (*Function<A,R>*) : final optional transformation.

Parallel Vs. Sequential

- A sequential implementation will create a single result container using the supplier() method.
- In a sequential implementation, the combine() method will not be called.
- A parallel implementation will partition the input and create a result container for each partition.
- The parallel implementation then will use the combiner function to merge the partition results into a combined result.

Example

- We can perform a mutable reduction operation on a stream using the **collect**(*Collector*) method.
- We have a list of Person, and we want to reduce this list into a map whose keys are countries and values are the sum of people age living in these countries.

```
public static Person builder
          (String name, int age, String country) {
    return new Person(name, age, country);
}

Collection<Person> people = new ArrayList<>();
people.add(Person.builder("Avi", 42, "IL"));
people.add(Person.builder("Boris",97, "RU"));
people.add(Person.builder("John", 21, "US"));
people.add(Person.builder("Jane", 18, "US"));
```

Example

```
Collector<Person, Map<String,Integer>, Map<String,Integer>> collector
= new Collector<Person, Map<String,Integer>, Map<String,Integer>>() {

    // Creates the mutable reduction container.
    public Supplier<Map<String, Integer>> supplier() {
        return ConcurrentHashMap::new;
    }

    // We will discuss map.merge() soon.
    public BiConsumer<Map<String, Integer>, Person> accumulator() {
        return (map, person) -> {
            map.merge(person.country, person.age, (i1,i2) -> i1+i2);
        };
    }
```

Example

```
    // We just return a map here.
     public BinaryOperator<Map<String, Integer>> combiner() {
          return (map1, map2) -> map1;
     }

     // Collector<T,A,R>. In our case T=R
     // We don't need final transformation here.
     public Function<Map<String,Integer>,
                     Map<String,Integer>> finisher() {
          return arg -> arg;
     }

     // Collectors also have a set of characteristics,
     // such as Collector.Characteristics.CONCURRENT.
     // We don't use characteristics here.
     public Set<Characteristics> characteristics() {
          return new HashSet<>();
     }
};
```

Map Merge

- **merge**(*K*, *V*, *BiFunction*) is new in Java 8.
- If the key K is missing or has a null value, the value V is inserted to the map with K as key.
- Otherwise, replace the existing value E with the result of the *Function*.
- The *Function* parameters are V and E.
- In our case we sum two values (ages) and replace the existing key K value.

Discussion

- Note that our *Collector* implementation will break in a parallel implementation.
- This is because we don't really combine anything in the combiner() method, just returning a map.
- However, for a sequential implementation - where we have a single result container - it's ok.
- Finally, we can perform the collect operation:

```
Map<String, Integer> map = people.stream().collect(collector);
```

Discussion

- Our code is lengthy and verbose.
- We need to implement the *Collector* interface though we don't care about transformation nor characteristics.
- We can use an alternative collect() method instead:
- **collect**(*Supplier*, *BiConsumer*, *BiConsumer*).

```
BiConsumer<Map<String, Integer>, Person>
         accumulator = (map, person) -> {
    map.merge(person.country, person.age, (i1, i2) -> i1 + i2);
};

map = people.stream().collect
    (ConcurrentHashMap::new, accumulator, (map1, map2) -> {});
```

Discussion

- Our code is now shorter and less verbose.
- We don't need to take care about transformation or characteristics now.
- Can we make it even better?
- Yes we can!
- With the help of the *Collectors* class.

Collectors

- The *Collectors* class contains factories for many common collectors.
- You can use them as a parameter for the collect() operation.
- Some examples:

```
Collection<String> names =
    Arrays.asList( "Vera", "Chuck", "Dave", "Dave" );

// A Collector that accumulates the input elements into a set.
Set<String> set = names.stream().collect(Collectors.toSet());
System.out.println(set);
// [Vera, Chuck, Dave]
```

Collectors

```
// A Collector that count the number of input elements.
Long americans = people.stream().filter
    (p ->
"US".equals(p.country)).collect(Collectors.counting());

Map<String, Integer> map = people.stream()
    .collect(Collectors.toMap(p -> p.name, p -> p.age));
System.out.println(map);
// {Avi=42, John=21, Boris=97, Jane=18}

// Average age of people.
Double average = people.stream()
    .collect(Collectors.averagingInt(p -> p.age));

String joined = people.stream()
    .map(p -> p.name).collect(Collectors.joining("-"));
System.out.println(joined);
// Avi-Boris-John-Jane
```

Grouping

- Related to map is groupBy().
- It takes a **classification function** and produces a map keyed by that function.
- The value is an element list that correspond to that key.

```
// getCountry is the classification function.
Map<String, List<Person>> citizens = people.stream()
    .collect(Collectors.groupingBy(Person::getCountry));
```

Downstream Collectors

- Collectors can be composed to produce more complex collectors.
- By taking a classifying function and a downstream collector as arguments.
- All elements mapped into the same bucket by the classifying function are passed to the downstream collector.

Downstream Collector Example

- As an example, we will rewrite our custom *Collector* from before, using a classification function and a downstream collector.

```
Collector<Person,?,Integer> downstream =
    Collectors.summingInt(Person::getAge);

Map<String, Integer> collect = people.stream().collect
    (Collectors.groupingBy(Person::getCountry, downstream));
```

- Now it looks much better.

Files Additions

- The following methods were added to the *Files* class in order to support streams:
  - Files.find(Path, int, BiPredicate, FileVisitOption...)
  - Files.lines(Path)
  - Files.lines(Path, Charset)
  - Files.list(Path)
  - Files.walk(Path, FileVisitOption...)
  - Files.walk(Path, int, FileVisitOption...)

Conclusion

- With our final example, we have managed to produce something such as:

```
SELECT SUM (AGE)
FROM PEOPLE
GROUP BY COUNTRY
```

- Looks familiar.
- With streams and collectors, working with collections is much easier then ever.
- We have applied an aggregation framework on top of our collections.

What's new in Java 8

- Lambda Expressions.
- Java.util.Function.
- Default Methods.
- Filter/map/reduce for Java.
- Date & Time API.
- Concurrency Additions.
- Reflection And Annotations.

Overview

- There are many complexities and factors that affect understanding of time: time zones, daylight saving, leap years, etc.
- 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.

Benefits

- 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.

Immutability

- Consider this:

```
java.util.Date date = new Date();
public Date getDate() { return date; }
public void setDate(Date date) { this.date = date; }
```

- The *Date* object in mutable.
- In getters and setters we usually check for nullity, and make a defensive copy of the object by cloning it.
- Not anymore.
- In java.time all package classes are immutable.
- There are no set methods.

Immutability

- Example:

```
Date date = new Date();
LocalDateTime time = LocalDateTime.now();
System.out.println(date.toString());
System.out.println(time.toString());

date.setSeconds(date.getSeconds() - 1);
LocalDateTime minusSecond = time.minusSeconds(1);
System.out.println(date.toString());
System.out.println(time.toString());
```

- Which outputs:

```
Tue Mar 18 14:02:37 EET 2014
2014-03-18T14:02:37.500
Tue Mar 18 14:02:36 EET 2014
2014-03-18T14:02:37.500
```

Sub Packages

- There are four sub-packages under java.time:
  - format – formatting and parsing instances.
  - zone – time zone supporting classes.
  - chrono – API for calendar systems other than the default ISO-8601.
  - temporal – extended API primarily for framework and library writers.

Date Classes

- The API provides four classes that deal exclusively with date information, without respect to time or time zone:
  - *LocalDate* – represents a year-month-date in the ISO calendar.
  - *YearMonth* – represents the month of a specific year.
  - *MonthDay* – represents the day of a specific month.
  - *Year* – represents a year.

Date Classes Examples

```
// Current date.
LocalDate today = LocalDate.now();

// ISO-8601 date format is YYYY-MM-DD.
LocalDate independence = LocalDate.of(1948, Month.MAY, 14);
assert "FRIDAY".equals(independence.getDayOfWeek().toString());

String april = LocalDate.ofYearDay
         (1000, 100).getMonth().toString();
assert april.equals("APRIL");

boolean leap = Year.isLeap(2014);

YearMonth bastille = YearMonth.of(1789, Month.JULY);
LocalDate bastilleDay = bastille.atDay(14);
assert bastilleDay.isBefore(independence);
```

Time Classes

- *LocalTime* – deals with time only.
- *LocalDateTime* – handles both date and time.
- Both classes do not store time zone information.
- Example:

```
// Current time.
LocalTime now = LocalTime.now();

// All three are equal.
LocalDateTime ldt1 = now.atDate(LocalDate.now());
LocalDateTime ldt2 = LocalDateTime.of(LocalDate.now(), now);
LocalDateTime ldt3 = LocalDateTime.now();
```

Time Zone Classes

- The API provides two classes for specifying a time zone or an offset:
  - ZoneId – specifies a time zone identifier.
  - ZoneOffset – an offset from GMT/UTC time.
- There are also three temporal-based classes that work with time zones:
  - *ZonedDateTime* – date and time with time zone and time zone offset.
  - *OffsetDateTime* - date and time with time zone offset but not time zone.
  - *OffsetTime* - time with time zone offset but not time zone.

Time Zone Example

```
// Returns a list of all available time zones.
Set<String> allZones = ZoneId.getAvailableZoneIds();

// MALAYSIA F1 Grand Prix starts at 16:00 30 MAR 2014.
LocalDate localDate = LocalDate.of(2014, Month.MARCH, 30);
LocalTime localTime = LocalTime.of(16, 0);

// Sets the time zone to Asia/Kuala_Lumpur.
ZonedDateTime malaysia = ZonedDateTime.of
        (localDate, localTime, ZoneId.of("Asia/Kuala_Lumpur"));
System.out.println(malaysia);
// 2013-03-30T16:00+08:00[Asia/Kuala_Lumpur]

// Convert to local time.
ZonedDateTime local =
        malaysia.withZoneSameInstant(ZoneId.systemDefault());
System.out.println(local);
// 2013-03-30T11:00+02:00[Europe/Athens]
```

Instant

- The *Instant* class represent an instantaneous point on an epoch based time-line.
- Useful for recording event time-stamps in applications.
- As opposed to the other classes in the package, it represents a machine based time.
- Example:

```
Instant epoch = Instant.EPOCH;
Instant now = Instant.now();
// Number of seconds passed since epoch.
long epochSeconds = now.getEpochSecond();
// Days passed since epoch until now.
long daysPassed = epoch.until(now, ChronoUnit.DAYS);
```

Legacy Code

- JDK 8 release also added several methods that allow conversion between java.util and java.time objects:
  - Calendar.toInstant() converts a *Calendar* to an *Instant*.
  - GregorianCalendar.toZonedDateTime() converts a *GregorianCalendar* to a *ZonedDateTime*.
  - GregorianCalendar.from(ZonedDateTime) creates a *GregorianCalendar* from a *ZonedDateTime*.
  - Date.from(Instant) creates a *Date* from an Instant.
  - Date.toInstant() converts a *Date* to an Instant.
  - TimeZone.toZoneId() converts a *TimeZone* to a *ZoneId*.

Notes

- There are more API classes and interfaces which are not covered here.
- For example: *Duration* and *Period*.
- We also haven't covered date formatting and parsing.
- Consult the API documentation for more information.

What's new in Java 8

- Lambda Expressions.
- Java.util.Function.
- Default Methods.
- Filter/map/reduce for Java.
- Date & Time API.
- Concurrency Additions.
- Reflection And Annotations.

LongAdder

- The *LongAdder* class provides an atomic way to update a common sum such as in collecting statistics.
- Internally, one or more variables maintain the sum.
- It should be preferred over *AtomicLong* when multiple threads update a common sum under high contention.
- Updates are done with the add() method. The set of maintaining variables may grow to reduce contention.
- The sum() method returns the current total, combined across the variables maintaining the sum.
- Similarly, there is a *DoubleAdder* companion class.

LongAccumulator

- The *LongAccumulator* class is conceptually similar to *LongAdder.*
- But instead, it's variables maintain a value updated with a supplied accumulation function.
- Again, prefer it over *AtomicLong* when multiple threads update a common value under high contention.
- Updates are done with the accumulate() method.
- The get() method returns the current value across the variables maintaining updates.
- There is also a *DoubleAccumulator* companion class.

LongAccumulator Example

```
// When accumulate() is being called, the function is applied.
// The left  parameter is the current sum value.
// The right parameter is the accumulate() parameter value.
LongBinaryOperator accumulationFunction = (l, r) -> l - r;

// The accumulator initial sum value is 100.
LongAccumulator stock = new
        LongAccumulator(accumulationFunction, 100);

stock.accumulate(5); //value is 95
stock.accumulate(5); //value is 90

// Resets the value to it's identity (initial value). E.g.: 100.
stock.reset(); // value is 100

stock.accumulate(5);  //value is 95
stock.accumulate(10); //value is 85
```

StampedLock

- Java 8 includes a new synchronization mechanism called *StampedLock*.
- It differentiates between exclusive and non-exclusive locks, similar to *ReentrantReadWriteLock*.
- However, it also allows for **optimistic** reads.
- When acquiring a lock, the lock returns a stamp of type long once the lock is granted.
- The lock's state consists of the stamp and a lock mode: Writing, Reading or Optimistic Reading.
- The stamp is used as a method parameter when calling unlock().

StampedLock

- The lock usage idiom is no different than any lock:

```
StampedLock lock = new StampedLock();
long stamp = lock.writeLock();
try {
    // Do something that needs exclusive access.
}
finally { lock.unlock(stamp); }
```

- Otherwise, what do we need the stamp for?
- The tryOptimisticRead() method returns a non-zero value if the lock is not currently held in write mode.
- Later, we can validate that the lock has not been acquired in a write mode since obtaining the stamp.

StampedLock Example

```
class Point {
  private double x, y;
  private final StampedLock sl = new StampedLock();

  void move(double deltaX, double deltaY) {
    long stamp = sl.writeLock();
    try {
      x += deltaX;
      y += deltaY;
    }
    finally {
      sl.unlockWrite(stamp);
    }
  }
```

StampedLock Example

```
double distanceFromZero() {
  long stamp;
  double currentX, currentY;

  do {
    // Note the optimistic mode: we don't use try/finally.
    // Also, stamp value is zero if a write lock is acquired.
    stamp = sl.tryOptimisticRead();
    currentX = x;
    currentY = y;
  }
  while (!sl.validate(stamp));

  return Math.sqrt
       (currentX * currentX + currentY * currentY);
}
}
```

Optimistic Locking

- validate() always returns false if the stamp is zero.
- It returns true only if the lock has not been exclusively acquired since the issuance of the stamp.
- Note that optimistic locking use is fragile. You should only read fields and hold them in local variables for later use after validation.
- For a long running computation adaptive spinning may be a killer.
- We can change our distance method to try an optimistic read, and if it fails, try a pessimistic one.

Example

```
double distanceFromZero() {
    long stamp = sl.tryOptimisticRead();

    double currentX = x, currentY = y;

    if (!sl.validate(stamp)) {
         // We can't validate the stamp. X and y may be mutated.
         // Block until a pesimistic read lock is acquired.
         stamp = sl.readLock();
         try {
             currentX = x;
             currentY = y;
          }
         finally { sl.unlockRead(stamp); }
    }

    return Math.sqrt
        (currentX * currentX + currentY * currentY);
}
```

Changing The Lock Mode

- You can also conditionally upgrade / downgrade a lock:
- The write upgrade for example will succeed if:
  - Already in a writing mode.
  - In a reading mode, where there are no other readers.
  - In an optimistic reading mode and lock is available.

```
sl.tryConvertToOptimisticRead(stamp);
sl.tryConvertToReadLock(stamp);
sl.tryConvertToWriteLock(stamp);
```

- Use optimistic mode for short read-only code segments which reduces contention and improves throughput.

Arrays Parallel Sorting

- With Java 8, you can sort arrays in parallel.
- You do it by calling **Arrays.*parallelSort*()**.
- You can parallel sort primitive arrays, arrays of types that implement *Comparable*, and also arrays of arbitrary types using a custom *Comparator*.
- Internally, the Fork/Join framework is used to assign sorting tasks to multiple threads.
- Note that by calling the method you are not guaranteed that the array will be sorted in parallel. It dependents on the array size and the machine parallelism.

ConcurrentHashMap

- *ConcurrentHashMap* has been retrofitted in Java 8.
- Performance is now better than ever.
- New idioms and operations corresponding with Lambda expressions were added.
- We will demonstrate some of the more interesting new features: forEach, reduce, search.
- Each performs over one of four variants: key, value, entry and (key, value).

forEach

```
ConcurrentHashMap<String, Person> map = new ConcurrentHashMap<>();
Person joe = Person.builder("joe", 42, "");
map.put(joe.getName(), joe);
Person schmo = Person.builder("schmo", 29, "");
map.put(schmo.getName(), schmo);

// The second parameter is a consumer function.
// We set the last seen field to now() for each person.
map.forEachValue(100, person ->
        person.setLastSeen(LocalDateTime.now()));

// The second parameter is a transformation function.
// We return the key as upper case.
// It doesn't affect the map key.
// The third parameter is again a consumer function.
Function<? super String, ?> transformer = String::toUpperCase;
Consumer<? super Object> action =
        o -> {System.out.println("Transformed key: " + o);};
map.forEachKey(100, transformer, action);
```

Parallelism Threshold

- The first parameter to the forEach methods is a parallelism threshold.
- It is the estimated number of elements needed for this operation to be executed in parallel.
- If the map size is lower, the method is executed sequentially.
- As always, you should measure performance values that trade off overhead versus throughput.

Reduce

- Note, there are much more reduce variants.

```
// threshold, transormer:Function<Person,Integer>,
// reducer:BiFunction<Integer,Integer,Integer>
int t1 = map.reduceValues(100, Person::getAge, (i1,i2) -> i1+i2);

// threshold, transormer:ToIntFunction<Person>,
// identity, reducer:IntBinaryOperator
int t2 = map.reduceValuesToInt
        (100, Person::getAge, 0, (left, right) -> left + right);
assert t1 == t2;

// Only threshold and reducer. No transformer here.
String allNames = map.reduceKeys
        (100, (s1, s2) -> String.format("%s %s" , s1, s2));
assert "joe schmo".equals(allNames);
```

Search

- search methods are less elegant.
- You should supply a *Function* that returns the first available non-null result, to a search method.
- It skips further search when a non-null result is found.

```
Function<String, String> notJoe =
         key ->  "joe".equals(key) ? null : key;
String key = map.searchKeys(100, notJoe);
assert "schmo".equals(key);

joe.setCountry(null);
schmo.setCountry("UK");
String country = map.searchValues(100, Person::getCountry);
assert "UK".equals(country);
```

The Fork/Join Framework

- We haven't covered improvements and additions to the Fork/Join Framework.
- It doesn't mean that these do not exist.
- Some notable additions are:
  - CompletionStage.
  - CompletableFuture.
  - CountedCompleter.
- Consult the API documentation for more.

What's new in Java 8

- Lambda Expressions.
- java.util.function.
- Default Methods.
- Filter/map/reduce for Java.
- Date & Time API.
- Concurrency Additions.
- Reflection And Annotations.

Repeating Annotations

- Prior to Java 8, duplicate annotations were not allowed in a class.

```
public @interface Love { String who(); }

@Love(who = "you")
@Love(who = "me")
// error: Duplicate annotation
public void confession {
}
```

- In Java 8, repeating annotations enable us to do so.
- For compatibility reasons, repeating annotations are stored in a container annotation. The compiler transforms the contained ones into it automatically.

Repeatable Annotation Type

- The annotation type must be annotated with the @*Repeatable* meta-annotation.
- The value of the @*Repeatable* meta-annotation is the type of the container annotation.

```
@Retention (RUNTIME)
@Target ({METHOD})
@Repeatable (Greetings.class)
public @interface Greeting {
    String value();
}
```

Container Annotation Type

- The container annotation type must have a value element with an array type.
- The component type of the array must be the repeatable annotation type.

```
@Retention (RUNTIME)
@Target ({METHOD})
public @interface Greetings {
    Greeting[ ] value();
}
```

A Usage Example

- Bellow is a usage example.

```
public class UserService {

    @Greeting("welcome")
    @Greeting("hello")
    public void register() { }

    @Greeting("hello")
    public void login() { }
}
```

- Let's see some interesting reflection facts.

Reflection Example

```
Method register = UserService.class.getMethod("register");
// Returns null once the annotation of the requested type is
repeated.
assert null == register.getAnnotation(Greeting.class);

// getAnnotationsByType is new in Java 8.
Greeting[] greetingByType =
         register.getAnnotationsByType(Greeting.class);
assert greetingByType.length == 2;

// The container annotation.
Greetings greetings = register.getAnnotation(Greetings.class);
Greetings[] greetingsByType =
         register.getAnnotationsByType(Greetings.class);

assert greetingsByType.length == 1;
assert greetingsByType[0] == greetings;

Arrays.asList(greetings.value()).forEach(System.out::println);
// @com.trainologic.wnij8.anno.Greeting(value=hello)
// @com.trainologic.wnij8.anno.Greeting(value=welcome)
```

Reflection Example

```
Method login = UserService.class.getMethod("login");

// This time, getAnnotation returns a value.
Greeting greeting = login.getAnnotation(Greeting.class);
assert "hello".equals(greeting.value());

Greeting[] greetings =
        login.getAnnotationsByType(Greeting.class);
assert greetings.length == 1;
assert greetings[0] == greeting;

assert 0 == login.getAnnotationsByType(Greetings.class).length;
assert null == login.getAnnotation(Greetings.class);
```

Discussion

- As we can see, the getAnnotation() method behaves differently once a target annotation is repeated:
  - Returns a single object if the target annotation is not repeated.
  - Returns null otherwise.
- This is reasonable, as there is no way to choose between multiple choices. It is also backward compatible.
- In contrast, the getAnnotationsByType() is consistent. It always returns an array of target annotation(s), whether single, repeated, or not existent.
- Consider using the new method.

Type Annotations

- As of the Java SE 8 release, annotations can be applied to any type use.
- This means that you can use type annotations everywhere you use a type.
- When writing a type annotation, you must set it's *ElementType* to **TYPE_USE**.
- Bellow is such a type annotation, followed by a usage example.

```
@Target({ElementType.TYPE_USE})
public @interface TypeMarker { }
```

Type Annotation Example

```
public void foo() {
    // A constructor with a type annotation:
    Map<String, Person> map = new @TypeMarker HashMap<>();

    // Type cast using a type annotation:
    HashMap<String, Person> other =
        (@TypeMarker HashMap<String, Person>) map;

    // Type annotations in instanceof statement:
    assert "string" instanceof @TypeMarker String;
}

// Throwing exception:
public void throwing() throws @TypeMarker Exception {}

// Inheritance:
abstract class MarkedList<T> implements @TypeMarker List<T> {}
```

Generic Type Parameters

- You can also use type parameters with a type annotation.
- By setting it's *ElementType* to **TYPE_PARAMTER**.
- Type annotation example:

```
@Target({ElementType.TYPE_PARAMETER})
public @interface GenericTypeMarker { }
```

- A usage example:

```
public class GenericTypeContainer<@GenericTypeMarker T> { }
```

Usage

- As with all annotations, type annotations do nothing on their own.
- Our example annotations will not work out of the box.
- Type Annotations allow improved analysis of Java code and can ensure even stronger type checking.
- It is up to frameworks and tools to make use of it.
- A nice existing tool is the Checker Framework. It can perform additional code checks based on a set of type annotations.
- Checks happen during compilation as the tool is a compiler plug-in.

Parameter And Executable

- The new *Parameter* class contains information about a method or a constructor parameters.
- Including it's type, name and modifiers.
- The new *Executable* class is a shared super-class for the common functionality of *Method* and *Constructor*.
- The *Executable* class contains a getParameters() method returning it's parameters.
- So now you can access method parameters with their names and modifiers.

Example

```
public String baz(final String s, int i) {
    return String.format("(%d)%s", i, s);
}

Method baz = ParametersExamples.class.getMethod
        ("baz" , String.class, int.class);

List<Parameter> list = Arrays.asList(baz.getParameters());
list.forEach(p -> {
    System.out.println(p);
    assert p.getDeclaringExecutable().equals(baz);
});

// java.lang.String arg0
// int arg1

// The names are not stored in the class file by default.
// Enable names storage by the compiler flag: -g:vars
```