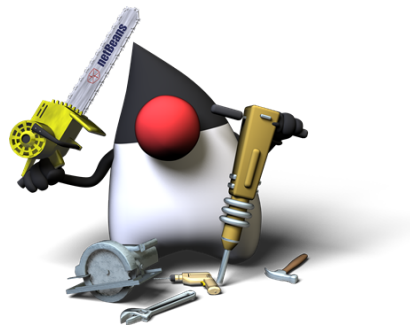
What's new in Java 8?



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

```
// 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

```
// @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
- Parameter 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)

```
    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

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

Sample of Predefined Functional Interfaces

Interface	Usage	Single Abstract Method
`Function<T, R>`	Single Argument Function	`R apply(T t)`
`BiFunction<T,U,R>`	Two Arguments Function	`R apply(T t, U u)`
`UnaryOperator<T>`	Unary Operator	`T apply(T t)`
`BinaryOperator<T>`	Binary Operator	`T apply(T t, T u)`
`Predicate<T>`	Boolean Predictate	`boolean test(T t)`
`Consumer<T>`	Consumes a Single Argument	`void accept(T t)`
`Supplier<T>`	Produces a Single Result	`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 Single Abstract Method
- 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 can't be referenced directly
- 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 (think unix pipes)
- Streams are not collections
- Java `Stream` allows programming in the "functional style"

Stream API

- Package `java.util.stream`

A Sample of the predefined Stream Operations (Generators)

Method	Usage
`Stream.of(cs)`	Generates a stream from the list of parameters
`Stream.empty()`	Generates an empty stream
`Stream.generate(s)`	Generates an infinite sequence by repeatedly calling the `Supplier s`
`Stream.iterate(s, f)`	Generates an infinite sequence by repetitively applying the function `f`, starting with `s` as seed
`Stream.concat(s1,s2)`	Creates a new stream by lazily concatenating the 2 streams
`Collection.stream()`	Creates a stream over the elements in the `Collection`
`Collection.parallelStream()`	Create a stream processed in parallel over the elements in the `Collection`

Stream API

A Sample of the predefined Stream Operations (Transformers)

Method	Usage
`filter(p)`	Returns a new stream with all the elements for which the `Predicate p` is true
`map(f)`	Returns a new stream by applying the `Function f` to all the elements of the stream
`flatMap(f)`	Returns a new stream replacing each element of the stream with the content of the stream produced by applying the `Function f`
`limit(n)`	Returns a new stream consisting of no more than the first `n` elements of the stream
`skip(n)`	Returns a new stream consisting of the elements of the stream after skipping the first `n` elements
`sorted(c)`	Returns a new stream with the elements sorted according to the `Comparator c`
`distinct()`	Returns a new stream with only the distinct elements of the stream

Stream API

A Sample of the predefined Stream Operations (Consumers)

Methods	Usage
`count()`	Return the number of elements in the stream
`reduce(id,op)`	Reduces the elements of the stream using the `BinaryOperator op` and the initial value `id`
`collect(c)`	Aggregates the stream elements into a collection according to the `Collector c`
`allMatch(p)`	Returns true if the `Predicate p` returns true for all the elements
`anyMatch(p)`	Returns true if the `Predicate p` returns true for any of the elements
`noneMatch(p)`	Returns true if the `Predicate p` return false for all the elements
`max(c)`	Returns an `Optional` containing the maximum element according to the `Comparator c`
`min(c)`	Returns an `Optional` containing the minimum element according to the `Comparator c`
`forEach(c)`	Calls the `Consumer c` on all the elements of the stream

Stream API

Example of stream use with Java 8

```java
// building streams
public static Stream<String> names() {
    Stream<String> names1 = Stream.of("the", "quick", "brown", "fox");
    Stream<String> names2 = Stream.of("jumps", "over", "the", "lazy", "dog");
    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, QUICK, BROWN, FOX, JUMPS, OVER, THE, LAZY, DOG

String aFewFibonacciNumbers =
    Stream.iterate(Pair.of(0, 1), (p) -> Pair.of(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 (An example of more exotic use)

- Not the intended primary usage of the Stream API but fun (trying to catch up to Haskell)
- Warning: Computing large primes using this Java `Stream` expression will result in `StackOverflowError` because Java doesn't have tail recursion.

In Haskell

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

In Java 8

```java
public class EratosteneSieve {

    // Hackish, uses peek() to redefine isPrime everytime a prime is found to remove its multiples from the
    // remaining list of integers. This will, ultimately overflow the Java stack
    private Predicate<Integer> isPrime = x -> true;

    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
```

Exceptions in Stream Processing

- Checked Exceptions can't be thrown from inside a Stream manipulation function. (they need to be handled or replaced with an Unchecked Exception)
- When an Exception is thrown from within a stream pipeline, the stack trace can be confusing

```
java.util.List<String> names = Arrays.asList("the", "brown", "fox", "jumps", "over", "", "the", "lazy", "dog");
int sentenceLength = names.stream()
    .mapToInt((name) -> {
        if (name.equals(""))
            // Couldn't throw a Checked Exception here, it would result in a compile error
            throw new RuntimeException("name can't be empty");
        return name.length();})
    .sum();
System.out.println(sentenceLength);
```

```
Exception in thread "main" java.lang.RuntimeException: name can't be empty
    at StreamExample2.lambda$0(StreamExample2.java:17)
    at java.util.stream.ReferencePipeline$4$1.accept(ReferencePipeline.java:210)
    at java.util.Spliterators$ArraySpliterator.forEachRemaining(Spliterators.java:948)
    at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:481)
    at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:471)
    at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:708)
    at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)
    at java.util.stream.IntPipeline.reduce(IntPipeline.java:456)
    at java.util.stream.IntPipeline.sum(IntPipeline.java:414)
    at StreamExample2.main(StreamExample2.java:21)
```

Optional Class

- The Class `Optional` provides protection against null values
- `Optional` wraps the nullable object into an `Optional` type

Sample of Optional Methods

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

Optional Class

Dealing with Null Pointers before Java 8

```
// 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().getCity() != null) {
    easyCompanyCity = easyCompany.getAddress().getCity();
}
```

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`
- Consider using sensible "zero" values instead (see Dave Cheney's article on zero values)

```
// Class with fields that can take a null value
public class Company {
    private Optional<Address> address; // Oops! forgot to initialize the address field
...
}
// 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
- Package `java.time`

Main Date Time Classes

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

Date Time formatting and parsing

- To format a `Date`, `Time` or `DateTime` object:
  - use Class `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, 10, 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 10:31:20 Z

// or use one of the predefined format
String isoTime = time.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME);
// 2018-03-09T10:31:20
```

Nashorn

- Nashorn is the new javascript interpreter in Java 8 (replacing Rhino)
- You can call Javascript from Java and Java from Javascript
- Nashorn let you execute shell scripts commands from Javascript on unix machines

```
// get the Nashorn javascript engine
ScriptEngineManager factory = new ScriptEngineManager();
ScriptEngine js = factory.getEngineByName("nashorn");
// store data into the javascript environment
js.put("name", "Nashorn");
try {
    // evaluate a javascript script (string or file input)
    js.eval("print('hello from ' + name); res=42;" + "wd1 = new java.io.File(\".\").getCanonicalPath();");
    // retrieve data from the javascript environment
    int res = (int) js.get("res");
    String wd = (String) js.get("wd");
    System.out.println("result of script is " + res);
    System.out.println("working directory is " + wd);
} catch (ScriptException e) {
    e.printStackTrace();
}
```

```
>>> hello from Nashorn
>>> result of script is 42
>>> working directory is /Users/robert/Documents/eclipse-ws/java8-new-features
```

That's all Folks!

- Use Java 8 new features wisely!

  Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

  `Brian Kernighan, creator of the AWK language`

References

- Java 8 Tutorial
- Extensible Interfaces
- Lambda Expressions
- Extensible Interfaces
- Lambda Expressions - Dr Dobbs
- Lambda and Streams in the Java 8 Library
- Java Stream Tutorial
- Java 8 Stream performance
- Handling Checked Exceptions in Java Stream

Attributions

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