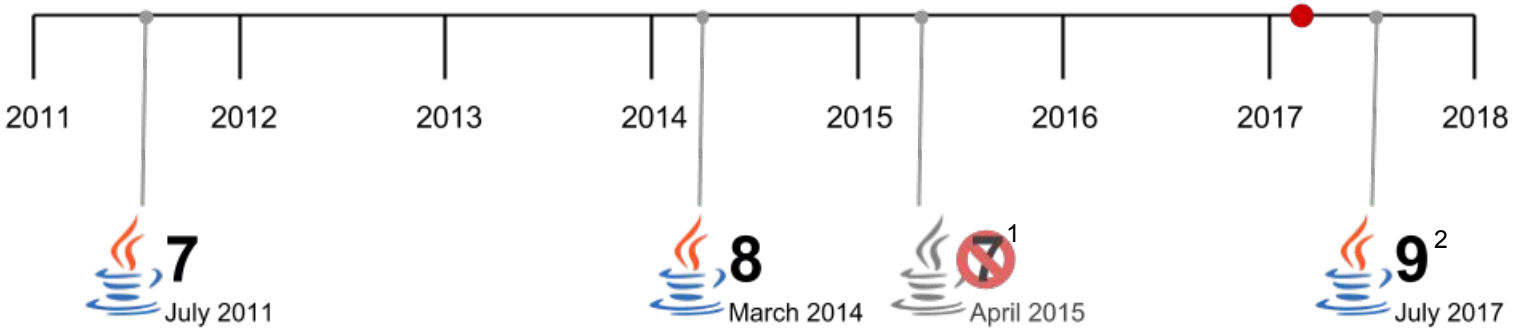
Java 8

Java release timeline



[1] https://java.com/en/download/faq/java_7.xml

[2] http://www.java9countdown.xyz/

What's new[3] - Language (1/3)

- **Lambdas & Method references**
- **Default methods (interfaces)**
- Repeating annotations
- Type annotations
- Improved type inference
- **Method parameter reflection**

[3] http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

What's new[3] - APIs (2/3)

- **Streaming (java.util.stream.*)**
- **Optional (java.util.optional)**
- **Time (java.time.*)**

[3] http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

What's new[3] - Other (3/3)

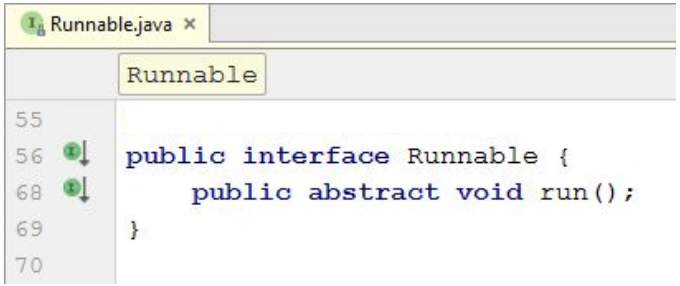
- Nashorn
- Security improvements
- JavaFX
- Compact profiles
- Improved javac and javadoc tool
- Unicode enhancements
- Concurrency improvements, DB improvements, networking improvements….
- Many more...

[3] http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

Lambdas

Passing behaviour as arguments

Lambdas


Runnable.java
Runnable
55
56
68
69
70
public interface Runnable {
public abstract void run();
}

Lambdas

Runnable.java

```
Runnable
55
56  public interface Runnable {
68      public abstract void run();
69  }
70
```

Lambdas.java

```
5       final Runnable r1 = new Runnable() {
6           @Override
7           public void run() {
8               System.out.println("Executing logic...");
9           }
10      };
11      r1.run();
```

Lambdas

Runnable.java

```
Runnable
55  @FunctionalInterface
56  public interface Runnable {
68      public abstract void run();
69  }
70
```

Lambdas.java

```
5       final Runnable r1 = new Runnable() {
6           @Override
7           public void run() {
8               System.out.println("Executing logic...");
9           }
10      };
11      r1.run();
```

Lambdas

Runnable.java

```
Runnable
55  @FunctionalInterface
56  public interface Runnable {
68      public abstract void run();
69  }
70
```

Lambdas.java

```
5       final Runnable r1 = new Runnable() {
6           @Override
7           public void run() {
8               System.out.println("Executing logic...");
9           }
10      };
11      r1.run();
12
13      final Runnable r2 = () -> System.out.println("Executing logic");
14      r2.run();
```

Lambdas

Java lambdas are syntactic sugar for anonymous classes implementing a functional interface

Lambdas - Functional Interfaces

Consumer.java

```
41  @FunctionalInterface
42  public interface Consumer<T> {
49      void accept(T t);
67  }
68
```

Predicate.java

```
41  @FunctionalInterface
42  public interface Predicate<T> {
49      boolean test(T t);
67  }
68
```

Function.java

```
41  @FunctionalInterface
42  public interface Function<T, R> {
49      R apply(T t);
67  }
68
```

Lambdas - Examples

Lambdas2.java

```
8       public void runnableExample() {
9           final Thread thread = new Thread(() -> {
10              System.out.println("Running thread now");
11              System.out.println("Executing some logic");
12              System.out.println("Ending thread");
13          });
14          thread.start();
15      }
16
17      public void consumerExample(final List<Integer> numbers) {
18          numbers.forEach((Integer number) -> System.out.println(number));
19          numbers.forEach((number) -> System.out.println(number));
20          numbers.forEach(number -> System.out.println(number));
21          numbers.forEach(System.out::println);
22      }
```

Lambdas - Examples

Lambdas2.java

```
    public List<Integer> streamExample(final List<String> list) {
        return list.stream()
                .map(Integer::parseInt)
                .filter(i -> i > 10)
                .map(i -> i * 10)
                .collect(Collectors.toList());
    }

    public Optional<String> optionalExample(final String in) {
        return Optional.ofNullable(in)
                .filter(s -> !s.isEmpty())
                .map(s -> "Value: " + s);
    }
}
```

Default methods

Default methods

Interfaces can have method implementations

Default methods

MyInterface v1.0

- abstract a()
- abstract b()

MyClass implements MyInterface

Default methods

MyInterface v1.0

- abstract a()
- abstract b()

MyInterface v2.0

- abstract a()
- abstract b()
- abstract c()

MyClass implements MyInterface

Default methods

MyInterface v1.0

- abstract a()
- abstract b()

MyInterface v2.0a

- abstract a()
- abstract b()
- ~~abstract c()~~

MyInterface v2.0b

- abstract a()
- abstract b()
- default c() {
  /* Implementation. */
  }

Default methods

DefaultMethodExample.java

```
1  package nl.jcore.java8demo.defaultmethods;
2
3  import sun.reflect.generics.reflectiveObjects.NotImplementedException;
4
5  public interface DefaultMethodExample {
6      void demonstrate();
7
8      default void demonstrateDefault() {
9          final String assignmentVar = "like any other";
10         String.format("A method implementation, %s.", assignmentVar);
11     }
12
13     default String demonstrateDefaultB() {
14         throw new NotImplementedException();
15     }
16 }
```

Method parameter reflection

Method parameter reflection

Accessing a method’s parameters’ names

Method parameter reflection

Reflection!

ParameterNamesExample.java

```
6   private class Test {
7       public void entryMethod(final String testArg1, final String testArg2) { }
8   }
9
10  private void getMethodInfo() throws NoSuchMethodException {
11      final Method method = Test.class.getDeclaredMethods()[0];
12      final int numParameters = method.getParameterCount();
13      final String arg0Name = method.getParameters()[0].getName();
14      System.out.println(numParameters); // Prints "2"
15      System.out.println(arg0Name); // Prints "arg0"
16  }
```

Method parameter reflection

```
C:\Users\Lennert>javac *.java -parameters
```

Compile code (javac) with -parameter flag
→ Allow e.g. Jackson or Spring to get parameter name at runtime

ParameterNamesController.java

```
7   @RestController
8   @RequestMapping("/api/1/demo/parameter-reflection/requestbody")
9   public class ParameterNamesController {
10      @RequestMapping
11      public void demoOld(@RequestParam("name") final String name) { }
12
13      @RequestMapping
14      public void demoNew(@RequestParam final String name) { }
15  }
16
```

Method parameter reflection

ParameterNamesDto.java

```
8   private class ParameterReflectionDto {
9       private final int id;
10      private final String name;
11
12      @JsonCreator
13      public ParameterReflectionDto(final int id, final String name) {
14          this.id = id;
15          this.name = name;
16      }
17
18      public int getId() {
19          return id;
20      }
21
22      public String getName() {
23          return name;
24      }
25  }
```

ParameterNamesDto.java

```
27  @RestController
28  @RequestMapping("/api/1/demo/parameter-names/dto")
29  public class ParameterReflectionController {
30      @RequestMapping
31      public void demo(final ParameterReflectionDto dto) {
32      }
33  }
```

Streaming API

Streaming API

"[Streams are] an abstraction for expressing efficient, SQL-like operations on a collection of data."[4]

[4] http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

Streaming API

From a list of users, get the first 10 names of users with age of 18 or higher

Streaming API

StreamingExample.java

StreamingExample | java8Streaming()

```
57      public List<String> java7StreamingEquivalent() {
58          final List<String> result = new ArrayList<>();
59          for (final User user : getUsers()) {
60              if (user.age >= 18) {
61                  result.add(user.getName());
62              }
63              if (result.size() >= 10) {
64                  break;
65              }
66          }
67          return result;
68      }
```

Streaming API

Input: list of users
→ Filter out users below age 18
→ Transform full users to names
→ Limit to 10

Streaming API

StreamingExample.java ×

StreamingExample | java8Streaming()

```
89      public List<String> java8Streaming() {
90          return getUsers().stream()
91                  .filter(u -> u.getAge() >= 18)
92                  .map(User::getName)
93                  .limit(10)
94                  .collect(Collectors.toList());
95      }
```

Streaming API

From a list of users, get the names and last transaction of the 10 users that last performed a transaction and are of age 18 or higher.

Streaming API

Input: list of users
→ Filter out users below age 18
→ Filter out users that have not performed any transaction
→ Sort by last transaction timestamp
→ Transform full users to names with last transaction
→ Limit to 10

Streaming API

StreamingExample.java

StreamingExample > java8Streaming()

```
70      public Map<String, Transaction> java7StreamingMapEquivalent() {
71          final Map<Long, User> sortedUsers = new TreeMap<>();
72          for (final User user : getUsers()) {
73              if (user.age >= 18 && user.hasTransactions()) {
74                  sortedUsers.put(getLastTransaction(user).getTimestamp().toEpochMilli(), user);
75              }
76          }
77          final Map<String, Transaction> result = new HashMap<>();
78          final ArrayList<Long> keys = new ArrayList<>(sortedUsers.keySet());
79          for (int i = keys.size() - 1; i >= 0; i--) {
80              final User user = sortedUsers.get(i);
81              result.put(user.getName(), getLastTransaction(user));
82              if (result.size() >= 10) {
83                  break;
84              }
85          }
86          return result;
87      }
```

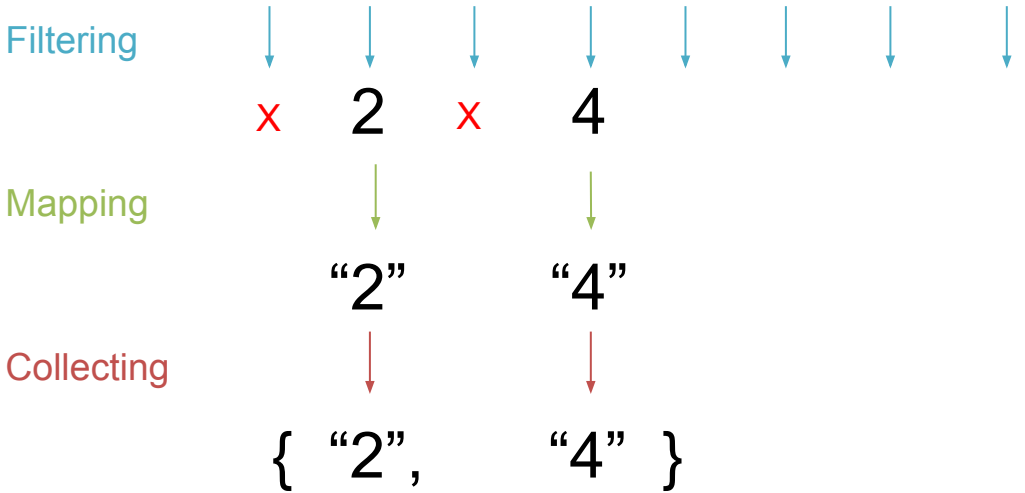
Streaming API

StreamingExample.java

StreamingExample > java8Streaming()

```
97      public Map<String, Transaction> java8StreamingMap() {
98          final Comparator<User> compareLastTransactionTimestamp =
99                  Comparator.comparingLong(u -> getLastTransaction(u).getTimestamp().toEpochMilli());
100         return getUsers().stream()
101                 .filter(u -> u.getAge() >= 18)
102                 .filter(User::hasTransactions)
103                 .sorted(compareLastTransactionTimestamp)
104                 .limit(10)
105                 .collect(Collectors.toMap(User::getName, this::getLastTransaction));
106     }
```

List<Integer> numbers = { 1, 2, 3, 4, 5, 6, 7, 8 };




```
numbers.stream()
    .filter(i -> i % 2 == 0)
    .map(String::valueOf)
    .limit(2)
    .collect(Collectors.toList());
```

Streaming

Declarative

Highly efficient

Parallelizable

Optional

Optional

"A container object which may or may not contain a non-null value."[5]

[5] https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html

Optional

- Traditionally *null* is used
- Nullpointer exceptions
- Holder for optional values

- C++: Optional
- C#: ?T
- Haskell: *Maybe*
- Kotlin: T?
- Scala: Option
- Swift: Optional

Optional

OptionalExample.java

```java
@RestController
@RequestMapping("/api/1/demo/optional")
public class OptionalExample {
    @RequestMapping
    public void searchTransactions(final Optional<String> terms, final Optional<Instant> fromTime) {
        final Instant reasonableDefaultFromTime = Instant.now().minus(1, ChronoUnit.DAYS);
        search(terms, fromTime.orElse(reasonableDefaultFromTime));
    }

    private void search(final Optional<String> terms, final Instant fromTime) {
    }
}
```

Optional

Initialization:

- Optional.of(valueThatMayNotBeNull);
- Optional.ofNullable(valueThatMaybeNull);

Optional

OptionalExample2.java

```
6       public void doSomethingIfPresent(final Optional<String> optString) {
7           if (optString.isPresent()) {
8               doSomethingWithString(optString.get());
9           }
10      }
11
12      private void doSomethingWithString(final String str) {
13      }
14  }
```

Optional

OptionalExample3.java

```
6       public void doSomethingIfPresent(final Optional<String> optString) {
7           optString.ifPresent(this::doSomethingWithString);
8
9           optString
10                  .map(String::trim)
11                  .filter(s -> !s.isEmpty())
12                  .ifPresent(this::doSomethingWithString);
13      }
14
15      private void doSomethingWithString(final String str) {
16      }
17  }
```

Optional

- Repository
- Streaming
- Controller

Alternatives

- @NonNull
- @Nullable

Time

Time

Java 7 java.util.Date:

- Not thread-safe
- Years start at 1900, months start at 1; days start at 0
- Poor concepts

Time

Java 8 java.time.*:

- Immutable values
- Domain Driven Design
- Enhanced support for non-ISO calendar systems

Time - Domain Driven Design

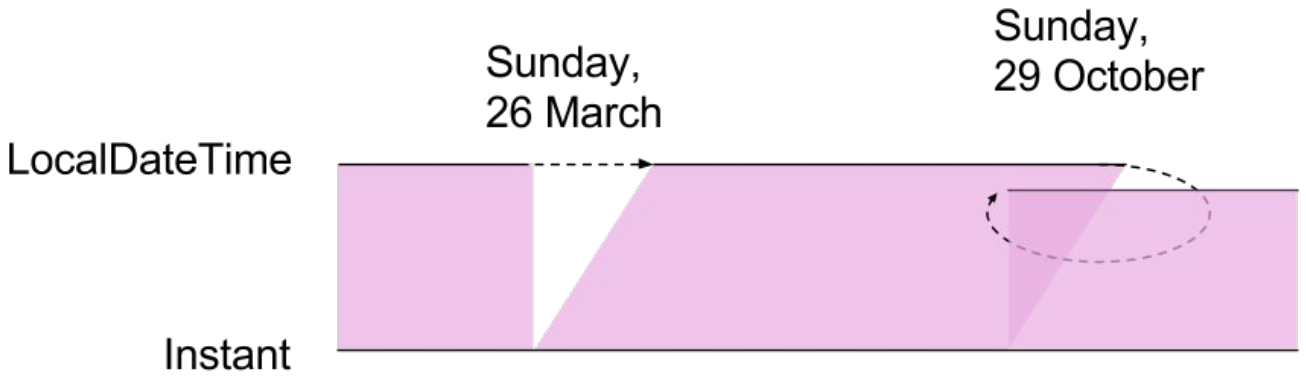
- LocalDate, LocalTime, LocalDateTime
- ZonedDateTime
- Period, Duration
- Instant
- Other classes for non-ISO calendaring systems

Time - Instant

"An instantaneous point on the time-line."[6]

[6] https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html

Time - Instant



Conclusion

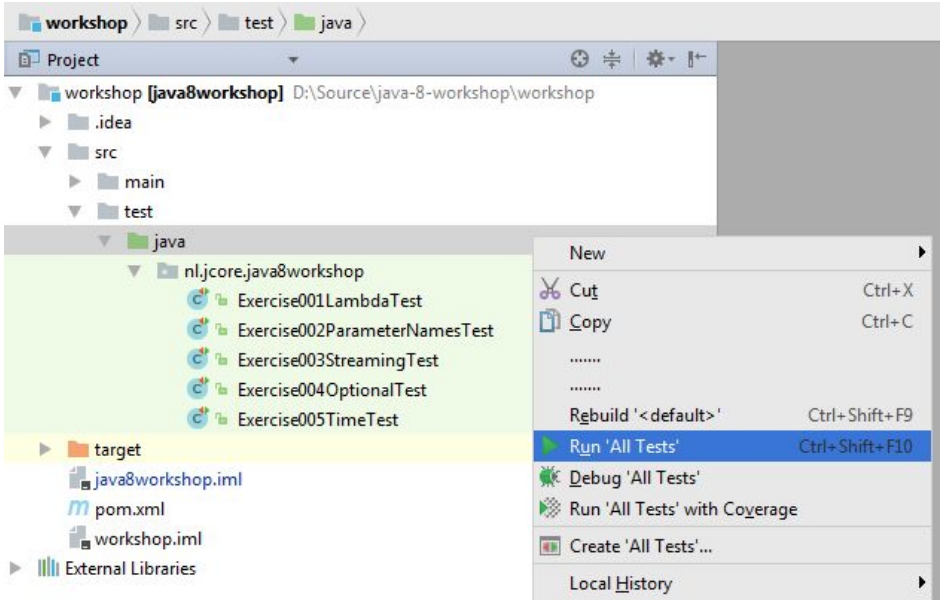
- Language
  - Lambdas
  - Default methods
  - Method parameter reflection
- APIs
  - Streaming
  - Optional
  - Time

Extended reading

Java 8 migration guide (compatibility information, removed features and APIs etc.)
Lambdas - A peek under the hood
Optional Method Parameters

Workshop

git clone https://github.com/LDMGN/java-8-workshop.git



- Open Maven project
  java-8-workshop/workshop
  with your IDE
- Run unit tests

Tip: *SHIFT + F10* to re-run last job (tests)

Lambda - Ambiguity

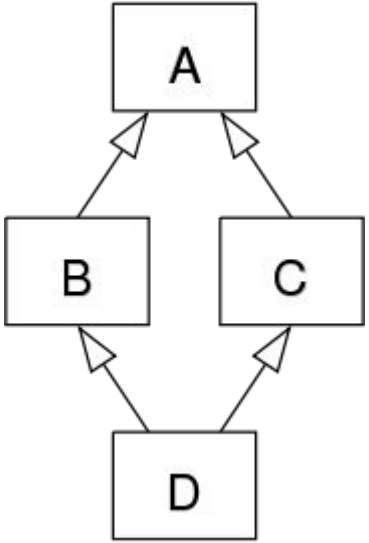
```
22      /**
23       * @return function that converts it's Integer input to a String
24       */
25      public static Function<Integer, String> functionToString() {
26          return Integer::toString;
27      }
28  }
29
```

Incompatible types.
Required: Function <Integer, String>
Found: Function <Integer, String>

→ Integer.toString(i)
→ (new Integer(i)).toString();

```
22      /**
23       * @return function that converts it's Integer input to a String
24       */
25      public static Function<Integer, String> functionToString() {
26          return String::valueOf;
27      }
28  }
```

Default methods - Diamond problem



DefaultExample.java

```
1  package nl.jcore.java8demo.defaultmethods;
2
3  interface A {
4      void test();
5  }
6
7  interface B extends A {
8      default void test() {
9          System.out.println("B");
10     }
11 }
12
13 interface C extends A {
14     default void test() {
15         System.out.println("C");
16     }
17 }
18
19 class D implements B, C {
20 }
21
22
23 class E implements B, C {
24     public void test() {
25         B.super.test();
26     }
27 }
28
```

D inherits unrelated defaults for test() from types B and C

Type inferencing

Java 6:

```
final Map<String, String> map = new HashMap<String, String>();
```

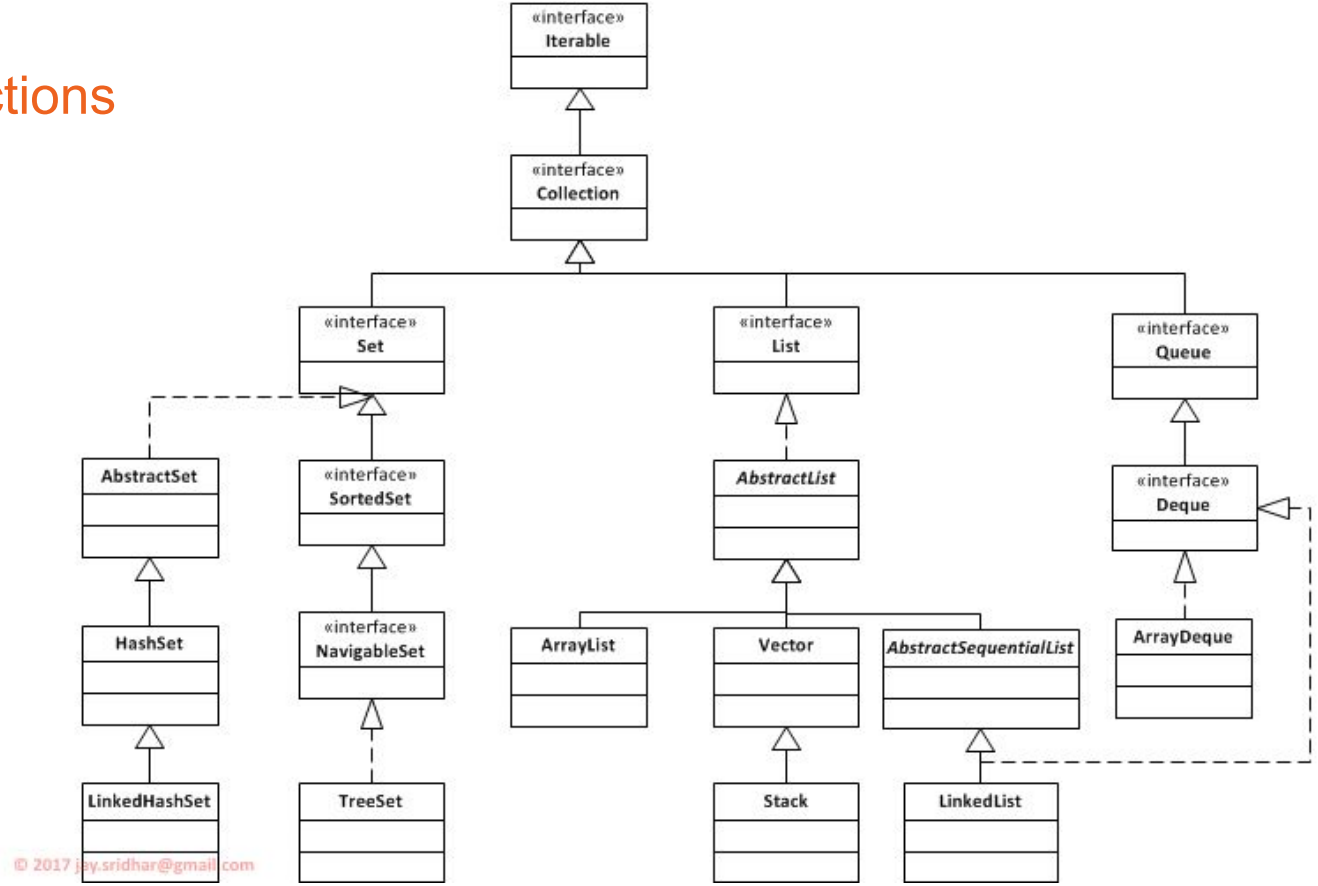
Java 7:

```
final Map<String, String> map = new HashMap<>();
final ImmutableMap<String, String> map = ImmutableMap<String, String>.of();
```

Java 8:

```
final ImmutableMap<String, String> map = ImmutableMap.of();
```

Collections



© 2017 jay.sridhar@gmail.com

Java 9

- Tooling
  - JShell (Read, Eval, Print Loop)
  - Microbenchmarking
- Language
  - Modular JDK / Jigsaw
  - Default garbage collector
- APIs
  - Process API
  - Enhanced deprecation
  - Improved logging API
  - HTTP 2.0

https://www.jcp.org/en/jsr/detail?id=379