Mature Java 8

- **language features**
    - lambdas
    - default and static methods
- **standard library improvements**
    - Stream API
    - Optional
    - CompletableFuture
- **libraries influence**
    - RxJava
    - Guava
    - JUnit

Lambdas

```java
public void sortByLength(List<String> strings) {
    Collections.sort(strings, new Comparator<String>() {
        @Override
        public int compare(String s1, String s2) {
            return Integer.compare(s1.length(), s2.length());
        }
    });
}
```

```java
public void sortByLength(List<String> strings) {
    Collections.sort(strings, new Comparator<String>() {
        @Override
        public int compare(String s1, String s2) {
            return Integer.compare(s1.length(), s2.length());
        }
    });
}
```

```
public void sortByLength(List<String> strings) {
    Collections.sort(strings, (String s1, String s2)
                -> Integer.compare(s1.length(), s2.length()));
}
```

```
public void sortByLength(List<String> strings) {
    Collections.sort(strings, (String s1, String s2)
                -> Integer.compare(s1.length(), s2.length()));
}
```

```
public void sortByLength(List<String> strings) {
    Collections.sort(strings, (s1, s2)
                -> Integer.compare(s1.length(), s2.length()));
}
```

```
public void sortByLength(List<String> strings) {
    Collections.sort(strings, Comparator.comparingInt(String::length));
}
```

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
static <T,U extends Comparable<? super U>> Comparator<T>	comparing(Function<? super T,? extends U> keyExtractor) Accepts a function that extracts a Comparable sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T,U> Comparator<T>	comparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator) Accepts a function that extracts a sort key from a type T, and returns a Comparator<T> that compares by that sort key using the specified Comparator.
static <T> Comparator<T>	comparingDouble(ToDoubleFunction<? super T> keyExtractor) Accepts a function that extracts a double sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T> Comparator<T>	comparingInt(ToIntFunction<? super T> keyExtractor) Accepts a function that extracts an int sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T> Comparator<T>	comparingLong(ToLongFunction<? super T> keyExtractor) Accepts a function that extracts a long sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T extends Comparable<? super T>> Comparator<T>	naturalOrder() Returns a comparator that compares Comparable objects in natural order.
static <T> Comparator<T>	nullsFirst(Comparator<? super T> comparator) Returns a null-friendly comparator that considers null to be less than non-null.
static <T> Comparator<T>	nullsLast(Comparator<? super T> comparator) Returns a null-friendly comparator that considers null to be greater than non-null.
static <T extends Comparable<? super T>> Comparator<T>	reverseOrder() Returns a comparator that imposes the reverse of the *natural ordering*.

```
Exception in thread "main" java.lang.NullPointerException
    at com.avsystem.java.Presentation$1.compare(Presentation.java:24)
    at com.avsystem.java.Presentation$1.compare(Presentation.java:21)
    at java.util.TimSort.countRunAndMakeAscending(TimSort.java:360)
    at java.util.TimSort.sort(TimSort.java:220)
    at java.util.Arrays.sort(Arrays.java:1512)
    at java.util.ArrayList.sort(ArrayList.java:1454)
    at java.util.Collections.sort(Collections.java:175)
    at com.avsystem.java.Presentation.sortByLength(Presentation.java:21)
    at com.avsystem.java.Presentation.main(Presentation.java:17)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.jav
a:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at com.intellij.rt.execution.application.AppMain.main(AppMain.java:147)
```

```
Exception in thread "main" java.lang.NullPointerException
    at java.util.Comparator.lambda$comparingInt$7b0bb60$1(Comparator.java:490)
    at java.util.TimSort.countRunAndMakeAscending(TimSort.java:360)
    at java.util.TimSort.sort(TimSort.java:220)
    at java.util.Arrays.sort(Arrays.java:1512)
    at java.util.ArrayList.sort(ArrayList.java:1454)
    at java.util.Collections.sort(Collections.java:175)
    at com.avsystem.java.Presentation.sortByLengthLambda(Presentation.java:30)
    at com.avsystem.java.Presentation.main(Presentation.java:17)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.jav
a:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at com.intellij.rt.execution.application.AppMain.main(AppMain.java:147)
```

Stream API

```java
public int parseAndSumForeach(List<String> strings) {
    int sum = 0;
    for (String s : strings) {
        int i = Integer.parseInt(s);
        if (i > 0) {
            sum += i;
        }
    }
    return sum;
}
```

```java
public long parseAndSum(List<String> strings) {
    return strings.stream()
            .mapToInt(Integer::parseInt)
            .filter(i -> i > 0)
            .sum();
}
```

```java
public int parseAndSumForeach(List<String> strings) {
    int sum = 0;
    for (String s : strings) {
        int i = Integer.parseInt(s);
        if (i > 0) {
            sum += i;
        }
    }
    return sum;
}
```

```java
public long parseAndSum(List<String> strings) {
    return strings.stream()
            .mapToInt(Integer::parseInt)
            .filter(i -> i > 0)
            .sum();
}
```

```java
public int parseAndSumForeach(List<String> strings) {
    int sum = 0;
    for (String s : strings) {
        int i = Integer.parseInt(s);
        if (i > 0) {
            sum += i;
        }
    }
    return sum;
}
```

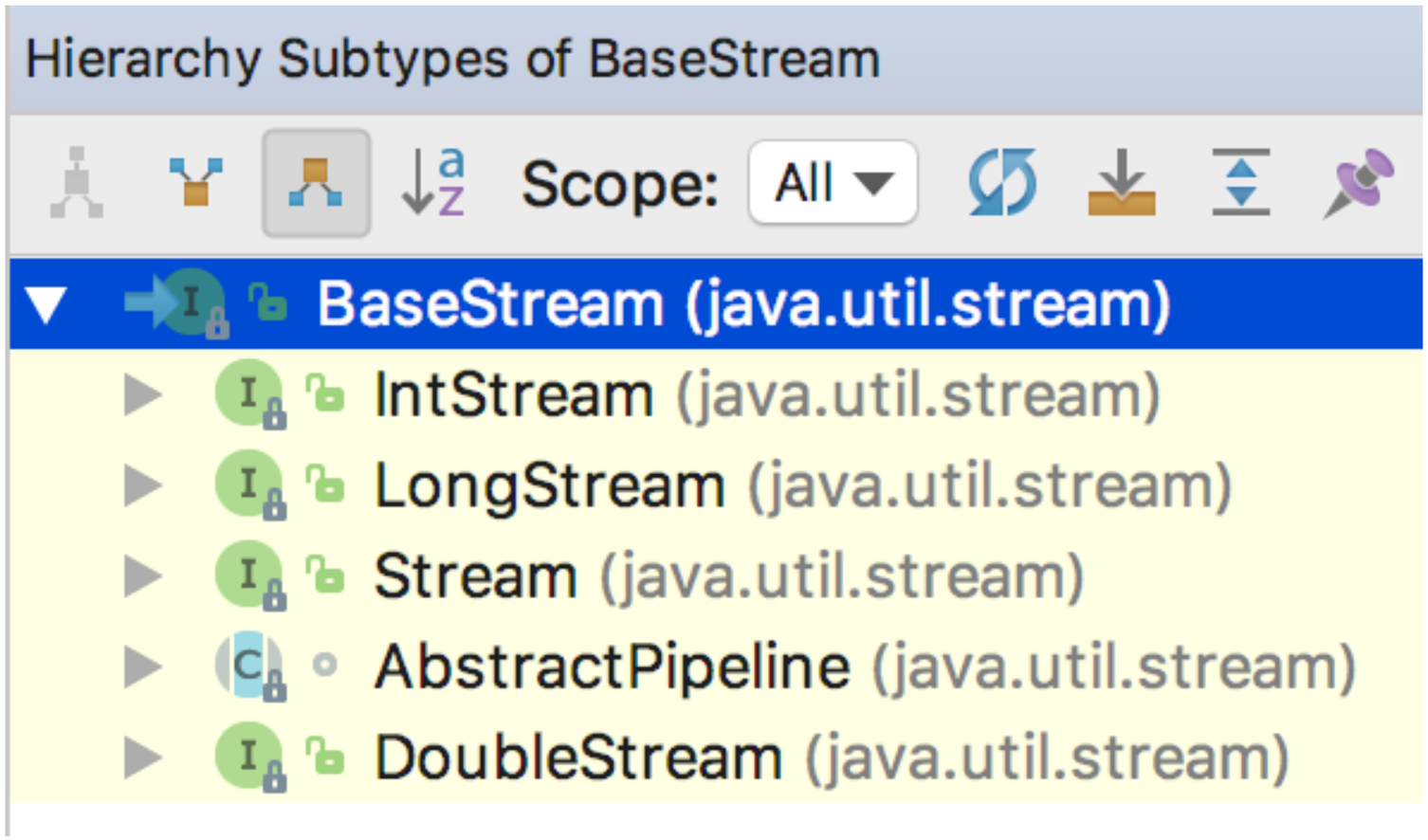
```java
public long parseAndSum(List<String> strings) {
    return strings.stream()
            .mapToInt(Integer::parseInt)
            .filter(i -> i > 0)
            .sum();
}
```

Protip

```
public long parseAndSum(List<String> strings) {
    return strings.stream()
            .mapToInt(Integer::parseInt)
            .filter(i -> i > 0)
            .sum();
    }
```

```
IntStream                    mapToInt(ToIntFunction<? super T> mapper)
```

Protip


Hierarchy Subtypes of BaseStream
Scope:
All
BaseStream (java.util.stream)
IntStream (java.util.stream)
LongStream (java.util.stream)
Stream (java.util.stream)
AbstractPipeline (java.util.stream)
DoubleStream (java.util.stream)

Protip

```
public long parseAndSum(List<String> strings) {
    return strings.stream()
            .mapToInt(Integer::parseInt)
            .filter(i -> i > 0)
            .sum();
}
```

```
static int                              parseInt(String s)
```

```
{
  "name": "Fred",
  "age": 20,
  "height": 185
}
```

```
{
  "name": "Staszek",
  "age": 29,
  "height": 170
}
```

```
{
  "name": "Fred",
  "age": 20,
  "height": 185
}
```

```
{
  "name": "Staszek",
  "age": 29,
  "height": 170
}
```

```
{
  "name": ["Fred", "Staszek"],
  "age": [20, 29],
  "height": [185, 170]
}
```

```
{
  "age": 20,
  "height": 185
}
```

```
{
  "age": 29,
  "height": 170
}
```

```
public Map<String, List<Integer>> group(List<Map<String, Integer>> data)
```

```
{
  "age": 20,
  "height": 185
}
```

```
{
  "age": 29,
  "height": 170
}
```


```
public Map<String, List<Integer>> group(List<Map<String, Integer>> data) {
    Map<String, List<Integer>> result = new HashMap<>();
    for (Map<String, Integer> map : data) {
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            List<Integer> resultPerKey =
                    result.computeIfAbsent(entry.getKey(), k -> new LinkedList<>());
            resultPerKey.add(entry.getValue());
        }
    }
    return result;
}
```

```
{
  "age": 20,
  "height": 185
}
```

```
{
  "age": 29,
  "height": 170
}
```


```java
public Map<String, List<Integer>> group(List<Map<String, Integer>> data) {
    Map<String, List<Integer>> result = new HashMap<>();
    for (Map<String, Integer> map : data) {
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            List<Integer> resultPerKey =
                    result.computeIfAbsent(entry.getKey(), k -> new LinkedList<>());
            resultPerKey.add(entry.getValue());
        }
    }
    return result;
}
```

```
{
  "age": 20,
  "height": 185,
  "age": 29,
  "height": 170
}
```


```
public Map<String, List<Integer>> group(List<Map<String, Integer>> data) {
    Map<String, List<Integer>> result = new HashMap<>();
    for (Map<String, Integer> map : data) {
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            List<Integer> resultPerKey =
                    result.computeIfAbsent(entry.getKey(), k -> new LinkedList<>());
            resultPerKey.add(entry.getValue());
        }
    }
    return result;
}
```

```
{
  "age": 20,
  "height": 185,
  "age": 29,
  "height": 170
}
```


```
public Map<String, List<Integer>> group(List<Map<String, Integer>> data) {
    Map<String, List<Integer>> result = new HashMap<>();

    for (Map<String, Integer> map : data) {
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            List<Integer> resultPerKey =
                    result.computeIfAbsent(entry.getKey(), k -> new LinkedList<>());
            resultPerKey.add(entry.getValue());
        }
    }
    return result;
}
```

```
{
  "age": 20,
  "height": 185,
  "age": 29,
  "height": 170
}
```


```
public Map<String, List<Integer>> group(List<Map<String, Integer>> data) {
    Map<String, List<Integer>> result = new HashMap<>();
    for (Map<String, Integer> map : data) {
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            List<Integer> resultPerKey =
                    result.computeIfAbsent(entry.getKey(), k -> new LinkedList<>());
            resultPerKey.add(entry.getValue());
        }
    }
    return result;
}
```

```
{
  "age": 20,
  "height": 185
}
```

```
{
  "age": 29,
  "height": 170
}
```


```
public Map<String, List<Integer>> group(List<Map<String, Integer>> data) {
    return data.stream()
            .flatMap(map -> map.entrySet().stream())
            .collect(
                    Collectors.groupingBy(Map.Entry::getKey,
                            Collectors.mapping(
                                    Map.Entry::getValue, Collectors.toList())))
            );
}
```

```
{
  "age": 20,
  "height": 185,
  "age": 29,
  "height": 170
}
```

```
public Map<String, List<Integer>> group(List<Map<String, Integer>> data) {
    return data.stream()
            .flatMap(map -> map.entrySet().stream())
            .collect(
                    Collectors.groupingBy(Map.Entry::getKey,
                            Collectors.mapping(
                                    Map.Entry::getValue, Collectors.toList())))
            );
}
```

```json
{
  "age": 20,
  "height": 185,
  "age": 29,
  "height": 170
}
```

```java
public Map<String, List<Integer>> group(List<Map<String, Integer>> data) {
    return data.stream()
            .flatMap(map -> map.entrySet().stream())
            .collect(
                    Collectors.groupingBy(Map.Entry::getKey,
                            Collectors.mapping(
                                    Map.Entry::getValue, Collectors.toList())))
            );
}
```

```
{
  "age": 20,
  "height": 185,
  "age": 29,
  "height": 170
}
```


```
public Map<String, List<Integer>> group(List<Map<String, Integer>> data) {
    return data.stream()
            .flatMap(map -> map.entrySet().stream())
            .collect(
                    Collectors.groupingBy(Map.Entry::getKey,
                            Collectors.mapping(
                                    Map.Entry::getValue, Collectors.toList())))
            );
}
```

```
{
  "age": 20,
  "height": 185,
  "age": 29,
  "height": 170
}
```


```
public Map<String, List<Integer>> group(List<Map<String, Integer>> data) {
    return data.stream()
            .flatMap(map -> map.entrySet().stream())
            .collect(
                    Collectors.groupingBy(Map.Entry::getKey,
                            Collectors.mapping(
                                    Map.Entry::getValue, Collectors.toList())))
            );
}
```

```
Collector<? super Entry<String, Integer>, ? extends Object, Map<String, List<Integer>>> collector
46                  .flatMap(map -> map.entrySet().stream())
47                  .collect(
48                  );
49          }
```

```
{
  "age": 20,
  "height": 185,
  "age": 29,
  "height": 170
}
```

```
public Map<String, List<Integer>> group(List<Map<String, Integer>> data) {
    return data.stream()
            .flatMap(map -> map.entrySet().stream())
            .collect(
                    Collectors.groupingBy(Map.Entry::getKey,
                            Collectors.mapping(
                                    Map.Entry::getValue, Collectors.toList())))
            );
}
```

Optional

Optional

```
public interface PersonDao {
    Optional<Person> findById(String id);
}
```

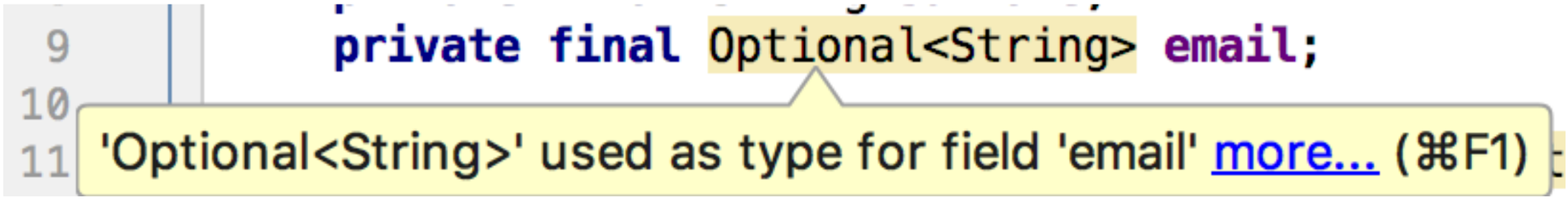
Optional

```
public interface PersonDao {
    Optional<Person> findById(String id);
}
```

Optional

```
public class Person {
    private final String name;
    private final String surname;
    private final Optional<String> email;
}
```

Optional


9 private final Optional<String> email;
10
11
'Optional<String>' used as type for field 'email' more... (⌘F1)

Optional

```
9       private final Optional<String> email;
10
11  'Optional<String>' used as type for field 'email' more... (⌘F1)
```

Optional

```
public class Person {
    private final String name;
    private final String surname;
    private final Optional<String> email;
}
```

Optional

```
public Person(
        String name,
        String surname,
        Optional<String> email,
        Optional<String> phoneNumber,
        Optional<LocalDate> birthdate
) {
    this.name = name;
    this.surname = surname;
    this.email = email;
    this.phoneNumber = phoneNumber;
    birthDate = birthdate;
}
```

Optional

```
public static Person newPerson() {
    return new Person(
                    "Fred",
                    "Fredowicz",
                    Optional.of("fred@fredowicz.com"),
                    Optional.empty(),
                    Optional.of(LocalDate.now())
    );
}
```

Optional

```
public static Person newPerson() {
    return new Person(
                    "Fred",
                    "Fredowicz",
                    "fred@fredowicz.com",
                    null,
                    LocalDate.now()
    );
}
```

Optional

```
public Person(String name, String surname) { }
public Person(String name, String surname, String email) {}
public Person(String name, String surname, String email, String phoneNumber) { }
public Person(String name, String surname, String email, String phoneNumber, LocalDate birthdate)
{ }
```

Optional

Optional

Optional

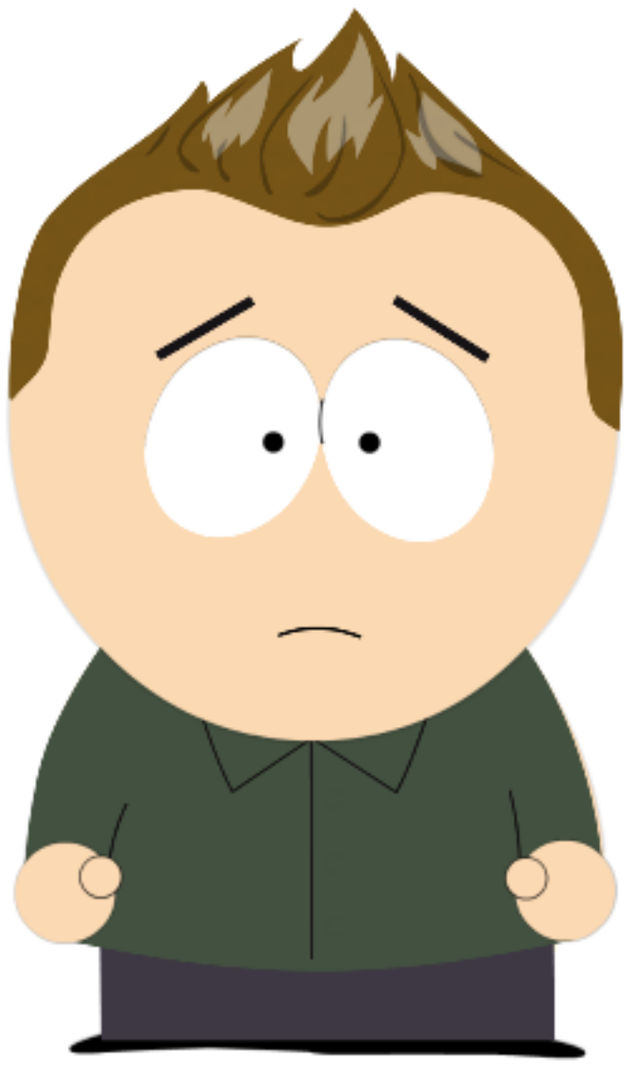
```
public class Person {
    private final String name;
    private final String surname;
    private final Optional<Integer> birthYear;

    public Person(String name, String surname, Optional<Integer> birthYear) {
        this.name = name;
        this.surname = surname;
        this.birthYear = birthYear;
    }
}
```

Optional

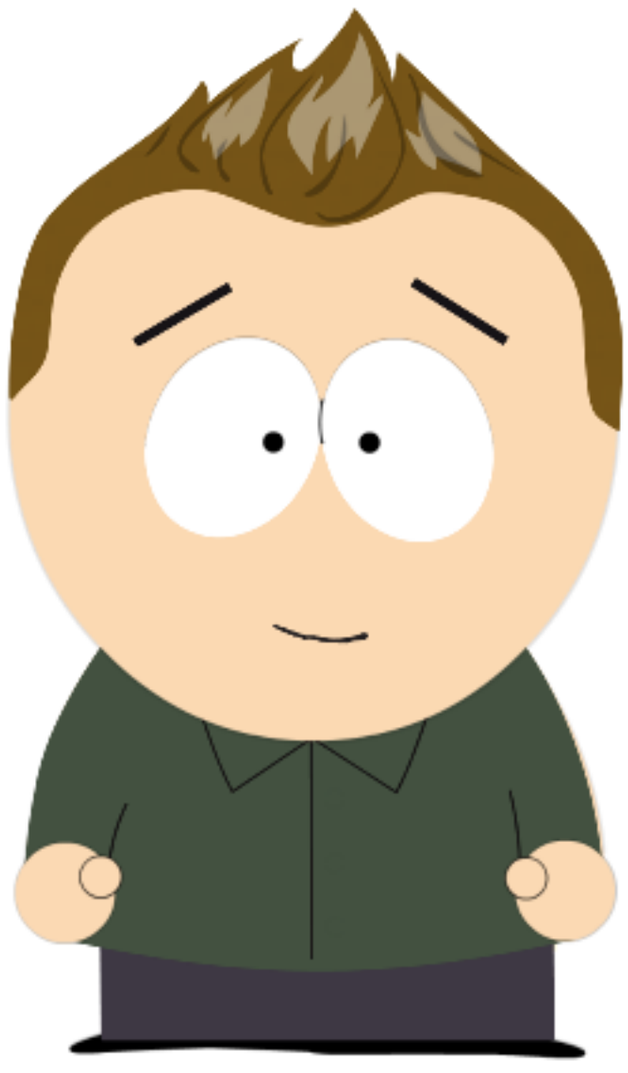
```
public class Person {
    private final String name;
    private final String surname;
    private final OptionalInt birthYear;

    public Person(String name, String surname, OptionalInt birthYear) {
        this.name = name;
        this.surname = surname;
        this.birthYear = birthYear;
    }
}
```

Date Time API

java.util.Date



org.joda.time.LocalDateTime



java.time.LocalDateTime

Note that from Java SE 8 onwards, users are asked to migrate to java.time (JSR-310) - a core part of the JDK which replaces this project.

CompletableFuture

CompletableFuture

```
public static void loadFromInternetAndPrint() {
    CompletableFuture<String> eventualString = loadIntAsStringFromInternet();

    eventualString
            .thenApply(Integer::parseInt)
            .thenCompose(i -> heavyComputation(i))
            .thenAccept(value -> value.ifPresent(System.out::println));
}

private static CompletableFuture<OptionalInt> heavyComputation(Integer i) {
    return CompletableFuture.supplyAsync(() -> {
        return (i > 0) ? OptionalInt.of(i) : OptionalInt.empty();
    });
}
```

CompletableFuture

```
public static void loadFromInternetAndPrint() {
    CompletableFuture<String> eventualString = loadIntAsStringFromInternet();

    eventualString
            .thenApply(Integer::parseInt)
            .thenCompose(i -> heavyComputation(i))
            .thenAccept(value -> value.ifPresent(System.out::println));
}

private static CompletableFuture<OptionalInt> heavyComputation(Integer i) {
    return CompletableFuture.supplyAsync(() -> {
        return (i > 0) ? OptionalInt.of(i) : OptionalInt.empty();
    });
}
```

CompletableFuture

```
public static void loadFromInternetAndPrint() {
    CompletableFuture<String> eventualString = loadIntAsStringFromInternet();

    eventualString
            .thenApply(Integer::parseInt)
            .thenCompose(i -> heavyComputation(i))
            .thenAccept(value -> value.ifPresent(System.out::println));
}

private static CompletableFuture<OptionalInt> heavyComputation(Integer i) {
    return CompletableFuture.supplyAsync(() -> {
        return (i > 0) ? OptionalInt.of(i) : OptionalInt.empty();
    });
}
```

CompletableFuture

```
public static void loadFromInternetAndPrint() {
    CompletableFuture<String> eventualString = loadIntAsStringFromInternet();

    eventualString
            .thenApply(Integer::parseInt)
            .thenCompose(i -> heavyComputation(i))
            .thenAccept(value -> value.ifPresent(System.out::println));
}

private static CompletableFuture<OptionalInt> heavyComputation(Integer i) {
    return CompletableFuture.supplyAsync(() -> {
        return (i > 0) ? OptionalInt.of(i) : OptionalInt.empty();
    });
}
```

CompletableFuture

```
public static void loadFromInternetAndPrint() {
    CompletableFuture<String> eventualString = loadIntAsStringFromInternet();

    eventualString
            .thenApply(Integer::parseInt)
            .thenCompose(i -> heavyComputation(i))
            .thenAccept(value -> value.ifPresent(System.out::println));
}

private static CompletableFuture<OptionalInt> heavyComputation(Integer i) {
    return CompletableFuture.supplyAsync(() -> {
        return (i > 0) ? OptionalInt.of(i) : OptionalInt.empty();
    });
}
```

CompletableFuture

```
public static void loadFromInternetAndPrint() {
    CompletableFuture<String> eventualString = loadIntAsStringFromInternet();

    eventualString
            .map(Integer::parseInt)
            .flatMap(i -> heavyComputation(i))
            .forEach(value -> value.ifPresent(System.out::println));
}

private static CompletableFuture<OptionalInt> heavyComputation(Integer i) {
    return CompletableFuture.supplyAsync(() -> {
        return (i > 0) ? OptionalInt.of(i) : OptionalInt.empty();
    });
}
```

CompletableFuture

```
public static void loadFromInternetAndPrint() {
    Stream<String> eventualString = loadIntAsStringFromInternet();

    eventualString
            .map(Integer::parseInt)
            .flatMap(i -> heavyComputation(i))
            .forEach(value -> value.ifPresent(System.out::println));
}

private static Stream<OptionalInt> heavyComputation(Integer i) {
    OptionalInt result = (i > 0) ? OptionalInt.of(i) : OptionalInt.empty();
    return Stream.of(result);
}
```

CompletableFuture

RxJava

RxJava - Single

```
public static void loadFromInternetAndPrint() {
    Single<String> eventualString = loadIntAsStringFromInternet();

    eventualString
            .map(Integer::parseInt)
            .flatMap(i -> heavyComputation(i))
            .subscribe(value -> value.ifPresent(System.out::println));
}
```

RxJava - Single

```
public static void loadFromInternetAndPrint() {
    Single<String> eventualString = loadIntAsStringFromInternet();

    eventualString
            .map(Integer::parseInt)
            .flatMap(i -> heavyComputation(i))
            .subscribe(
                    value -> value.ifPresent(System.out::println),
                    error -> logError(error)
            );
}
```

RxJava - Single

```
public static void loadFromInternetAndPrint() {
    Single<String> eventualString = loadIntAsStringFromInternet();

    eventualString
            .map(Integer::parseInt)
            .flatMap(i -> heavyComputation(i))
            .subscribe(
                    value -> value.ifPresent(System.out::println),
                    error -> logError(error)
            );
}
```

RxJava

RxJava

	single items	multiple items
synchronous	`T getData()`	`Iterable<T> getData()`
asynchronous	`Future<T> getData()`	`Observable<T> getData()`

RxJava

	single items	**multiple items**
synchronous	`T`	`Stream<T>`
asynchronous	`CompletableFuture<T>`	`Observable<T>`

RxJava - Observable

```
public static void loadFromInternetAndPrint() {
    Observable<String> strings = loadIntAsStringFromInternet();

    strings
            .map(Integer::parseInt)
            .flatMap(i -> heavyComputation(i))
            .subscribe(
                    value -> value.ifPresent(System.out::println),
                    error -> logError(error)
            );
}
```

RxJava - Single

```
public static void loadFromInternetAndPrint() {
    Single<String> eventualString = loadIntAsStringFromInternet();

    eventualString
            .map(Integer::parseInt)
            .flatMap(i -> heavyComputation(i))
            .subscribe(
                    value -> value.ifPresent(System.out::println),
                    error -> logError(error)
            );
}
```

RxJava - Observable

```
public static void loadFromInternetAndPrint() {
    Observable<String> strings = loadIntAsStringFromInternet();

    strings
            .map(Integer::parseInt)
            .flatMap(i -> heavyComputation(i))
            .subscribe(
                    value -> value.ifPresent(System.out::println),
                    error -> logError(error)
            );
}
```

RxJava

Reactive Streams

Reactive Streams

```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}

public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}

public interface Subscription {
    public void request(long n);
    public void cancel();
}

public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {}
```

Guava

Guava - obsoletes

- Optional

Guava - obsoletes

- Optional
- Ordering

Guava - obsoletes

- Optional
- Ordering
- ListenableFuture

Guava - obsoletes

- Optional
- Ordering
- ListenableFuture
- Functional utils (Function, Predicate etc.)

Guava - obsoletes

- Optional
- Ordering
- ListenableFuture
- Functional utils (Function, Predicate etc.)
- FluentIterable

Guava - obsoletes

- Optional
- Ordering
- ListenableFuture
- Functional utils (Function, Predicate etc.)
- FluentIterable
- Range

Guava - obsoletes

- Optional
- Ordering
- ListenableFuture
- Functional utils (Function, Predicate etc.)
- FluentIterable
- Range
- StringJoiner

Guava - still cool

- Cache

Guava - still cool

- Cache
- Immutable collections

Guava - still cool

- Cache
- Immutable collections
- String operations

Guava - still cool

- Cache
- Immutable collections
- String operations
- Primitives

Guava - still cool

- Cache
- Immutable collections
- String operations
- Primitives
- Event bus

Guava - still cool

- Cache
- Immutable collections
- String operations
- Primitives
- Event bus
- etc.

JUnit 5

JUnit 5

JUnit 5 = *JUnit Platform* + *JUnit Jupiter* + *JUnit Vintage*

JUnit Jupiter

JUnit Jupiter

```
assertTrue(() -> 2 == 2, () -> "Lazily evaluated condition and message");

Throwable exception = assertThrows(IllegalArgumentException.class, () -> {
    throw new IllegalArgumentException("a message");
});
assertEquals("a message", exception.getMessage());
```

JUnit Jupiter

```
public static void assertTrue(BooleanSupplier booleanSupplier,
                              Supplier<String> messageSupplier) {
    AssertTrue.assertTrue(booleanSupplier, messageSupplier);
}


public static <T extends Throwable> T assertThrows(
        Class<? extends Throwable> expectedType,
        Executable executable) {
    return AssertThrows.assertThrows(expectedType, executable);
}
```

JUnit Jupiter

```
public interface EqualsContract<T>  {
    T createValue();

    @Test
    default void valueEqualsToItself() {
        T value = createValue();
        Assertions.assertEquals(value, value);
    }

    @Test
    default void valueDoesNotEqualNull() {
        T value = createValue();
        assertFalse(value.equals(null));
    }
}

public class StringTest implements EqualsContract<String>, ComparableContract<String> {
    @Override
    public String createValue() {
        return "fred";
    }
}
```

JUnit Jupiter

```
@TestFactory
Stream<DynamicTest> dynamicTestsFromStream() {
    return Stream.of("A", "B", "C").map(
            str -> dynamicTest("test" + str, () -> { /* ... */ }));
}

@TestFactory
Stream<DynamicTest> dynamicTestsFromIntStream() {
    // Generates tests for the first 10 even integers.
    return IntStream.iterate(0, n -> n + 2).limit(10).mapToObj(
            n -> dynamicTest("test" + n, () -> assertTrue(n % 2 == 0)));
}
```

JUnit 5

Summary

Summary

- lambdas are cool - except exceptions

Summary

- lambdas are cool - except exceptions
- streams are cool - but for simple, linear operations

Summary

- lambdas are cool - except exceptions
- streams are cool - but for simple, linear operations
- optionals are cool - but you need to be consistent

Summary

- lambdas are cool - except exceptions
- streams are cool - but for simple, linear operations
- optionals are cool - but you need to be consistent
- date time API is cool

Summary

- lambdas are cool - except exceptions
- streams are cool - but for simple, linear operations
- optionals are cool - but you need to be consistent
- date time API is cool
- CompletableFutures are lame - use RxJava

Summary

- lambdas are cool - except exceptions
- streams are cool - but for simple, linear operations
- optionals are cool - but you need to be consistent
- date time API is cool
- CompletableFutures are lame - use RxJava
- quarter of Guava is obsolete

Summary

- lambdas are cool - except exceptions
- streams are cool - but for simple, linear operations
- optionals are cool - but you need to be consistent
- date time API is cool
- CompletableFutures are lame - use RxJava
- quarter of Guava is obsolete
- JUnit 5 is completely redesigned thanks to Java 8