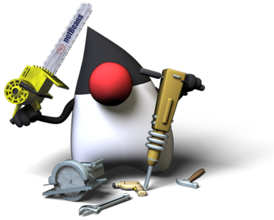
What's new in Java 8?



© 2018 Robert Monnet

Improvements in Java 8

Lambda Expressions

- Extensible Interfaces
- Functional Interfaces
- Lambda Expressions
- Method References

The Stream API

Java library new features

- Optional Class
- New Date Time API

Nashorn

Extensible Interfaces

- Adding a method to an interface breaks backward compatibility.
- To fix this problem, Java 8 is adding default and static methods to Interfaces.

Example of Interface Extension before Java 8

```
// Need to add method addAll() to ICollection Interface,
// must create new Interface to avoid breaking existing classes.
public interface ICollection<E> {
    void add(E element);
    ...
}

public interface ICollection2<E> extends ICollection<E> {
    // new method added to the Interface.
    void addAll(E... elements);
    ...
}

// Existing classes do not benefit from new method
// or must be updated to point to new interface.
public class List<E> implements ICollection<E> {
    ...
}
```

Extensible Interfaces

Example of Interface Extension with Java 8

```java
// Need to add method addAll() to ICollection Interface,
// add a default method to preserve backward compatibility.
public interface ICollection<E> {
    void add(E element);

    // new method with default implementation,
    // can be overridden in subclasses.
    default void addAll(E... elements) {
        for (E element : elements) {
            add(element);
        }
    }
    ...
}

// Existing classes benefit from new method
// and stay backward compatible with new interface.
public class List<E> implements ICollection<E> {
    // no change required
    // automatically picks up the new addAll() method.
    ...
}
```

Functional Interface

- A Functional Interface is any interface that defines a single method (non-including public methods on Object, default and static methods)
- Also called a SAM (Single Abstract Method)
- Optional `@FunctionalInterface` helps the compiler enforce the Single Abstract Method

Example of Functional Interface

```java
// @FunctionalInterface annotation is optional but let the
// compiler checks that the interface is truly a SAM.
@FunctionalInterface
public interface ICalculator {

    // single abstract method
    int add(int a, int b);

    // default method, doesn't count as abstract method
    default int sub(int a, int b) {
        return add(a, -1 * b);
    }

    // method from Object class, doesn't count as abstract method
    @Override
    String toString();

}
```

Lambda Expression

- A Lambda Expression is a short-hand notation to implement a Functional Interface
- Types are optional and inferred from the Functional Interface
- Cannot use a Lambda Expression in places that do not expect a Functional Interface

Before Java 8 (Anonymous Class)

```java
public static int doSomeCalculations(ICalculator calc) {
    ...
}
...
// In Java 7, we need to define an anonymous class to specify the SAM.
int res = doSomeCalculations(new ICalculator() {
    @Override
    public int add(int a, int b) {
        return a + b;
    }
});
...
```

Lambda Expression

- Syntax for Lambda is
  - `(params) -> expression` (implicitly returns the value of the expression)
  - `(params) -> { expression, ..., [return expression] }`

With Java 8 (Lambda Expression)

```
public static int doSomeCalculations(ICalculator calc) {
    ...
}
...
// In Java 8, Lambda Expression replaces the
// anonymous class.
int res = doSomeCalculations((a, b) -> a + b);
...
```

Useful Predefined Functional Interfaces

- `java.util.function`
- many specialized variants available for different types, number of parameters, …

Sample of Predefined Functional Interfaces

Usage	Interface	SAM
Single Argument Function	`Function<T, R>`	`{ R apply(T t); }`
Two Arguments Function	`BiFunction<T,U,R>`	`{ R apply(T t, U u); }`
Unary Operator	`UnaryOperator<T>`	`{ T apply(T t); }`
Binary Operator	`BinaryOperator<T>`	`{ T apply(T t, T u); }`
Boolean Predictate	`Predicate<T>`	`{ boolean test(T t); }`
Consumes a Single Argument	`Consumer<T>`	`{ void accept(T t); }`
Produces a Single Result	`Supplier<T>`	`{ T get(); }`

Method References

- We can simplify Lambda Expression further using Method References
- a Lambda Expression can be replaced by a Method Reference if the Method Reference has the same signature as the SAM
- syntax for method reference: `class::method` or `object::method`

Static Method used as Method Reference

```
// Example of method reference using a Static Method
@FunctionalInterface
public interface IStringFormatter {
    String format(String delimiter, List<String> list);
}
...
public static String formatString(IStringFormatter fmt) {
    ...
}
...
// Normal Lambda Expression
System.out.println(formatString((delim, list) -> String.join(delim, list)));

// Short-hand using Static Method Reference
System.out.println(formatString(String::join));
...
```

Method References

- Method Reference can also use an Object Method Reference
- Useful when the Object is not changing

Object Method used as Method Reference

```
// Using a standard library Functional Interface:
//  Consumer<T> { void accept(T t) }

public static void processList(Consumer<String> proc) {
    List<String> list = Arrays.asList("the", "brown", "fox");
    for (String name : list) {
        proc.accept(name);
    }
}

// Normal Lambda Expression
processList((element) -> System.out.println(element));

// Short-hand using an Object Method Reference
processList(System.out::println);
...
```

Method References

- Method Reference can also use an Object Method Reference when the object is not available
- Same syntax as Static Method Reference
- The first parameter of the Functional Interface is used as the target Object
- Useful when the Object changes during processing

Object Method used as a Method Reference (without an object)

```
// Using a Standard Library Functional Interface:
public Interface Comparable<T> {
    int compareTo(T 0);
    ...
}

String[] names = { "the", "brown", "fox" };

// Using a Normal Lambda Expression
Arrays.sort(names, (s1, s2) -> s1.compareTo(s2));
print(names);

// Short-hand using an Object Method Reference
// Reference Method using Class::Method, 1st arg. on compareTo is used as target
Arrays.sort(names, String::compareTo);
print(names);
```

Method References

- Last Type of Method Reference is a Constructor Reference
- syntax is `Class:new`

```
// Using a Standard Library Functional Interface:
// Supplier<T> { T get() }

public static <T> Collection<T> toCollection(
                Supplier<Collection<T>> constructor, T[] array) {
    // supplier is used to construct the collection
    Collection<T> res = constructor.get();
    for (T element : array) {
        res.add(element);
    }
    return res;
}
...
String[] names = { "the", "brown", "fox" };

// Using a Normal Lambda Expression
printCollection(toCollection(() -> new LinkedList<String>(), names));

// Using a Constructor Reference
printCollection(toCollection(LinkedList::new, names));
...
```

Stream API

> *We should have some ways of coupling programs like garden hose – screw in another segment when it becomes necessary to massage data in another way. This is the way of IO also.*
>
> `Doug McIlroy, inventor of Unix pipes`

- A Java `Stream` is an output or input sequence of objects. Operations can be used to generate, transform or consume streams.
- Streams are not collections.
- Java `Stream` allows programming in the "functional style".

Stream API

- `java.util.stream`
- Main interface is `Stream`

A Sample of the predefined Stream Operations (Generators)

Usage	Method
Generates a stream from the list of parameters	`Stream.of(cs)`
Generates an empty stream	`Stream.empty()`
Generates an infinite sequence by repeatedly calling the `Supplier s`	`Stream.generate(s)`
Creates a new stream by lazily concatenating the 2 streams	`Stream.concat(...)`

A Sample of the predefined Stream Operations (Transformers)

Usage	Method
Returns a new stream with all the elements for which the `Predicate p` is true	`filter(p)`
Returns a new stream by applying the `Function f` to all the elements of the stream	`map(f)`
Returns a new stream with the elements sorted according to the `Comparator c`	`sorted(c)`

Stream API

A Sample of the predefined Stream Operations (Consumers)

Usage	Methods
Performs a reduction on the elements of the stream using the `BinaryOperator op` and `id` as the initial value	`reduce(id, op)`
Aggregates the stream elements into a collection according to the `Collector c`	`collect(c)`
Returns true if the `Predicate p` returns true for all the elements	`allMatch(p)`
Returns true if the `Predicate p` returns true for any of the elements	`anyMatch(p)`
Returns an `Optional` containing the maximum element in the stream according to the `Comparator c`	`max(c)`
alls the `Consumer c` on all the elements of the stream	`forEach(c)`

- any many more

Stream API

Example of stream use with Java 8

```java
// building streams
public static Stream<String> names() {
    Stream<String> names1 = Stream.of("the", "brown", "fox");
    Stream<String> names2 = Stream.of("jumps", "over", "the", "moon");
    return Stream.concat(names1, names2);
}

// operating on streams

boolean allNamesLongerThan3Char = names().allMatch((n) -> n.length() > 3);
// >>> false

int maxNameLength = names().map(String::length).max(Integer::compareTo).orElse(0);
// >>> 5

String upperCaseNameList = names().map(String::toUpperCase).collect(Collectors.joining(", "));
/// >>> THE, BROWN, FOX, JUMPS, OVER, THE, MOON

String aFewFibNumbers =
    Stream.iterate(new Pair<Integer,Integer>(0, 1), (p) -> new Pair(p.right(), p.left() + p.right()))
        .map((p) -> p.right().toString())
        .limit(20).collect(Collectors.joining(", "));
// >>> 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

Stream API (More exotic use)

- Not the intended primary usage of the Stream API but fun (trying to catch up to Haskell)

In Haskell

```
primes :: [Int]
primes = sieve [2..]
sieve (x:xs) = x : sieve [y | y <- xs, y `mod` x > 0]
```

In Java 8

```
public class EratosteneSieve {

    private Predicate<Integer> isPrime = x -> true;
    // Hack, expand the predicate as we find new primes to filter all their multiples
    private Stream<Integer> primes = Stream.iterate(2, i -> i + 1).filter(i -> isPrime.test(i))
            .peek(i -> isPrime = isPrime.and(v -> v % i != 0));
    ...
    int thousandthPrime = new EratosteneSieve().getPrimes().skip(999).findFirst().get();
     ...
```

Stream Processing Performance

- Java 8 provides an easy mechanism to parallelize Stream processing
- `Collection.parallelStream()` or `Stream.parallel()`
- Parallel is not always faster, check the performance
- Actually stream may not be faster than classic iteration (see this DZone article)

```
final int N = 1000;
Instant t0 = Instant.now();
Random rand = new Random();
double sumSqrt = rand.doubles().map(Math::sqrt).limit(N).sum();
Instant t1 = Instant.now();
rand = new Random();
double sumSqrtPar = rand.doubles().parallel().map(Math::sqrt).limit(N).sum();
Instant t2 = Instant.now();
System.out.println(
        "serial   sum sqrt of " + N + " random numbers in " + Duration.between(t0, t1).toMillis() + "ms)");
System.out.println(
        "parallel sum sqrt of " + N + " random numbers in " + Duration.between(t1, t2).toMillis() + "ms)");
```

```
sum sqrt of 1000 random numbers:    serial =  64ms, parallel =   9ms
sum sqrt of 1000000 random numbers: serial = 101ms, parallel = 284ms
```

Optional Class

- The `Optional` class provides protection against null values

Dealing with Null Pointers in Java 7

```java
// Class with fields that can take a null value
public class Company {
    private Address address;
    public Address getAddress() {
        return address;
    }
}
// Containing another class with fields that can take a null value
public class Address {
    private String city;
    public String getCity() {
        return city;
}
// And so on ...
// How can we safely retrieve values from the object tree?
Company easyCompany = new Company();
String easyCompanyCity = null;
if (easyCompany.getAddress() != null
        && easyCompany.getAddress().getStreet() != null) {
    easyCompanyCity = easyCompany.getAddress().getCity();
}
```

Optional Class

- `Optional` wraps the nullable object into an `Optional` type

Sample of Optional Methods

Usage	Method
Create an empty field	`Optional.empty()`
Create a field for a non-null value	`Optional.of(v)`
Create a field for a potentially null value	`Optional.ofNullable(v)`
return the value contained in the `Optional` or `other`	`orElse(other)`
return true if the `Optional` is not empty	`isPresent()`
call the consumer on the content if the `Optional` is not empty	`ifPresent(c)`
return empty `Optional` or apply the function `f` to the content if present	`flatMap(f)`

Optional Class

Dealing with Null Pointers using Optional in Java 8

```
// Class with fields that can take a null value
public class Company {
    private Optional<Address> address = Optional.empty();
    public void setAddress(Address address) {
        this.address = Optional.ofNullable(address);
    }
}
// Containing another class with fields that can take a null value
public class Address {
    private Optional<String> city = Optional.empty();
    public void setCity(string city) {
        this.city = Optional.ofNullable(city);
}
// Now we don't have to worry about NullPointerException
Company easyCompany = new Company();
String easyCompanyCity =
    easyCompany.getAddress().flatMap(Address::getCity).orElse("unknown");
}
```

Optional Class

- But there is still potential for `NullPointerException`
- Make sure you initialize your `Optional` fields to `empty`

```
// Class with fields that can take a null value
public class Company {
    // Oops! forgot to initialize the address field
    private Optional<Address> address;
...
}
// Now we think we don't have to worry about NullPointerException
// but we still get one
Company easyCompany = new Company();
String easyCompanyCity =
    easyCompany.getAddress().flatMap(Address::getCity).orElse("unknown");
}
```

```
>>> Exception in thread "main" java.lang.NullPointerException
    at CompanyExample.main(CompanyExample.java:23)
```

Date Time Classes

- Java 8 has new Date and Time classes inspired by the Joda-Time API
- `java.time`

Main Date Time Classes

Usage	Class
Represents a point on the time-line based on the Unix Standard Epoch. Useful for timing events	`Instant`
Models a Date without Time component anchored in the local Time Zone	`LocalDate`
Models a Time anchored in the local Time Zone	`LocalTime`
Represents the amount of Time between two Dates	`Period`
Models a Date with a Time component anchored in the local Time Zone	`LocalDateTime`
Models a Date with a Time component associated with any Time Zone	`ZonedDateTime`
Represents the different Time Zones	`ZoneId`
Represents the amount of Time between two DateTime objects accounting for Time Zones	`Duration`

Date Time formatting and parsing

- To format a `Date`, `Time` or `DateTime` object:
  - use `java.time.format.DateTimeFormatter`
  - `format(DateTimeFormatter format)` method on any of the `Date`, `Time` or `DateTime` object

Example of Date Time formatting in Java 8

```
ZonedDateTime time = ZonedDateTime.of(2018, 3, 9, 9, 31, 20, 0, ZoneId.of("Z"));

// can specify the exact format
DateTimeFormatter zuluFormat =
        DateTimeFormatter.ofPattern("MM/dd/yyyy HH:mm:ss X");
String zuluTime = time.format(zuluFormat);
// 03/09/2018 09:31:20 Z

// or use one of the predefined format
String isoTime = time.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME);
// 2018-03-09T09:31:20
```

References

- Java 8 New Features
  - Java Stream Tutorial

Attributions

- Duke's image is from Wikimedia "Duke: Java Mascot".
- This presentation is using the excellent remark framework.