Zaawansowane programowanie obiektowe

Lecture 3
(incl. language and lib. changes in Java 8+)

Szymon Grabowski

sgrabow@kis.p.lodz.pl

http://szgrabowski.kis.p.lodz.pl/zpo18/

Łódź, 2018

Enumerated types (enums)

An enumerated type: a user-defined type which explicitly enumerates all the possible values for a variable of this type.

Example (C++):

```
enum Day {SUNDAY, MONDAY, TUESDAY,
WEDNESDAY, THURSDAY, FRIDAY, SATURDAY};
```

Enum benefits: type safety, code clarity, compact declaration, no run-time performance penalty.

Enums in Java (5.0+)...

...are better than in e.g. C++
(but note: in C++11 enums are strongly typed).

Basic use and syntax is alike:
enum Season { WINTER, SPRING, SUMMER, FALL }

The Java *enum* is a special type of class (an *enum type*). It allows you to add arbitrary methods and fields, to implement arbitrary interfaces, and more. Enum types provide high-quality implementations of all the Object methods (*equals*, *hashCode* etc.) and *compareTo* from Comparable<E>.

There are some automatically created methods when an enum is created: values(), ordinal() and more.

```
for(Season s: Season.values())
   System.out.println("season no " + (s.ordinal() + 1) + " is " + s.name();
```


3

More on enums

All enums implicitly subclass the abstract java.lang.Enum.

It implements the Comparable and Serializable interfaces.

Java introduces two new collections, EnumSet and EnumMap, which are only meant to optimize the performance of sets and maps when using enums.
(You could use a more general HashMap with enums too.
But it may be not so efficient.)

Enums (i.e., subclasses of java.lang.Enum)
are (implicitly) final (unless the enum contains at least one enum constant that has a class body).

See also:
http://stackoverflow.com/questions/22074497/java-enum-tostring-method/22075272#22075272

4

Enum with fields and methods, example

[W. Clay Richardson et al., *Professional Java, JDK 5 Edition*, Wrox, 2005]

```
enum ProgramFlags {
  showErrors(0x01),  // set by the constructor
  includeFileOutput(0x02),   // set by the ctor
  useAlternateProcessor(0x04);   // set by the ctor

  private int bit;
  ProgramFlags(int bitNumber) { bit = bitNumber; }  // ctor, can't be public
  public int getBitNumber() { return(bit); }
}

public class EnumBitmapExample {
  public static void main(String[] args) {
      ProgramFlags flag = ProgramFlags.showErrors;
         System.out.println("Flag selected is: " +
         flag.ordinal() +  " which is " + flag.name() );
   }
}  // ordinal() returns the pos. of the constant in the list (from 0)
```

Abstract method in an enum

```
public enum Currency implements Runnable{
          PENNY(1) {
                    @Override
                    public String color() {
                              return "copper";
                    }
          }, NICKLE(5) {
                    @Override
                    public String color() {
                              return "bronze";
                    }
          }, DIME(10) {
                    @Override
                    public String color() {
                              return "silver";
                    }
          }, QUARTER(25) {
                    @Override
                    public String color() {
                              return "silver";
                    }
          };
```

```
          private int value;

          public abstract String color();

          private Currency(int value) {
                    this.value = value;
          }
          ..............
}
```

Another example:
abstract validate() method for different form
fields (email, phone etc.)
– each form field is a different enum instance.

Prefer 2-element enum types over boolean params

[*Effective Java*, Item 40]

You might have a Thermometer type with a static factory that takes a value of
public enum TemperatureScale { FAHRENHEIT, CELSIUS }

Two benefits:

• Thermometer.newInstance(TemperatureScale.CELSIUS) looks much better than Thermometer.newInstance(true),

• you can easily add KELVIN to TemperatureScale later.

Java 8: towards a more functional language

New elements:

- lambda expressions,
- default methods,
- functional interfaces,
- streams (the package java.util.stream),
- collections API additions,
- concurrency API additions,
- …

…and minor changes

- Joining strings with a delimiter is finally easy: `String.join(", ", a, b, c)` instead of `a + ", " + b + ", " + c`.
- Integer types now support unsigned arithmetic.
- The `Math` class has methods to detect integer overflow.
- Use `Math.floorMod(x, n)` instead of `x % n` if `x` might be negative.
- There are new mutators in `Collection` (`removeIf`) and `List` (`replaceAll`, `sort`).
- `Files.lines` lazily reads a stream of lines.
- `Files.list` lazily lists the entries of a directory, and `Files.walk` traverses them recursively.
- There is finally official support for Base64 encoding.
- Annotations can now be repeated and applied to type uses.
- Convenient support for null parameter checks can be found in the `Objects` class.

From: C.S.Horstmann, *Java SE 8 for the Really Impatient*

Functional programming paradigm

Functional programming: a style of writing programs, that treats computation as the evaluation of mathematical functions and avoids state and mutable data.

Functional programming prefers functions that produce results
that depend only on their inputs (like math functions),
not on the program state.

It is a declarative programming paradigm; programming done with expressions. In (pure) functional code, calling function f twice with the same value for argument x will produce the same result f(x) both times.

Eliminating side effects, i.e. changes in state that do not depend on the function inputs can make it much easier to understand and predict the behavior of a program.

http://en.wikipedia.org/wiki/Functional_programming

Functional programming in practice

Purely functional (i.e., no side-effects)
and non-pure functional languages:
Haskell, Common Lisp, Scheme, OCaml, Clojure,
Erlang, Elixir, Elm...

Multiparadigm (mixing functional and OO):
F#, Scala.

With elements of functional programming:
Python (generators, list comprehensions, reduce…),
Ruby, Groovy,
C# (LINQ, lambda expressions, extension methods,
anonymous types)…

And Java, since v8.

Lambda expressions

(parameters) -> expression

or

(parameters) -> { statements; }

```
n -> n % 2 != 0;
(char c) -> c == 'y';
(x, y) -> x + y;
(int a, int b) -> a * a + b * b;
() -> 42
() -> { return 3.14 };
(String s) -> { System.out.println(s); };
() -> { System.out.println("Hello World!"); };
```

When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses.

When there is a single statement, curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.

http://java.dzone.com/articles/java-lambda-expressions-basics

Beware

```
(int x) -> { if (x >= 0) return 1; }   // compile error!
```

This is an invalid lambda expression.
It’s illegal to return a value in some branches
but not in others.

Functional interface

Functional interface is
an interface with a single abstract method.

There were f. i. before Java 8…

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}

public interface Callable<V> {
    V call() throws Exception;
}

public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}

public interface Runnable {
    public void run();
}
```

http://java.dzone.com/articles/java-lambda-expressions-basics

New functional interfaces (java.util.function)

```java
public interface Predicate<T> {
    boolean test(T t);
}

public interface Function<T,R> {
    R apply(T t);
}

public interface BinaryOperator<T> {
    T apply(T left, T right);
}

public interface Consumer<T> {
    void accept(T t);
}

public interface Supplier<T> {
    T get();
}
```

http://java.dzone.com/articles/java-lambda-expressions-basics

@FunctionalInterface

This annotation in front of a f. i. has two benefits:

• compile-time check if the annotated entity is indeed a functional interface,

• javadoc page includes a statement that this interface is a functional interface.

Type of lambda expression

```
Predicate<Integer> isOdd = n -> n % 2 != 0;
BinaryOperator<Integer> sum = (x, y) -> x + y;
Callable<Integer> callMe = () -> 42;
Block<String> printer =  (String s) -> { System.out.println(s); };
Runnable runner = () -> { System.out.println("Hello World!"); };
```

The type of the lambda expression is determined by the compiler from the context based on the target type
(you never specify it explicitly).

Same lambda code, different types
(because they are bound to a different target type):

```
Callable<String> callMe = () -> "Hello";
PrivilegedAction<String> action = () -> "Hello"
```

http://java.dzone.com/articles/java-lambda-expressions-basics

Why lambda? Shorter code!

Example. The interface Iterable contains now the method
void forEach(Consumer<? super T> action),
which performs the given action on the contents of the Iterable.

We want to use forEach to transpose the *x* and *y* coordinates of every element in a list of java.awt.Point.

Old style (using an anonymous inner class):

```
pointList.forEach(new Consumer() {
    public void accept(Point p) {
        p.move(p.y, p.x);
    }
});
```

New style (using a lambda expression):

```
pointList.forEach(p -> p.move(p.y, p.x));
```

http://www.lambdafaq.org/why-are-lambda-expressions-being-added-to-java/

addActionListener

```
JButton testButton = new JButton("Test Button");

testButton.addActionListener(e -> {
  System.out.println("Click detected by lambda listener");
});
...

frame.add(testButton, BorderLayout.CENTER);
```

http://blueskyworkshop.com/topics/Java-Pages/lambda-expression-basics/

Strategy pattern with lambdas

```
interface ValidationStrategy {
    public boolean execute(String s);
}

static private class IsAllLowerCase implements ValidationStrategy {
    public boolean execute(String s){
        return s.matches("[a-z]+");
    }
}
static private class IsNumeric implements ValidationStrategy {
    public boolean execute(String s){
        return s.matches("\\d+");
    }
}

static private class Validator{
    private final ValidationStrategy strategy;
    public Validator(ValidationStrategy v){
        this.strategy = v;
    }
    public boolean validate(String s){
        return strategy.execute(s); }
}
```

https://github.com/java8/Java8InAction/tree/master/src/main/java/lambdasinaction/chap8

Strategy pattern with lambdas, cont'd

```
// old school
Validator v1 = new Validator(new IsNumeric());
System.out.println(v1.validate("aaaa"));
Validator v2 = new Validator(new IsAllLowerCase ());
System.out.println(v2.validate("bbbb"));


// with lambdas
Validator v3 = new Validator((String s) -> s.matches("\\d+"));
System.out.println(v3.validate("aaaa"));
Validator v4 = new Validator((String s) -> s.matches("[a-z]+"));
System.out.println(v4.validate("bbbb"));
```

https://github.com/java8/Java8InAction/tree/master/src/main/java/lambdasinaction/chap8

JdbcTemplate example: old Java and Java 8

(org.springframework.jdbc.core.JdbcTemplate)

```
jdbcTemplate.query("SELECT bar FROM Foo WHERE baz = ?",
                   new PreparedStatementSetter() {
                     @Override
                     public void setValues(PreparedStatement ps)
                       throws SQLException
                     {
                       ps.setString(1, "some value for baz");
                     }
                   },
                   new RowMapper<SomeModel>() {
                     @Override
                     public SomeModel mapRow(ResultSet rs, int rowNum)
                       throws SQLException
                     {
                       return new SomeModel(rs.getInt(1));
                     }
                   }
    );
```

```
jdbcTemplate.query("SELECT bar FROM Foo WHERE baz = ?",
                   ps -> ps.setString(1, "some value for baz"),
                   (rs, rowNum) -> new SomeModel(rs.getInt(1))
);
```

[R. Fischer, *Java Closures and Lambda*, 2015], chap. 1

Lambdas may reference the class members and local variables

…but they must be effectively final.

```
final String separator = ",";
Arrays.asList( "a", "b", "d" ).forEach(
    ( String e ) -> System.out.print( e + separator ) );
```

```
String separator = ",";
Arrays.asList( "a", "b", "d" ).forEach(
  ( String e ) -> System.out.print( e + separator ) );
// separator = ".";
```

Compile-error if uncommented:
error: local variables referenced from a lambda expression must be final or effectively final

```
    ( String e ) -> System.out.print( e + separator ) );
```

http://www.javacodegeeks.com/2014/05/java-8-features-tutorial.html

What we can’t do with a lambda

We can’t declare function types like: (String, String) -> int.

We can’t declare variables of those types.
Can’t even assign a lambda to a variable of type Object,
since Object (obviously) is not a functional interface.

Note however we can assign a lambda to variable
(examples earlier, e.g.
Predicate<Integer> isOdd = n -> n % 2 != 0;)

Avoid too long/complicated lambdas,
they have no names and documentation.

Use generic types and methods

(J. Bloch, *Effective Java, 3rd ed.*, Item 42)

```
List<String> words = ... ;
Collections.sort(words, (s1, s2) ->
        Integer.compare(s1.length(), s2.length()));
```

```
List words = ... ;
Collections.sort(words, (s1, s2) ->
        Integer.compare(s1.length(), s2.length()));  // ?
```

Won’t compile! Fix the lambda with:

(String s1, String s2) -> …

Function descriptor

Any lambda expression may be thought of as
an anonymous representation of
a function descriptor of a functional interface.
(Imprecise definition: a function descriptor is,
usually, the single method of
this single-method interface.)

An alternative way representing a function descriptor
is with a concrete method of an existing class
(shown later).

http://www.lambdafaq.org/what-are-method-references/

Generic functional interfaces not always work as we could expect…

This is a correct piece of code:

```
BiFunction<String, String, Integer> comp
   = (first, second) ->
      Integer.compare(first.length(), second.length());
```

Yet, we can’t now use
java.util.Arrays.sort(…, comp);
→ Arrays.sort wants a Comparator, not a BiFunction.

Lambdas can be used only in a context expecting a functional interface

We want to compute a definite integral, using such an invocation:

```
integrate((double x) -> x + 1, 2, 6);
```

or

```
integrate((double x) -> f(x), 2, 6);
```

Here's the code:

```
public double integrate(DoubleFunction<Double> f,
                                   double a, double b)
{
  // stupid alg, of course
  return (f.apply(a) + f.apply(b)) * (b-a) / 2.0;
}
```

Based on: Urma et al., *Java 8 in Action*, 2014.

Lambdas and checked exceptions

If the body of a lambda expression may throw a **checked** exception, that exception needs to be declared in the abstract method of the target interface.

```
Runnable sleeper = () ->
  { System.out.println("Zzz"); Thread.sleep(1000); };
// Error: Thread.sleep can throw a
// checked InterruptedException !
```

Two solutions possible:

1. try ... catch in the lambda expression,
2. use another interface, whose single abstract method can throw the exception.

In our example: Callable is fine, since it can throw any exception. More precisely: Callable<Void> and don't forget to add *return null;* in the lambda.

From anonymous classes to lambda expressions

```
private void shadowVariablesInAnonymousClass() {
    // 1. this refer different in lambda and anonymous class
    // 2. shadow variables
    int a = 10;
    Runnable r1 = () -> {
        // int a =2; // can't compile
        System.out.println(a);
        System.out.println(this);
    };
    Runnable r2 = new Runnable() {
        @Override public void run() {
            int a = 2;
            System.out.println(a);
            System.out.println(this);
        }
    };
}
```

https://github.com/yehaote/java_8_practice/blob/master/src/main/java/mi/practice/java/eight/effective/RefactorDemo.java

Another issue

```java
interface Task {
    public void execute();
}


public static void doSomething(Runnable r) {
    r.run();
}

public static void doSomething(Task a) {
    a.execute();
}

private static void lambdaConflict() {
    doSomething(new Task() {
        @Override public void execute() {
            System.out.println("Danger danger!!");
        }
    });
    //doSomething(() -> System.out.println("Danger danger!!"));
    doSomething((Task) () -> System.out.println("Danger danger!!"));
}
```

https://github.com/yehaote/java_8_practice/blob/master/src/main/java/mi/practice/java/eight/effective/RefactorDemo.java

Functional interfaces for primitive types (int, long, double)

```
IntFunction<String> intToString = i -> Integer.toString(i);
ToIntFunction<String> parseInt = str -> Integer.valueOf(str);
IntPredicate isEven = i -> i % 2 == 0;
ToIntBiFunction<String,String> maxLength =
  (left,right) -> Math.max(left.length(), right.length());
IntConsumer printInt = i -> System.out.println(Integer.toString(i));
ObjIntConsumer<String> printParsedIntWithRadix =
  (str,radix) -> System.out.println(Integer.parseInt(str,radix));
IntSupplier randomInt = () -> new Random().nextInt();
IntUnaryOperator negateInt = i -> -1 * i;
IntBinaryOperator multiplyInt = (x,y) -> x*y;
IntToDoubleFunction intAsDouble = i -> Integer.valueOf(i).doubleValue();
DoubleToIntFunction doubleAsInt = d -> Double.valueOf(d).intValue();
IntToLongFunction intAsLong = i -> Integer.valueOf(i).longValue();
LongToIntFunction longAsInt = x -> Long.valueOf(x).intValue();
```

[R. Fischer, *Java Closures and Lambda*, 2015], chap. 2

Functional interfaces for primitive types, example

```
static void methodBeingCalled(IntFunction<String> function) {}
...
methodBeingCalled((int i) -> Integer.toString(i));
```

correct

```
static void methodBeingCalled(Function<Integer, String> function) {}
...
methodBeingCalled((int i) -> Integer.toString(i));
```

error: incompatible types: incompatible parameter types in lambda expression

```
methodBeingCalled((int i) -> Integer.toString(i));
```

[R. Fischer, *Java Closures and Lambda*, 2015], chap. 2

Method references

String::valueOf

or

Integer::compare

are examples of references to static methods.

Analogously to lambda expressions,

they do not capture any instance or local variables.

Use example. The method from java.util.Arrays:

```
public static <T> void sort(T[] a, Comparator c);
```

expects a Comparator for its second argument.

The method Integer.compare has a signature that is type-compatible with Comparator’s function descriptor – that is, its compare method – so it would be legal to call Arrays.sort like this:

```
Arrays.sort(myIntegerArray, Integer::compare)
```

http://www.lambdafaq.org/what-are-method-references/

Using method references

```
public static boolean isGreenApple(Apple apple) {
    return "green".equals(apple.getColor());
}
public static boolean isHeavyApple(Apple apple) {
    return apple.getWeight() > 150;
}

public interface Predicate<T>{ #A
    public boolean test(T t);
}

static List<Apple> filterApples(List<Apple> inventory,
                                Predicate<Apple> p) { #B
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (p.test(apple)) { #C
            result.add(apple);
        }
    }
    return result;
}
```

#A Included for clarity (normally simply imported from `java.util.function`)
#B A method is passed as parameter named `p`
#C Does the apple match the condition represented by `p`?

Using method references, cont'd

Filtering data with a (static) method passed as an argument:

```
filterApples(inventory, Apple::isGreenApple)
```

or

```
filterApples(inventory, Apple::isHeavyApple)
```

http://www.manning.com/urma/Java8Lambdas_MEAP_ch1.pdf

Instance method references (1 / 2)

There are two ways of referring to instance methods.
One analogous to the static case,
replacing the form ReferenceType::Identifier with
ObjectReference::Identifier.

Example: pointList.forEach(System.out::print);

Not very useful, as the argument to forEach (or any other method accepting a function in this way) cannot refer to the element that it is processing.

We often want to do smth like:
pointList.forEach(/* transpose x and y of this element */);
(assuming the elements of pointList belong to a class
having a method transpose())

http://www.lambdafaq.org/what-is-the-syntax-for-instance-method-references/

Instance method references (2 / 2)

Here helps another syntactic variant of
instance method references.
Assume we have a class TransPoint with a method
void transpose() { int t = x; x = y; y = t; }.

The form TransPoint::transpose — where a reference type rather than an object reference is used in conjunction with an instance method name — is translated by the compiler into a lambda expression like this:
(MyPoint pt) -> { pt.transpose(); }
— that is, a lambda expression is synthesized with a single parameter that is then used as the receiver for the call of the instance method.
So the syntax pointList.forEach(TransPoint::transpose);
achieves the result we wanted.

http://www.lambdafaq.org/what-is-the-syntax-for-instance-method-references/

Constructor references (SomeClass::new)

```
List<String> labels = ... ;
Stream<Button> stream = labels.stream().map(Button::new);
List<Button> buttons = stream.collect(Collectors.toList());
```

Which Button constructor? It is taken from the context; here map(...) is applied for a stream of Strings, hence the Button(String) constructor is called for each element from the list labels.

Interesting use with array types:
`int[]::new` is a constructor reference with one parameter: the length of the array.
It is equivalent to the lambda `x -> new int[x]`.

Default methods (1 / 2)

pointList.forEach(…) is a useful syntax,
but the (old) Java Collection Framework classes
don't have such a method.
Not easy to add it, as the JCF is heavily based on
interfaces and the framework coherence would be destroyed…

Can we add new methods to existing interfaces?
No, as ALL the classes implementing them would be
forced to change too.

The solution is a new concept to the language:
default methods (a.k.a. virtual extension methods,
defender methods).
Default methods enable interfaces to evolve without
introducing incompatibility with existing implementations.

Default methods (2 / 2)

The change is very significant:
the Java interfaces now can also declare concrete methods
in the form of default implementations.

For example, Iterator could hypothetically be extended by
a method that skips a single element:

```
interface Iterator {
    // existing method declarations
    default void skip() {
        if (hasNext()) next();
    }
}
```

With such a (hypothetically changed) Iterator, all implementing classes now automatically expose a skip method (and may use its default implementation, or override like for any virtual method), which a client code can call exactly as with abstract interface methods.

Multiple inheritance?

Well, yes and no.

Classical Java interfaces =
multiple inheritance of (interface) types

New Java interfaces (with default methods) =
multiple inheritance of behavior

Still no multiple inheritance of state (like in C++)!

Conflicting methods

```
public interface A {
  default void hello()
  { System.out.println("Hello World from A"); }
}

public interface B extends A {
  default void hello()
  { System.out.println("Hello World from B"); }
}

public class C implements B, A {
  public static void main(String... args)
  { new C().hello(); }
}
```

What does it print?

Answer:
Hello World from B

But what if B does not extend A? Then we have a compile-time error.

Forcing hello() from A:

```
public class C implements B, A {
    public void hello() {
        A.super.hello();
    }
    ...
}
```

New syntax:
superinterface (A),
keyword super,
method name (hello)

Now, what will it print?

```
interface A {
    default void m() { System.out.println("hello from A"); }
}
interface B extends A {
    default void m() { System.out.println("hello from B"); }
}
interface C extends A {}
class D implements B, C {}
```

```
C c = new D();
c.m();
```

Answer: hello from B
Because the static type of c is unimportant (remember, late binding!);
what counts is that it is an instance of D,
whose most specific version of m is inherited from B.

http://www.lambdafaq.org/what-about-the-diamond-problem/

More on the method resolution

Rule #1:
Classes win over interfaces.
If a class in the superclass chain has a declaration for the method (concrete or abstract), you're done, and defaults are irrelevant.

Rule #2:
More specific interfaces win over less specific ones (where specificity means "subtyping").
E.g., a default from List wins over a default from Collection.

Rule #3:
If there is not a unique winner according to the above rules, concrete classes must disambiguate manually.

http://mail.openjdk.java.net/pipermail/lambda-dev/2013-March/008435.html

Private methods in interfaces (Java 9)

```
public interface DBLogging{
    String MONGO_DB_NAME = "ABC_Mongo_Datastore";
    String NEO4J_DB_NAME = "ABC_Neo4J_Datastore";
    String CASSANDRA_DB_NAME = "ABC_Cassandra_Datastore";

    default void logInfo(String message){
      log(message, "INFO")
    }
    default void logWarn(String message){
      log(message, "WARN")
    }
    default void logError(String message){
       log(message, "ERROR")
    }
    default void logFatal(String message){
       log(message, "FATAL")
    }

    private void log(String message, String msgPrefix){
       Step1: Connect to DataStore
       Setp2: Log Message with Prefix and styles etc.
       Setp3: Close the DataStore connection
    }
    // Any other abstract methods
}
```

Idea:
code sharing between non-abstract methods in an interface

http://www.journaldev.com/12850/java9-private-methods-in-interface

Stream

A stream is a sequence of values.

There are stream types defined in java.util.stream:
Stream for streams of reference values
IntStream, LongStream, and DoubleStream for streams of primitives.

Streams are similar to iterators
but they are not associated with any particular storage mechanism.
A stream is either partially evaluated – some of its elements
remain to be generated – or exhausted, when its elements are all used up.

A stream can have as its source
an array, a collection, a generator function, or an IO channel;
alternatively, it may be the result of an operation on another stream.

Example:

```
List<String> strings = …;
IntStream ints = strings.stream().map(s -> s.length()).filter(i -> i%2 != 0);
```

http://www.lambdafaq.org/what-is-a-stream/

Stream, cont'd

Previous example: this code sets up a pipeline
which will first produce a stream of int values
corresponding to the lengths of the elements of strings,
then pass on the odd ones only.

But none of this happens as a result of the declaration of ints!
Streams are lazy!

Processing only takes place when a statement like
ints.forEach(System.out::println);
uses an eager terminal operation to pull values
down the pipeline.

http://www.lambdafaq.org/what-is-a-stream/

Creating a stream

```
Stream<String> words = Stream.of(text.split("\\W+"));

Stream<String> tale = Stream.of("Once", "upon", "a", "time...");

Arrays.stream(array, from, to);

Stream<String> empty = Stream.empty();
// generic type <String> is inferred;
// same as Stream.<String>empty()
```

What we can do with a Stream (examples)

operation	interface used	λ signature	return type	return value
	sample lazy/intermediate operations			
filter	Predicate<T>	T → boolean	Stream<T>	stream containing input elements that satisfy the **`Predicate`**
map	Function<T,R>	T → R	Stream<R>	stream of values, the result of applying the **`Function`** to each input element
sorted	Comparator<T>	(T, T) → boolean	Stream<T>	stream containing the input elements, sorted by the **`Comparator`**
limit, skip			Stream<T>	stream including only (resp. skipping) first *n* input elements
	sample eager/terminal operations			
reduce	BinaryOperator<T>	(T, T) → T	Optional<T>	result of reduction of input elements (if any) using supplied **`BinaryOperator`**
findFirst	Predicate<T>	T → boolean	Optional<T>	first input element satisfying **`Predicate`** (if any)
forEach	Consumer<T>	T → void	void	void, but applies the method of supplied **`Consumer`** to every input element

findFirst accepts no parameter, i.e. findFirst() !

http://www.lambdafaq.org/what-is-a-stream/

More on Streams

Stream types define intermediate operations (resulting in new streams), e.g. map, and terminal operations (resulting in non-stream values), e.g. forEach, reduce, DoubleStream.average.

Streams may be ordered or unordered.
A stream whose source is an array or a List (or an iterative application of a function) is ordered; one whose source is a Set is unordered.
Order is preserved by most intermediate operations; exceptional operations are sorted and unordered.

5 operation types in the streams framework, examples

Intermediate	Stateful Intermediate	Short-circuit Stateful Intermediate	Terminal	Short-circuit Terminal
filter()	distinct()	limit()	forEach()	anyMatch()
map()	sorted()		toArray()	allMatch()
peek()	skip()		reduce()	noneMatch()
			collect()	findFirst()
			min()	findAny()
			max()	
			count()	

Streams of primitives

There are stream specializations for primitive types: IntStream, LongStream, DoubleStream…

Conversions are possible:

```
List<String> strings = Arrays.asList("a", "b", "c");
strings.stream()                        // Stream<String>
        .mapToInt(String::length)       // IntStream
        .longs()                        // LongStream
        .mapToDouble(x -> x / 10.0)     // DoubleStream
        .boxed()                        // Stream<Double>
        .mapToLong(x -> 1L)             // LongStream
        .mapToObj(x -> "")              // Stream<String>
        ...
```

http://www.techempower.com/blog/2013/03/26/everything-about-java-8/

Stream processing on an example

[https://www.slideshare.net/SimonRitter/
lessons-learnt-with-lambdas-and-streams-in-jdk-8]

- A stream pipeline consists of three types of things
  - A source
  - Zero or more intermediate operations
  - A terminal operation
    - Producing a result or a side-effect

Source

```
int total = transactions.stream()
  .filter(t -> t.getBuyer().getCity().equals("London"))
  .mapToInt(Transaction::getPrice)
  .sum();
```

Intermediate operation

Terminal operation

allMatch, a simple example

```
boolean allFinished()
{
        // for (Cyclist c : cyclistSet)
        //     if (!c.finished())
        //         return false;
        // return true;
        return cyclistSet.stream().allMatch(c -> c.finished());
}
```

Or:

```
return cyclistSet.stream().allMatch(Cyclist::finished);
```

flatMap

```
Stream<List<Integer>> integerListStream = Stream.of(
  Arrays.asList(1, 2),
  Arrays.asList(3, 4),
  Arrays.asList(5)
);

Stream<Integer> integerStream = integerListStream
     .flatMap((integerList) -> integerList.stream());
integerStream.forEach(System.out::println);

  List<String> phrases = Arrays.asList(
       "sporadic perjury",
       "confounded skimming",
       "incumbent jailer",
       "confounded jailer");

  List<String> uniqueWords = phrases
       .stream()
       .flatMap(phrase -> Stream.of(phrase.split(" +")))
       .distinct()
       .sorted()
       .collect(Collectors.toList());
  System.out.println("Unique words: " + uniqueWords);
```

Stream.generate

generate

```
static <T> Stream<T> generate(Supplier<T> s)
```

Returns an infinite sequential unordered stream where each element is generated by the provided `Supplier`. This is suitable for generating constant streams, streams of random elements, etc.

Type Parameters:

`T` - the type of stream elements

Parameters:

`s` - the `Supplier` of generated elements

Returns:

a new infinite sequential unordered `Stream`

Use with limit().

```
Stream.generate(Math::random).limit(10).forEach(System.out::println);
```

iterate

static IntStream	iterate(int seed, IntUnaryOperator f) Returns an infinite sequential ordered IntStream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc.

```
IntStream.iterate(0, i -> i + 2)
   .limit(100)
   .forEach(System.out::println);
// prints: 0, 2, …, 198
```

Overloaded iterate in Java 9

limit() is OK, but can't be used with a condition.

New iterate() version, with an extra argument in the middle:

```
Stream.iterate(1, i -> i <= 10, i -> 2 * i)
     .forEach(System.out::println);
// output: 1 2 4 8
```

peek

The peek method is handy for debugging:
it yields another stream with the same elements as the original, but a function is invoked every time an element is retrieved.

Example:

```
Object[] powers = Stream.iterate(1.0, p -> p * 2)
    .peek(e -> System.out.println("Fetching " + e))
    .limit(20)
    .toArray();
```

For debugging, you can have peek call a method into which you set a breakpoint.

Parallel streams

Methods:

• parallelStream() in Collection class –
returns a possibly parallel stream,

• parallel() in BaseStream interface – returns an equivalent stream that is parallel. May return itself, either because the stream was already parallel, or because the underlying stream state was modified to be parallel

```
Stream<String> parallelWords = Stream.of(wordArray).parallel();
```

With a parallel stream the operations should be stateless!

WRONG CODE!
Race condition!

```
int[] shortWords = new int[8];
words.parallelStream().forEach(
   s -> { if (s.length() < 8) shortWords[s.length()]++; });
      // Error—race condition!
System.out.println(Arrays.toString(shortWords));
```

Different (and usually wrong) results each time are quite likely to be shown…

[C. Horstmann, *Core Java for the Impatient*], chap. 8.14

Previous problem – correct solution

```
Map<Integer, Long> shortWordCounts =
   words.parallelStream()
      .filter(s -> s.length() < 8)
      .collect(groupingBy(
         String::length, counting()));
```

It is the programmer's responsibility to ensure that any functions passed to parallel stream operations are safe to execute in parallel.

The best way = avoid mutable state.

In the example above: parallelization is safe as
the strings are grouped by length and then counted.

[C. Horstmann, *Core Java for the Impatient*], chap. 8.14

Don’t modify the collection that is backing a stream while carrying out a stream operation!

WRONG CODE! Outcome undefined.

```
Stream<String> words = wordList.stream();
words.forEach(s -> if (s.length() < 8) wordList.remove(s));
// Error—interference
```

[C. Horstmann, *Core Java for the Impatient*], chap. 8.14

Streams are a useful abstraction

Cay Horstmann speaking:

In my CS1 course, I give simple loop problems, such as "find all words that are long (> 10 characters) and end in the letter y." So, students have to write a loop with an if and a &&, and then, right then and there, they must decide where to put the result. Should they be printed? Collected in an array list?

```
for (String w : ...)
   if (w.length() > 10 && w.endsWith("y"))
      do the right thing with w
```

With streams, you just say:

```
stream.filter(w -> w.length() > 10).filter(w -> w.endsWith("y"))
```

and you get another stream back.
You can print, collect into an array list, or pass it along somewhere else.
And of course, if you wanted to parallelize the computation,
it's easy to do:

```
stream.parallel().filter(w -> w.length() > 10).filter(w -> w.endsWith("y"))
```

http://www.coderanch.com/t/630207/java/java/Java-SE-Impatient-language-features

takeWhile, dropWhile (Java 9)

```
Stream.of("a", "b", "c", "", "e")
     .takeWhile(s -> !String.isEmpty(s));
     .forEach(System.out::print);

Console: abc
```

Use it only for ordered streams (same for dropWhile)!

(For unordered ones, takeWhile will return an arbitrary subset of those elements that pass the predicate.)

```
Stream.of("a", "b", "c", "de", "f")
     .dropWhile(s -> s.length <= 1);
     .forEach(System.out::print);

Console: def
```

What does it do? (Java 9)

```
Files.lines(htmlFile)
    .dropWhile(line -> !line.contains("<meta>"))
    .skip(1)
    .takeWhile(line -> !line.contains("</meta>"))
```

http://blog.codefx.org/java/dev/java-9-stream/

Comparator

New methods: comparing, thenComparing, nullsFirst, naturalOrder, reversed, comparingInt, comparingLong, thenComparingInt…

```
Comparator c =
      Comparator
            .comparing(User::getLastName)
            .thenComparing(User::getFirstName);
```

```
Comparator<File> fileComparator =
     Comparator.nullsLast(
       Comparator.comparing(File::getName)
     );
```


68

java.util.Arrays.parallelSort(…)

The sorting algorithm is a parallel sort-merge that breaks the array into sub-arrays that are themselves sorted and then merged.
When the sub-array length reaches a minimum granularity, the sub-array is sorted using the appropriate Arrays.sort method.
If the length of the specified array is less than the minimum granularity, then it is sorted using the appropriate Arrays.sort(...).
The algorithm requires a working space no greater than the size of the original array.
The ForkJoin common pool is used to execute any parallel tasks.

```
PhenomII 3.0 GHz, 6 cores,
100M uniformly random longs:
sequential sort: 9.853s
parallel sort: 3.564s
```

http://download.java.net/jdk8/docs/api/java/util/Arrays.html#parallelSort%28long[]%29

New methods in List and Collection

```
List<Integer> aList = new ArrayList<>(
                          Arrays.asList(5, -2, 0, 1, -3, 0, 7));

aList.removeIf(x -> x < 0);
// added removeIf(Predicate<? super E> filter) in Collection

aList.sort(null);  // null Comparator -> natural ordering
aList.replaceAll(x -> (x == 0 ? null : x));

System.out.println(aList);
```

What does it do?

```java
IntStream stream = IntStream.of(1, 2);
stream.forEach(System.out::println);

// That was fun! Let's do it again!
stream.forEach(System.out::println);
```

```java
IntStream.iterate(0, i -> ( i + 1 ) % 2)
          .distinct()
          .limit(10)
          .forEach(System.out::println);
```

Alas, Java8 streams are verbose…

Simple task:
convert a comma-delimited string like "1,2,3,4"
into a list of integers.

Java 8+:

```
Stream.of(s.split(",")).map(Integer::parseInt).collect(Collectors.toList());
```

C# 4.0+:

```
s.Split(",").Select(int.Parse).ToList();
```

JavaScript:

```
s.split(",").map(Number)
```

Python:

```
[int(x) for x in s.split(",")]  // 3.x, 2.7
// or: map(int, s.split(",")) in Python 2.7
```

https://wrschneider.github.io/2016/06/26/java-8-too-little-too-late.html

java.util.Optional<T>

```
import java.util.*;

public class OptionalTest
{
  private static Integer f1(Integer i)  { return i+1; }

  private static Integer f2(Integer i)  { return i+2; }

  private static Integer f3(Integer i)  { return i+3; }

  public static void main(String ... args) {
    Integer i = 4;
    String s = Optional.ofNullable(i).
                  map(OptionalTest::f1).map(OptionalTest::f2).map(OptionalTest::f3).
                  orElse(-1).toString();
    System.out.println("result = " + s);

    i = null;
    s = Optional.ofNullable(i).
           map(OptionalTest::f1).map(OptionalTest::f2).map(OptionalTest::f3).
           orElse(-1).toString();
    System.out.println("result = " + s);
  }
}
```

More examples:
http://java.dzone.com/articles/optional-java-8-cheat-sheet

result = 10
result = -1

Dealing with Optionals

```java
public class Person {

    private Optional<Car> car;

    public Optional<Car> getCar() {
        return car;
    }
}
```

```java
public class Insurance {

    private String name;

    public String getName() {
        return name;
    }
}
```

```java
public String getCarInsuranceName(Optional<Person> person) {
    return person.flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 .orElse("Unknown");
}
```

https://github.com/java8/Java8InAction/tree/master/src/main/java/lambdasinaction/chap10

Terminal stream operations returning Optional<T>

```
Optional<T> reduce(BinaryOperator<T> accumulator)
Optional<T> min(Comparator<? super T> comparator)
Optional<T> max(Comparator<? super T> comparator)
Optional<T> findFirst()
Optional<T> findAny()
```

Modules (Java 9)

Goals of the modular approach:

• strong encapsulation: precisely define which parts of our code will be available for other modules,
• Java code (finally) has the possibility to know about its own dependencies and thus have reliable configuration,
• …yet loose coupling of modules possible.

(Modules are the building blocks of microservices, modules enforce the single responsibility principle, ...)

Motivation:

have you experienced NoClassDefFoundErrors at runtime? It means missing dependencies, i.e. an unreliable configuration.

https://labs.consol.de/development/2017/02/13/getting-started-with-java9-modules.html

What is Java 9 module?

A module is a grouping of packages.
Basically a (standard) JAR file, but one of the files is called module-info.java.
It declares our module and defines:

- the unique name of our module,
- which other modules our module depends on,
- which packages are to be exported to be used by other modules.

Public and protected members of unexported packages are inaccessible outside the module.

Module name & structure:
reversed-domain-pattern for names preferred (as for packages).

```
src
+- main
   +- de.consol.devday.service
      +- module-info.java
      +- de
         +- consol
            +- devday
               +- service
                  +- EventService.java
```

https://labs.consol.de/development/2017/02/13/getting-started-with-java9-modules.html

module-info.java examples

module de.consol.devday.service
{ **exports de.consol.devday.service;** }
// exporting a package, i.e., allows other modules
// to read everything in package de.consol.devday.service

module de.consol.devday.service
{ **exports de.consol.devday.service to de.consol.devday.admin** }
// only the module de.consol.devday.admin can read
// the package de.consol.devday.service!

module de.consol.devday
{ **requires de.consol.devday.service;** }
// declares a dependency

https://labs.consol.de/development/2017/02/13/getting-started-with-java9-modules.html

Loose coupling of modules (1 / 2)

Say we don’t want the de.consol.devday module to know about the actual implementation of the service it consumes.

We can refactor de.consol.devday.service to only provide a service interface (call it de.consol.devday.service.EventService), and move the impl. to a separate module de.consol.devday.talk.service.

Another module can provide an implementation by enhancing its module declaration:

```
module de.consol.devday.talk.service {
   requires de.consol.devday.service;
   exports de.consol.devday.talk.service;
   provides de.consol.devday.service.EventService
      with de.consol.devday.talk.service.TalkService;
}
```

Loose coupling of modules (2 / 2)

The consuming module (→ de.consol.devday) can register as a consumer of a service by applying the uses keyword together with the full-qualified name of the interface:

```
module de.consol.devday {
    requires de.consol.devday.service;
    uses de.consol.devday.service.EventService;
}
```

That’s it. The module does not know anything about the impl. of EventService, nor does it need know about the module that provides it.
Using the service is straightforward:

```
ServiceLoader.load(EventService.class)
// returns an Iterable<T>, containing all service implementations
of a given interface that are offered by modules on the modulepath
using the "provides ... with ..." statement.
```

New Date and Time API in Java 8

Date and Time API in (old) Java: among the most criticized.

What's wrong with java.util.Date (Java 1.0):

- constructors that accept year arguments require offsets from 1900 (a source of bugs),
- January is represented by 0 instead of 1 (a source of bugs),
- Date doesn't describe a date but describes a date-time combination,
- Date's mutability makes it unsafe to use in multithreaded scenarios without external synchronization.

java.util.Calendar (Java 1.1) to the rescue? Not quite:

- not possible to format a calendar,
- January is represented by 0 instead of 1 (again),
- ...

java.sql's Date, Time, and Timestamp classes extend java.util.Date (--> same problems)

http://www.javaworld.com/javaworld/jw-04-2013/130408-j101-date-and-time-api.html

JSR 310: Date and Time API design principles

- immutability and thread safety,
- fluency (now(), from() etc.),
- clarity,
- extensibility.

The new API distinguishes between
machine and human views of a timeline,
which is an always increasing sequence of instants,
or points along the timeline.

The machine view: a sequence of integral values
relative to the epoch (e.g., midnight, January 1, 1970).
The human view: a set of fields
(e.g., year, month, day-of-month, hour etc.).

Packages

About 60 classes!

Main package: java.time

Other packages:

- java.time.chrono (generic API)
- java.time.format (formatting and parsing date-time objects)
- java.time.temporal
  (field, unit, or adjustment access to a temporal object)
- java.time.zone (time zones and their rules)

Instant and Duration; example

```java
import java.time.Duration;
import java.time.Instant;

public class MachineTimeDemo
{
   public static void main(String[] args)
   {
      System.out.printf("EPOCH = %s%n", Instant.EPOCH);
      System.out.printf("MAX = %s%n", Instant.MAX);
      System.out.printf("MIN = %s%n", Instant.MIN);

      System.out.printf("Now = %s%n", Instant.now());
      System.out.printf("50 seconds past epoch = %s%n",
                        Instant.ofEpochSecond(50));
      System.out.printf("Parsed = %s%n",
                        Instant.parse("2007-12-03T10:15:30Z"));

      System.out.printf("ZERO = %s%n", Duration.ZERO);

      System.out.printf("30-second duration = %s%n",
                        Duration.ofSeconds(30));
      System.out.printf("Parsed = %s%n",
                        Duration.parse("PT3M20S"));
   }
}
```

```
EPOCH = 1970-01-01T00:00:00Z
MAX = +1000000000-12-31T23:59:59.999999999Z
MIN = -1000000000-01-01T00:00:00Z
Now = 2013-07-23T17:22:33.044Z
50 seconds past epoch = 1970-01-01T00:00:50Z
Parsed = 2007-12-03T10:15:30Z
ZERO = PT0S
30-second duration = PT30S
Parsed = PT3M20S
```

http://www.javaworld.com/javaworld/jw-04-2013/130408-j101-date-and-time-api.html?page=3

Instant and Duration getters

Instant:

int getNano() returns the number of nanoseconds past the last second

boolean isAfter(Instant otherInstant) returns true when this Instant comes after the specified Instant

Duration:

long getSeconds() returns the number of seconds in the duration

boolean isNegative() returns true when this Duration is negative

Instant and Duration object manipulation

Instant minusSeconds(long secondsToSubtract)
returns a copy of the given Instant decreased by the specified number of seconds.

Instant plusNanos(long nanosToAdd)
returns a copy of the given Instant increased by the specified number of nanoseconds.

Duration dividedBy(long divisor)
returns a copy of the given Duration divided by the specified value.

Duration minusDays(long daysToSubtract)
returns a copy of the given Duration decreased by daysToSubtract.

Duration multipliedBy(long multiplicand)
returns a copy of the given Duration multiplied by the specified value.

Duration plusHours(long hoursToAdd)
returns a copy of the given Duration increased hoursToAdd.

LocalDateTime

```
import java.time.*;
import java.time.temporal.*;
```

```
LocalDateTime localDateTime = LocalDateTime.now();
System.out.printf("Date-time: %s%n", localDateTime);
System.out.printf("Date-time: %s%n", localDateTime.minusWeeks(7).minusHours(1));
System.out.printf("Date-time: %s%n", localDateTime.plusMonths(2));
System.out.println();
LocalDateTime localDateTime2 = localDateTime.withYear(2016).minusMonths(9);
System.out.printf("Date-time: %s%n",
  localDateTime2.with(TemporalAdjusters.lastDayOfMonth()));
```

```
Date-time: 2018-11-07T19:22:03.420601600
Date-time: 2018-09-19T18:22:03.420601600
Date-time: 2019-01-07T19:22:03.420601600

Date-time: 2016-02-29T19:22:03.420601600
```

Adjusters

Most core classes implement the TemporalAdjuster interface.
Usage:
pass it to a with() method such as LocalDateTime's
LocalDateTime with(TemporalAdjuster adjuster) method.

```
LocalDate localDate = LocalDate.of(2010, 12, 1);
LocalTime localTime = LocalTime.of(10, 15);
localDateTime = localDateTime.with(localDate).with(localTime);
System.out.printf("Date-time: %s%n", localDateTime);
```

```
Date-time: 2010-12-01T10:15
```

Some other date/time classes

public final class Period extends Object implements TemporalAmount, Serializable

A date-based amount of time,
such as '2 years, 3 months and 4 days'.
This class models a quantity or amount of time in terms of years, months and days.
Like Duration, but date-based, not time-based.

public final class YearMonth extends Object implements Temporal, TemporalAdjuster, Comparable<YearMonth>, Serializable

A year-month in the ISO-8601 calendar system, such as 2007-12.

Respecting time zones

```
void flightTime()
{
  ZoneId LHR = ZoneId.of("Europe/London");
  ZoneId SFO = ZoneId.of("America/Los_Angeles");
  LocalDate date = LocalDate.of(2013, Month.SEPTEMBER, 14);
  LocalTime takeoff = LocalTime.of(12, 50);
  LocalTime landing = LocalTime.of(16, 20);
  Duration flightTime = Duration.between(
    ZonedDateTime.of(date, takeoff, LHR),
    ZonedDateTime.of(date, landing, SFO)
  );
  System.out.println("Flight time: " + flightTime);
}
```

Flight time: PT11H30M

Example from: http://es.slideshare.net/ags313/java8-new-things

Why ‘lambda’ (calculus)

In Russell and Whitehead’s [1910–13] Principia Mathematica the notation for the function f with $f(x) = 2x + 1$ is $2\hat{x} + 1$. Church originally intended to use the notation $\hat{x}.2x + 1$. The typesetter could not position the hat on top of the x and placed it in front of it, resulting in ${}_{\wedge}x.2x + 1$. Then another typesetter changed it into $\lambda x.2x + 1$.

http://enl.usc.edu/~jkna/fpl/church.pdf