Java 8 (with cats)

- Basics for all I could find of new stuff for Java 8.
- Some Java 7 too.

Thorbjørn Ravn Andersen - tra@statsbiblioteket.dk

JVM stuff first...

invokedynamic

New byte code instruction added.

Allows user space code to help Hotspot to resolve type information at runtime instead of compile time.

Is very helpful to "duck typing"-languages like JRuby.

Also used for creating the object representation for λ-expressions.

http://blog.headius.com/2008/09/first-taste-of-invokedynamic.html

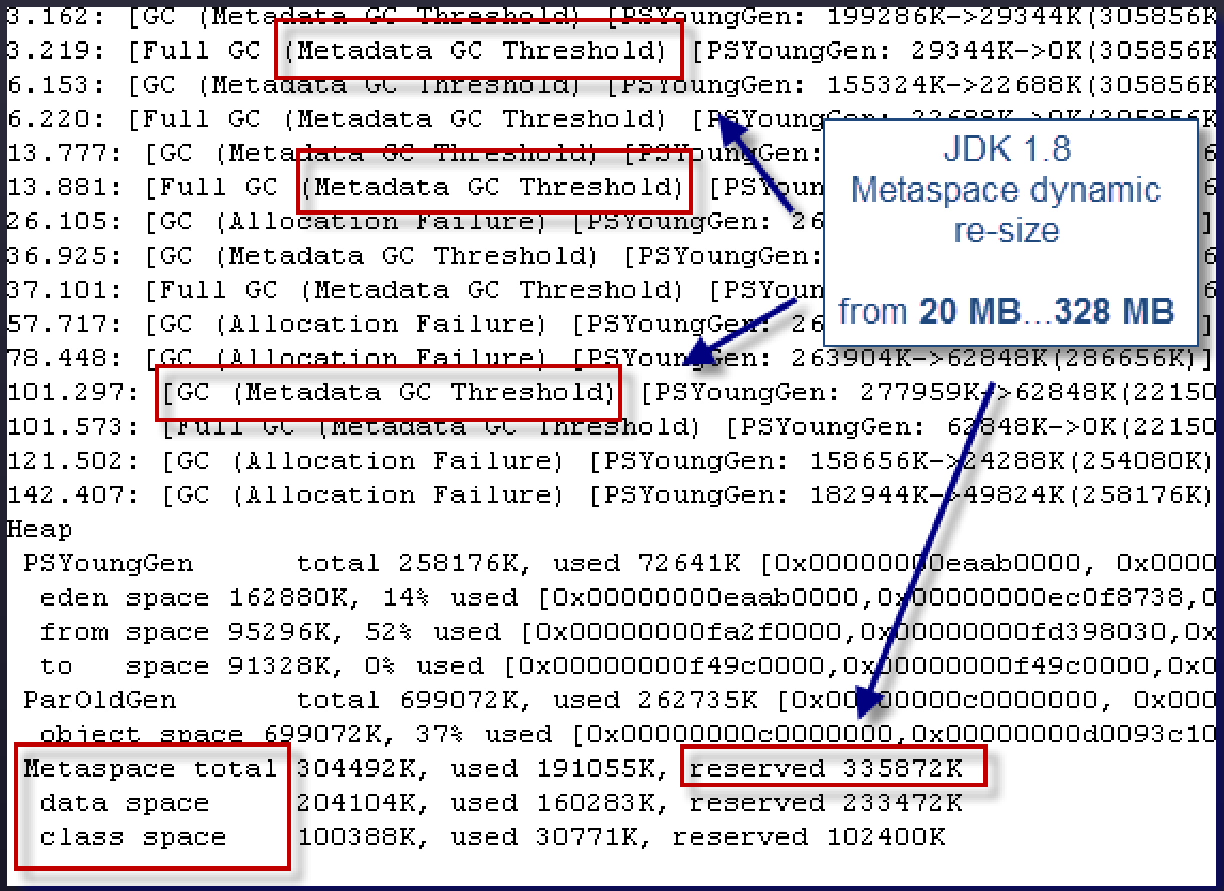
Garbage Collection

Permgen is replaced with Metaspace which use native memory instead of a fixed size pool.

Can grow *much* bigger!

- `-verbose:gc` - activate simple GC logging.
- `-XX:+PrintGCDetails` - activate detailed GC logging.
- `-Xloggc:<file>` - send GC log to *file* instead of console.
- `-XX:+PrintGCDateStamps` - add date stamp to every line.
- `-XX:+PrintGCTimeStamps` - adds seconds since JVM start to every line.

Example:



```
3.162: [GC (Metadata GC Threshold) [PSYoungGen: 199286K->29344K(305856K
3.219: [Full GC (Metadata GC Threshold) [PSYoungGen: 29344K->0K(305856K
6.153: [GC (Metadata GC Threshold) [PSYoungGen: 155324K->22688K(305856K
6.220: [Full GC (Metadata GC Threshold) [PSYoungGen: 22688K->0K(305856K
13.777: [GC (Metadata GC Threshold) [PSYoungGen:                       6
13.881: [Full GC (Metadata GC Threshold) [PSYoun                       6
26.105: [GC (Allocation Failure) [PSYoungGen: 26                       ]
36.925: [GC (Metadata GC Threshold) [PSYoungGen:                       6
37.101: [Full GC (Metadata GC Threshold) [PSYoun                       6
57.717: [GC (Allocation Failure) [PSYoungGen: 26                       ]
78.448: [GC (Allocation Failure) [PSYoungGen: 263904K->62848K(286656K)]
101.297: [GC (Metadata GC Threshold) [PSYoungGen: 277959K->62848K(22150
101.573: [Full GC (Metadata GC Threshold) [PSYoungGen: 62848K->0K(22150
121.502: [GC (Allocation Failure) [PSYoungGen: 158656K->24288K(254080K)
142.407: [GC (Allocation Failure) [PSYoungGen: 182944K->49824K(258176K)
Heap
 PSYoungGen      total 258176K, used 72641K [0x00000000eaab0000, 0x0000
  eden space 162880K, 14% used [0x00000000eaab0000,0x00000000ec0f8738,0
  from space 95296K, 52% used [0x00000000fa2f0000,0x00000000fd398030,0x
  to   space 91328K, 0% used [0x00000000f49c0000,0x00000000f49c0000,0x0
 ParOldGen       total 699072K, used 262735K [0x00000000c0000000, 0x000
  object space 699072K, 37% used [0x00000000c0000000,0x00000000d0093c10
 Metaspace total 304492K, used 191055K, reserved 335872K
  data space     204104K, used 160283K, reserved 233472K
  class space    100388K, used 30771K, reserved 102400K
```

https://dzone.com/articles/java-8-permgen-metaspace

Garbage collectors

Default garbage collector is still Parallel (which can stop the world).

Concurrent-Mark-Sweep performs better due to background work but can still stop the world.

Alternative G1 is for +4GB heaps which can do string deduplication. May be default in Java 9. Rarely stops the world.

Don't mess with it unless you know exactly what you are doing. This includes accepting JVM flags for previous versions of Java!

New tools

Java Packager - "performs task related to packaging and signing Java/JavaFX applications". JavaFX is a new GPU-powered UI for Java, and is not covered in this presentation.

jdeps - static dependencies of applications and libraries.

jjs - JVM-based JavaScript engine useful for scripting (more later)

Java Flight Recorder can collect low level data for Java Mission Control to allow after-the-fact incident analysis. (Commercial, can be enabled at runtime). Worth looking into.

Advanced Management Console can give an overview of Java applications and their JRE's in an organization. Runs in WebLogic 12. (Very commercial)

javac

New option `-parameters` saves the names of constructor and method parameters so they can be retrieved by reflection. New checkbox in Eclipse.

New option `-profile X` ensures that the source conforms to compact profile X (a well defined subset of the JRE).

- "compact1" is for simple command line programs.
- "compact2" is "compact1" + RMI/JDBC/XML
- "compact3" is "compact2" + JMX/JNDI/security/annotations

https://blogs.oracle.com/jtc/entry/a_first_look_at_compact

Nashorn - JSR-223 Javascript engine

Newly written for Java 8 to utilize `invokedynamic`.

- is 2-3 times slower than the highly optimized V8 engine after warmup.
- can easily be used for scripts with `jjs -scripting` .
- has full access to the standard Java libraries including JavaFX.
- can be used for configuration scripts (code as opposed to stringly typed properties) with the JSR-223 classes.
- Can use many Node.js packages.
- Netbeans 8+ and IntelliJ 13+ can be used for debugging.

Nashorn scripting:

```
#!/usr/bin/env jjs -scripting

print(<<EOD);
...${arguments[0]}---
EOD
```

Full access to JVM including JavaFX. Sweet spot is single-class-sized application possibly with a GUI. Main developer side project is "Nasven" to put Maven artifacts on the classpath.

Does not yet appear to have reached critical mass or found a killer-application.

Miscellaneous.

- Endorsed dirs mechanism and extension mechanism are deprecated as of 8u40.
- Scheduled regular security updates with predictable numbers.
- JRE expires automatically when the next security update is released.

But what can we do in *our own code*?

I/O

New I/O classes replacing `java.io.File` with friends. Not hard bound to underlying operating system. File system implementations can be added (e.g. ZipFileSystem).

`java.nio.file.Path` - text-only representation of a location in a given file system. Bridge: `File.toPath()`.

`java.nio.file.Paths` - operations on paths.

`java.nio.file.Files` - operations on file systems.

Note: All output shown has been generated on Linux. Comments indicate what would be printed if the expression was printed.

Path + Paths:

```
Paths.get("/tmp/foo");
Paths.get(args[0]);
Paths.get(URI.create("file:///Users/joe/FileTest.java"));
FileSystems.getDefault().getPath("/users/sally");

Path p5 = Paths.get(System.getProperty("user.home"),
                    "logs", "foo.log");

p5.toAbsolutePath() // "/home/tra/logs/foo.log"
p5.toUri()          // "file:///home/tra/logs/foo.log"
Paths.get("/proc", ".", "..", "tmp").normalize() // "/tmp"

p5.toRealPath()
// Exception ...NoSuchFileException: /home/tra/logs/foo.log
```

```
Paths.get("/tmp").resolve("log")
// /tmp/log

Paths.get("/tmp/foo").relativize(Paths.get("/tmp/bar"))
// ../bar

Paths.get("/tmp/foo").equals(Paths.get("/tmp/foo"))
// true

Paths.get("/tmp/foo").endsWith("foo")
// true
Paths.get("/tmp/foo").endsWith("oo")
// FALSE <- not string level.
```

`Files.exists(Path, LinkOption...)` and `Files.notExists(Path, LinkOption...)` can be used to examine if a Path exists or not. If both returns false, the existence of the file cannot be verified.

`Files.isReadable(Path)`, `Files.isWritable(Path)`, and `Files.isExecutable(Path)` examines the permissions at the time of execution. Beware race conditions.

`Files.isSameFile(Path, Path)` compares two paths - respecting symbolic links - to see if they locate the same file in the file system.

`Files.delete(Path)` throws an exception if the deletion fails (NoSuchFileException, DirectoryNotEmptyException, IOException, SecurityException) which can be examined to handle the problem.

`Files.deleteIfExists(Path)` does not throw an exception if the file did not exist.

`Files.copy(Path, Path, CopyOption...)` provides a built-in copy method. Target must not exist, unless `CopyOption.REPLACE_EXISTING` is provided.

Directories can be copied. Files inside directories will not be copied.

When copying a symbolic link, the target of the link is copied. To copy the symbolic link itself provide `CopyOption.NOFOLLOW_LINKS` or `CopyOption.REPLACE_EXISTING`.

`CopyOption.COPY_ATTRIBUTES` copies the file attributes too. Notably last-modified-time.

`Files.copy(...)` also accepts InputStream and OutputStream to copy single byte streams.

`Files.move(Path, Path, CopyOption...)` moves a file or directory. The move fails if the target file exists, unless `CopyOption.REPLACE_EXISTING` is provided. If the target is a symbolic link, the link is replaced but what it pointed to is unaffected.

Empty directories can be moved. If the directory is not empty, the move is allowed when the directory can be moved without moving the contents of that directory. (For Unix that would be possible within the same partition by renaming the directory)

`CopyOptions.ATOMIC_MOVE` performs the move as an atomic operation. An exception is thrown if this is not supported by the file system. Doing this guarantees that any process watching the directory accesses a complete file.

Metadata (yay!)

- `Files.size(Path)` - in bytes
- `Files.isDirectory(Path, LinkOption)`
- `Files.isRegularFile(Path, LinkOption...)`
- `Files.isSymbolicLink(Path)`
- `Files.isHidden(Path)`
- `Files.getLastModifiedTime(Path, LinkOption...)` and `Files.setLastModifiedTime(Path, FileTime)`

More metadata!

- `Files.getOwner(Path, LinkOption...)` and `Files.setOwner(Path, UserPrincipal)`
- `Files.getPosixFilePermissions(Path, LinkOption...)` and `Files.setPosixFilePermissions(Path, Set<PosixFilePermission>)`
- `Files.getAttribute(Path, String, LinkOption...)` and `Files.setAttribute(Path, String, Object, LinkOption...)`
- `Files.readAttributes(Path, String, LinkOption...)` and `Files.readAttributes(Path, Class<A>, LinkOption...)`
- `Files.getFileStore(Path)` - get total space, unallocated space, and usable space (and more).

Reading/writing/creating

Many methods now take an optional `OpenOption` parameter. The `StandardOpenOptions` enum provide:

- `WRITE` - open for write access.
- `APPEND` - append new data to end of file (WRITE, CREATE)
- `TRUNCATE_EXISTING` - truncates file to zero bytes (WRITE)
- `CREATE_NEW` - create new file, throw exception if already exists.
- `CREATE` - open file if it exists, create new otherwise.
- `DELETE_ON_CLOSE` - delete the file when the stream is closed. Good for temporary files.
- `SPARSE` - hint that new file will be sparse. Supported by NTFS and possibly others.
- `SYNC` - keep the file (both content and metadata) synchronized with the underlying storage device.
- `DSYNC` - keep the file content synchronized with the underlying storage device.

Misc helpers:

```
byte[] fileArray = Files.readAllBytes(Paths.get(...));
Files.write(Paths.get(...), fileArray);
Files.createFile(Path)
Files.createTempFile(Path, String, String, FileAttribute<?>)
```

Helpers for as efficient I/O as possible:

```
Files.newBufferedReader(Path, Charset, OpenOption...);
Files.newBufferedWriter(Path, Charset, OpenOption...);
Files.newInputStream(Path, OpenOption...);
Files.newOutputStream(Path, OpenOption...);
Files.newByteChannel(Path, OpenOption...)
```

```
FileSystems.getDefault().getRootDirectories()
Files.createDirectory(Path)
Files.createTempDirectory(Path, String, FileAttribute<?>)
Files.createSymbolicLink(Path, Path, FileAttribute<?>)
Files.createLink(Path, Path) // hard link!
Files.readSymbolicLink(Path) // resolve link
```

File names in a directory can be read one by one with `Files.newDirectoryStream(Path, ...)`. An optional glob filter can be used.

Walking the file tree:

Implement `FileVisitor<Path>` or override `SimpleFileVisitor<Path>` and use it with `Files.walkFileTree(Path, FileVisitor)`. Can weld under water.

For most uses `File.walk(Path)` may be sufficient.

Watching a directory for changes:

- Create a `WatchService`
- For each directory, get a `WatchKey` by registering it and the events you want with the watch service
- Create infinite listening loop listening for events.
- When an event occurs, the key is signalled and put in the watch service queue.
- Retrieve the key from the watch service queue. The file name is in the key.
- Process all pending events for the key as needed.
- Reset the key, and resume waiting for events.
- Close the service.

https://docs.oracle.com/javase/tutorial/essential/io/notification.html

WatchService Events:

- `ENTRY_CREATE`
- `ENTRY_DELETE`
- `ENTRY_MODIFY`
- `OVERFLOW` - events may have been lost or discarded. Does not require registration.

This is untested by me. It is adapted from the Oracle Java Tutorial page.

Lots of more fun!

- Asynchronous I/O - call backs - needed for scalability
- Socket-channel functionality - sockets refactored
- Create own FileSystemProvider (like ZipFileSystem)
- `java.io.Console` - Character-based console device. `readPassword()`

You'll know when you need it!

https://docs.oracle.com/javase/8/docs/technotes/guides/io/fsp/zipfilesyster

Dizzy yet?

new Date+Time API (JSR-310)

http://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html

The JODA library has for a long time been recommended instead of the default `Calendar` and `Date` classes, especially for differences.

JSR-310 set out to produce a new set of date and time handling classes for the standard Java library, where the obvious choice was to adapt JODA, but the architect disagreed with some of the design decisions, and took the opportunity to fix them. JSR-310 was included in Java 8 under `java.time.*`.

The JODA web site recommends using the new routines instead of JODA for Java 8 onwards. There is not an immediate upgrade path from JODA to `java.time.*` so some manual work is needed to do so.

Highlights:

- Values are always immutable (also known as thread safe).
- Easy to express *durations* useful relative to a given time.
- Machine time - `Instant` - is separated from human perception of time (like days in a calendar, clock on a wall).
- Human time - `LocalDateTime` - as seen on calendars and watches has no notion of machine time.
- The concept of time *passing* is abstracted out - `Clock` - making it much easier to write tests involving time.
- Chronologies are abstracted out, allowing non-standard calendars. Useful in Japan and Thailand.

java.util.Date.toInstant()

"To help bridge the gap between the old and new API's, the venerable Date class now has a new method called toInstant() which converts the Date into the new representation. This can be especially effective in those cases where you're working on an API that expects the classic form, but would like to enjoy everything the new API has to offer."

```
System.out.println(new java.util.Date().toInstant());
// 2016-01-12T09:37:14.910Z
```

Note: Z==Zulu/UTC time!

Bridging methods:

- `Calendar.toInstant()` converts the Calendar object to an Instant.
- `GregorianCalendar.toZonedDateTime()` converts a GregorianCalendar instance to a ZonedDateTime.
- `GregorianCalendar.from(ZonedDateTime)` creates a GregorianCalendar object using the default locale from a ZonedDateTime instance.
- `Date.from(Instant)` creates a Date object from an Instant.
- `Date.toInstant()` converts a Date object to an Instant.
- `TimeZone.toZoneId()` converts a TimeZone object to a ZoneId.

LocalDate + LocalTime + LocalDateTime

Note: `Month` and `DayOfWeek` enums can make code more readable. `Year` class has `isLeap()` method to help identify leap years.

- LocalDate represents a date as seen from the context of the observer, like a calendar on the wall.
- LocalTime represents a specific time in a day as seen from the context of the observer, like a clock on the wall.
- LocalDateTime combines both.

Note that there is no trailing timezone indicator.

```
LocalDateTime.now() // 2016-01-12T10:37:14.908
LocalDate.of(2012, Month.DECEMBER, 12) // 2012-12-12
LocalDate.ofEpochDay(150) // 1970-05-31
LocalTime.of(17, 18) // 17:18
LocalTime.parse("10:15:30") // 10:15:30
```

Method naming conventions 1/2:

- `of` - creates an instance where the factory is primarily validating the input parameters, not converting them.
- `from` - converts the input parameters to an instance of the target class, which may involve losing information from the input.
- `parse` - parses the input string to produce an instance of the target class.
- `format` - uses the specified formatter to format the values in the temporal object to produce a string.
- `get` - returns a part of the state of the target object.
- `is` - queries the state of the target object.

Method naming conventions 2/2:

- `with` - returns a copy of the target object with one element changed; this is the immutable equivalent to a set method on a JavaBean.
- `plus` - returns a copy of the target object with an amount of time added.
- `minus` - returns a copy of the target object with an amount of time subtracted.
- `to` - converts this object to another type.
- `at` - combines this object with another.

Adjustments can be made:

```
LocalDate.now().withDayOfMonth(10).withYear(2010)
// 2010-01-10
LocalDate.now().plusWeeks(3).plus(3, ChronoUnit.WEEKS)
// 2016-02-23
```

"Temporal adjusters" know of calendars.

```
LocalDate.now()
.with(java.time.temporal.TemporalAdjusters.lastDayOfYear())
// 2016-12-31
```

Different precisions may be interesting, like rounding to the number of seconds or days a given time corresponds to:

```
LocalDateTime.now().truncatedTo(ChronoUnit.DAYS)
// 2016-01-12T00:00
```

An exception is thrown if the truncation unit is incompatible with the value.

Additionally the `MonthDay` class is useful for birthdays. The `YearMonth` class is well suited for credit card expiration dates.

All these classes are supported in JDBC by the getObject/setObject methods, but do not have dedicated helper methods.

Timezones

Instructional video: (without cats)

https://www.youtube.com/watch?v=-5wpm-gesOY

`ZoneOffset` is the period offset from UTC.

```
ZoneOffset offset = ZoneOffset.of("+2:00");
```

`ZoneId` is an identifier for a time zone region. Use "PLT" or longer like "Europe/Copenhagen".

`ZonedDateTime` is "a date and time with a fully qualified time zone. This can resolve an offset at any point in time. The rule of thumb is that if you want to represent a date and time without relying of the context of a specific server, you should use ZonedDateTime.

```
ZoneId id = ZoneId.of("Europe/Copenhagen");
ZonedDateTime zdt = ZonedDateTime.of(LocalDateTime.now(), id);
System.out.println(zdt);
// 2016-01-12T12:45:16.923+01:00[Europe/Copenhagen]
```

Strings can be parsed:

```
ZonedDateTime.parse("2007-12-03T10:15:30+01:00[Europe/Paris]")
.truncatedTo(ChronoUnit.HOURS)
// 2007-12-03T10:00+01:00[Europe/Paris]
```

Dates and times can be formatted:

```
LocalDateTime.now().format(
    DateTimeFormatter.ofPattern("YYYY-MM-dd HH:mm"))
// 2016-01-12 16:09
```

When needing to serialize the zoned date times, convert them to an `OffsetDateTime`, where the timezone is resolved to an offset.

```
ZoneId id = ZoneId.of("Europe/Copenhagen");
ZonedDateTime zdt =
    ZonedDateTime.of(LocalDateTime.now(), id);

OffsetDateTime odt = OffsetDateTime.from(zdt);
System.out.println(odt); // 2016-01-12T12:52:49.665+01:00
```

There are many helper methods to massage these values further.

Periods

"Periods represents a date-based value such as '3 months and a day', which is a distance on the timeline" in terms of wall time. Can be used for calculations, and to find the "difference" between two date/times. Periods are aware of daylight savings time.

```
Period period = Period.of(1,2,3);
System.out.println(period);
// P1Y2M3D  // 1 year, 2 months, 3 days
System.out.println(LocalDateTime.now().plus(period));
// 2017-03-15T13:04:08.969
System.out.println(Period.between(LocalDate.now(),
    LocalDate.now().plusMonths(1))); // P1M
```

Durations

Durations are like Periods, but time-based instead, and do not take daylight savings time in consideration. For a Duration a day is *always* 24 hours.

```
Duration duration = Duration.ofSeconds(3, 5);
System.out.println(duration); // PT3.000000005S
System.out.println(Duration.between(
    LocalTime.now(), LocalTime.now().plusMinutes(1)));
// PT1M
```

Note that for differences the ChronoUnit enums have a `between(...)` method

```
Instant then = Instant.now();
Thread.sleep(0,50); // 50 nanoseconds
System.out.println(
    ChronoUnit.NANOS.between(then, Instant.now()));
// 1000000
```

(The tick resolution on Linux for Thread.sleep is 1 ms, so this is the minimum time the scheduler will let the sleep last).

Clock

"Most temporal-based objects provide a no-argument `now()` method that use the system clock and the default time zone, *and* a one-argument now(Clock) method that allows you to pass in an alternative Clock."

If for *any* reason you cannot use the clock as-is from the underlying operating system, using a `java.time.Clock` allows you to control it fully. Note there are two kinds, ticking and standing still:

- `Clock.offset(Clock, Duration)` returns a ticking clock that is offset by the specified Duration.
- `Clock.systemUTC()` returns a clock representing the Greenwich/UTC time zone.
- `Clock.fixed(Instant, ZoneId)` always return the same Instant. For this clock, *time stands still*. Useful in tests.

https://docs.oracle.com/javase/tutorial/datetime/iso/clock.html

Chronology + ChronoLocalDate + ChronoLocalDateTime + ChronoZonedDateTime

These classes support the needs of developers using non-ISO calendaring systems.

These classes are there purely for developers who are working on highly internationalized applications that need to take into account local calendaring systems, and they shouldn’t be used by developers without these requirements. Some calendaring systems don’t even have a concept of a month or a week and calculations would need to be performed via the very generic field API.

Date and Time Formatting

Although the java.time.format.DateTimeFormatter provides a powerful mechanism for formatting date and time values, you can also use the java.time temporal-based classes directly with java.util.Formatter and String.format, using the same pattern-based formatting that you use with the java.util date and time classes.

Left as an exercise for the interested reader O:-)

Concurrency

http://www.infoq.com/articles/Java-8-Quiet-Features
http://winterbe.com/posts/2015/05/22/java8-concurrency-tutorial-atomic-concurrent-map-examples/

`StampedLock`: Fast optimistic lock, but if failing (which should be rarely) you will have to redo your work.

`LongAdder`: Faster than AtomicLong (different way to handle contention).

Parallel Sorting: `Arrays.parallelSort(myArray)` distributes over cores. Underlying implementation cannot be tuned, and performance degrades under high load.

Repeatable `@Annotations`

```
@Retention( RetentionPolicy.RUNTIME )
 public @interface Cars {
     Manufacturer[] value() default{};
 }
 @Manufacturer("Mercedes Benz")
 @Manufacturer("Toyota")
 @Manufacturer("BMW")
 @Manufacturer("Range Rover")
 public interface Car { }

 @Repeatable(value = Cars.class )
 public @interface Manufacturer {
     String value();
 };
```

Note: Several `@Manufacturer` annotations on `Car` (silently put in a list).

Static checking using annotations:

The JVM itself does not yet enforce any kind behavior based on annotations on source code.

`javac` and IDE's do!

Note that annotations mentioned in the following may be in different packages and not immediately interchangeable.

IntelliJ 1/2:

Respects the following directly as part of the source analysis:

- `@NotNull` - null value is forbidden to return (for methods) and hold (for local variables and fields).
- `@Nullable` - null value is perfectly valid to return (for methods), pass to (for parameters) and hold (for local variables and methods)
- `@NonNls` - string is not to be internationalized.
- `@Contract` - hint the compiler about input and return values.
- `@ParametersAreNonnullByDefault` - annotate a method once, instead of all parameters with `@NotNull`.

Also respects JSR-305 and FindBugs annotations directly if part of the project. Annotations may be put outside Java source in an annotations.xml file (needs to be configured in the SDK).

IntelliJ 2/2

Null analysis can be done with "Analyze | Infer Nullity" and appropriate annotations added.

Checker framework checks at compile time:

- `@NotNull` - flag if null is being assigned.
- `@ReadOnly` - flag any attempt to change the object.
- `@Regex` - is this string assigned a valid regular expression *string*?
- `@Tainted` + `@Untainted` - avoid mixing data that does not go together like user input being used in system commands, or sensitive data in log statements.
- `@m` - ensure units are dealt with properly.

It may also be able to detect SQL injection.

http://types.cs.washington.edu/checker-framework/

String

- String.substring() does not hold on to underlying char array.
- G1 Garbage Collection deduplicates strings.
- String.join(..) allows for easy concatenation of strings.

String.substring(...)

Note: Important implementation change!

- `String.substring` used the same underlying char array from the original string, so it could not be garbage collected when the original string went out of scope.
- Fixed in Java 1.7.0_06.

http://stackoverflow.com/a/20275133/53897

Note: substring is now O(n) instead of O(1). Discussion on impact at https://www.reddit.com/comments/1qw73v

String.join(...)

New helper method. First argument is separator, remaining arguments are joined with the separator.

```
System.out.println(
    String.join(" ", "Hello", "World")
);
```

prints `Hello World`.

Use `Collectors.joining()` or `StringJoiner` for more advanced cases.

Random numbers

"Java 8 has added a new method called `SecureRandom.getInstanceStrong()` whose aim is to have the JVM choose a secure provider for you. "

Exact Math

"In cases where the size is int or long and overflow errors need to be detected, the methods `addExact`, `subtractExact`, `multiplyExact`, and `toIntExact` throw an `ArithmeticException` when the results overflow. "

```
Math.multiplyExact(1_000_000, 1_000_000);
// Exception in thread "main"
//    java.lang.ArithmeticException: integer overflow
```

http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

Process

Three new methods in the Process class -

- `destroyForcibly()` - terminates a process with a much higher degree of success than before.
- `isAlive()` tells if a process launched by your code is still alive.
- A new overload for `waitFor()` lets you specify the amount of time you want to wait for the process to finish. This returns whether the process exited successfully or timed-out in which case you might terminate it.

Interfaces:

- static methods
- default methods

Interfaces can now hold code. This helps adding functionality to existing API's without breaking backward compatibility.

Overview:
http://www.studytrails.com/java/java8/java8_collections_new_methods.jsp

Interfaces - static methods:

Works like for classes. They are available directly without having to instantiate a class (which is beneficial for λ-expressions).

```
System.out.println(Function.identity().apply(42));
// 42
```

Many new static methods on Comparator, making it much easier to create a new comparator without having to code the full `compare(..)` method.

Of course this can be abused...

```java
public interface Test {
    static void main(String[] args) {
        System.out.println("I'm ok!");
    }
}
```

http://stackoverflow.com/q/34710274/53897

Interfaces - default methods:

Pre-Java 8 it was impossible to add new methods to interfaces without breaking existing code as these new methods needed to be implemented too.

A `default` method has an implementation directly in the interface, and does not *have* to be implemented (but may) when instantiated.

```
    List<String> l = Arrays.asList("abc", "Bc", "a");
    l.sort(Comparator.naturalOrder());
    System.out.println(l); // [Bc, a, abc]
```

Spot the two methods that wasn't possible in Java 7.

The implementation in `java.util.List<E>` looks like:

```
default void sort(Comparator<? super E> c) {
    ....
}
```

1. This allows restricted multiple inheritance (see Scala Traits)
2. Implementing multiple interfaces defining the exact same default method is a compilation error.
3. You cannot define variables in an interface so you cannot keep state in default methods except by passing in a state keeping object.

Comparators

Boring - all blogs/torturials use this as *the* example.

Tons of new helper methods! See how easy it is to sort strings by *length*:

```
Comparator c = Comparator.comparing(
                   new Function<String, Integer>() {
                       @Override
                       public Integer apply(String s1) {
                           return s1.length();
               }})); // <- Ooh, LISP!
```

See http://stackoverflow.com/a/24442897/53897 and http://blog.jooq.org/2014/01/31/java-8-friday-goodies-lambdas-and-sorting/

Optional - defusing `null` values

Tired of checking for `null` in return values?

A `java.util.Optional<T>` either holds exactly one T instance, or is empty.

```
System.out.println(Optional.of("!").isPresent());
// true
System.out.println(Optional.empty().isPresent());
// false
System.out.println(Optional.ofNullable(null).isPresent());
// false
```

Best practice now for methods which return `T` which may be `null` is to return `Optional<T>`. For legacy methods, wrap in `ofNullable`.

Methods:

- `empty()` - return empty Optional instance.
- `filter(Predicate)` - if value is present and matches predicate return Optional describing value else an empty Optional.
- `flatMap(Function)` - if value is present, apply function to value, else return empty Optional.
- `get()` - gets the element or throw `NoSuchElementException`
- `ifPresent(Consumer)` - if value is present, invoke consumer with value, otherwise do nothing.
- `isPresent()` - returns true if value present, false otherwise.
- `of(T)` - throw exception if value==null, or return Optional value.
- `ofNullable(T)` - returns optional with value if non-null, otherwise an empty Optional.
- `orElse(T)` - return value held if present, otherwise return argument.
- `orElseThrow(Supplier<X>)` - return value if present, otherwise throw the exception created by the provided supplier.

Deep breath!

```
drwxr-xr-x   2 root r     48 2005-     16:59 bin
drwxr-xr-x   3 root      024 2005-11-27 16:42 boot
lrwxrwxrwx   1 root       11 2005-11-27 16:29 cdrom -> media/cdrom
drwxr-xr-x  11 root      576 2005-11-27 16:57 dev
drwxr-xr-x  52 ro       3072 2005-11-27 17:01 etc
drwxrwxr-x   4 roo      1024 2005-11-27 16:46 home
drwxr-xr-x   2 root root 1024 2005-11-27 16:32 initrd
lrwxrwxrwx   1 root root   28 2005-11-27 16:41 initrd.img -> boot/initrd.i
.4.27-2-386
drwxr-xr-x   8 root root  4096 2005-11-27 16:56 lib
drwxr-xr-x   2 root root 12288 2005-11-27 16:29 lost+found
```

It all comes together...

- λ-expressions
- Streams
- java.util.function.*

Because sometimes, you need a rainbow butterfly unicorn kitten.

λ-expressions:

λ-expressions allows for implementing interfaces much more concisely than anonymous classes.

λ-calculus is a formal system in mathematical logic by Alonzo Church for expressing computation based on function abstraction and application using variable binding and substitution. Lambda calculus is a universal model of computation equivalent to a Turing machine. λ is used in lambda terms (also called lambda expressions) to ***denote binding a variable in a function****.*

tl;dr - nicer syntax on anonymous classes.

Three parts:

```
(String s) -> "Hello " + s
```

- Zero or more comma-separated variable definitions in parenthesis.
- `->`
- Statement or block to invoke. Return type deduced by compiler.

λ-expressions are *declarations*, not *invocations*. Actually invoking the code must be done "outside" the λ-expression itself.

A single statement:

```
() -> 42
() -> null

(int x) -> x + 1
(String s) -> "Hello " + s

(int x, String s) -> x + " " + s

(int a, int b, int c) -> a + b + c
```

Again: Zero or more comma separated variable definitions in parenthesis, `->`, and an expression to be evaluated.

Compiler deduces return type from invocation context (may be enforced by type casting the λ-expression)

Block:

```
() -> { System.out.println(
            System.currentTimeMillis()
        ) }


(String s) -> { log.debug("{}", s);
                return s;
              }
```

Block is surrounded by `{` and `}` and contains zero or more statements. `return` statements determines return type. If there is no `return` statement, it corresponds to a `void` method.

Inferred parameter types:

The parameter type(s) may be explicitly mentioned:

```
(String s) -> "Hello " + s
```

The compiler may be able to deduce the parameter type(s) from the context, and then it may be omitted:

```
(s) -> "Hello " + s
```

If there is only one parameter and its type is inferred, the parenthesis is optional:

```
s -> "Hello " + s
```

Examples:

```
(x) -> x + 1
(s) -> "Hello " + s
(a, b, c) -> a + b + c


x -> x + 1
s -> "Hello " + s
s -> { log.debug("{}", s); return s; }
```

The compiler makes an effort to infer the parameter types and the result type from the surrounding context. It may give up, and the parameters then have to be explicitly typed or a typecast applied to the whole λ-expression.

Typecasting a λ-expression:

```
Comparator.comparing((Function<String, Integer>) (s1) -> {
   return s1.length();
}));
```

Here `(Function<String, Integer>)` is the typecast. This also allows the compiler to infer the type of `s1`.

IntelliJ uses this for refactorings to preserve exact meaning.

Scope:

λ-expressions use lexical scope in the same way as a normal `{}`-delimited block, so they are allowed to refer to variables outside the λ-expression itself if they are "final or effectively final".

"Effectively final variables" mean variables where "*declaring it final would not cause a compilation failure.*"

Variables may be overridden if needed.

`this` is unchanged inside the lambda, and does *not* refer to the lambda itself but the surrounding object!

http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html

Functional Interface underneath:

Regardless where a λ-expression is used, it `<blink>`*must*`</blink>` implement an interface with only one abstract method! This is officially called a *Functional Interface*.

A λ-expression may only throw the checked exceptions declared in the interface implemented.

The JRE has functional interfaces with up to two parameters (including native types). None of these allow throwing checked exceptions. This is important with Streams which use these.

Word of God: Write exception wrappers.

Method references

Some simple λ-expressions may be abbreviated even further with the new method reference syntax.

```
context::method
```

These two are identical:

```
e -> System.out.println(e)
System.out::println
```

Reference to a constructor:

```
e -> new Double(e)
```

can be abbreviated as

```
Double::new
```

Type information may be provided:

```
HashSet<String>::new
```

Reference to a an instance method of a type

`Class::instanceMethod` can be used to refer to the instanceMethod on an instance of Class. These two are identical:

```
(String e) -> e.length()
String::length
```

Example:

```
List<String> l = Arrays.asList("abc", "Bc", "a");
l.sort(Comparator.comparing(String::length));
System.out.println(l); // [a, Bc, abc]
```

Reference to a static method:

Use `myObject::myStaticMethod`.

Reference to an instance method on a specific variable:

Example:

```
Optional.of("!").ifPresent(System.out::println);
```

.

Again, `System.out::println` is shorthand for `e -> System.out.println(e)`.

IntelliJ 15+:

- Convert method reference to lambda expression and back.
- Convert anonymous type to method reference.
- Add inferred lambda parameter types.
- Shows javadoc for method implemented with cursor on `->` or `::` and Ctrl-Q.
- Add type and parenthesis to single inferred parameter - `s->{}` -> `(String s)->{}`
- Debugger supports lambda expressions inside chained method calls (like streams).

Eclipse 4.5+ Ctrl-1:

- Convert anonymous class to lambda expression and back.
- Convert method reference to lambda expression and back.
- Add inferred lambda parameter types.
- Remove/add parenthesis around single inferred parameter.
- Convert lambda expression body from expression to block and back.
- View method implemented by hovering mouse on `->` or `::`. Use Ctrl-hover to navigate to declaration.
- Debugger supports lambda expressions.

@FunctionalInterface

Interfaces *intended* to be functional interfaces - allowing λ-expressions to implement them - can explicitly be annotated with `@FunctionalInterface`. If so, compilers are required to check:

- Is an interface (and not something else)
- Has exactly *one* abstract method.

However, the compiler will treat ***any interface meeting the definition of a functional interface as a functional interface*** regardless of whether or not a `@FunctionalInterface` annotation is present on the interface declaration.

`java.*` functional interfaces in the JRE

- java.awt.KeyEventDispatcher
- java.awt.KeyEventPostProcessor
- java.io.FileFinder
- java.io.FilenameFilter
- java.lang.Runnable
- java.lang.Thread.UncaughtExceptionHandler
- java.nio.file.DirectoryStream.Filter
- java.nio.file.PathMatcher
- java.time.temporal.TemporalAdjuster
- java.time.temporal.TemporalQuery
- java.util.Comparator
- java.util.concurrent.Callable

- java.util.function.BiConsumer
- java.util.function.BiFunction
- java.util.function.BinaryOperator
- java.util.function.BiPredicate
- java.util.function.BooleanSupplier
- java.util.function.Consumer
- java.util.function.DoubleBinaryOperator
- java.util.function.DoubleConsumer
- java.util.function.DoubleFunction
- java.util.function.DoublePredicate
- java.util.function.DoubleSupplier
- java.util.function.DoubleToIntFunction
- java.util.function.DoubleToLongFunction
- java.util.function.DoubleUnaryOperator

- java.util.function.Function
- java.util.function.IntBinaryOperator
- java.util.function.IntConsumer
- java.util.function.IntFunction
- java.util.function.IntPredicate
- java.util.function.IntSupplier
- java.util.function.IntToDoubleFunction
- java.util.function.IntToLongFunction
- java.util.function.IntUnaryOperator

- java.util.function.LongBinaryOperator
- java.util.function.LongConsumer
- java.util.function.LongFunction
- java.util.function.LongPredicate
- java.util.function.LongSupplier
- java.util.function.LongToDoubleFunction
- java.util.function.LongToIntFunction
- java.util.function.LongUnaryFunction
- java.util.function.ObjDoubleConsumer
- java.util.function.ObjIntConsumer
- java.util.function.ObjLongConsumer

- java.util.function.Predicate
- java.util.function.Supplier
- java.util.function.ToDoubleBiFunction
- java.util.function.ToDoubleFunction
- java.util.function.ToIntBiFunction
- java.util.function.ToIntFunction
- java.util.function.ToLongBiFunction
- java.util.function.ToLongFunction
- java.util.function.UnaryOperator
- java.util.logging.Filter
- java.util.prefs.PreferenceChangeListener

Note that the many variants with Long/Int/Double/Binary is to support native non-object types. If you only use objects you can ignore them. The Bi-prefix means two arguments instead of one.

Streams: Iterators on steroids

Streams - an new Java 8 API, for handling multiple items one item at a time, resulting in an answer.

- Map-reduce with LOTS of syntactic sugar.
- Lazy evaluation.
- Parallel/sequential processing in the current JVM.
- Looks like functional programming and SQL.
- But isn't.
- But it looks like!

It is enough to get actual work done! The API is so rich that I only look at a subset of features.

Map-Reduce:

Handle a large input set consisting of similar objects, by:

1. Decide if running sequentially or in parallel.
2. Specify what to do with each input element. ("map" input element to output element)
3. Specify how to have each output element contribute to the final result. ("reduce")

An example could be: "given a list, square each element and return their sum".

The framework then distributes out the input elements, let the mappers work, collects the results and reduce them to the final result.

There is more to it than that but we work in-memory in a single JVM

Stream recipe:

Do this for every stream you need. Streams cannot be reused.

1. Start with a `Collection<T>`.
2. Get `Stream<T>` with either `.stream()` or `.parallelStream()`
3. Apply zero or more transformations like `.map(...)`
4. Reduce the stream to the final result with e.g. `.collect(Collectors.toList())`.

```
Arrays.asList(1,2,3,4).stream() // Stream<Integer>
      .map(e -> e + 1)
      .collect(Collectors.toList()) // [2, 3, 4, 5]
```

There are many helper methods to make 1+2 easier.

Beware of autoboxing which happens here!!

"given a list, square each element and return their sum"

```
Arrays.asList(1,2,3,4).stream()      // Stream<Integer>
      .mapToInt(Integer::intValue)  // IntStream
      .map(e -> e * e)
      .sum() // 30
```

Autoboxing is avoided by using `mapToInt(...)` to convert from `Stream<Integer>` to `IntStream`! Additionally `IntStream` has the very handy `sum()` method returning the sum of all the elements in the stream.

Native streams:

```
IntStream.of(1, 2, 3, 4).map(e -> e * e).sum()      // 30
IntStream.range(1, 5).map(e -> e * e).sum()         // 30
LongStream.rangeClosed(1, 4).map(e -> e * e).sum() // 30

IntStream si = Arrays.stream(new int[] {1, 2, 3});
LongStream sl = Arrays.stream(new long[] {1, 2, 3});
DoubleStream sd = Arrays.stream(new double[] {1, 2, 3});
```

DoubleStream hasn't range/rangeClose, but all the rest.

Stream methods returning a Stream:

In other words, these can be chained.

- `concat(Stream1, Stream2)` - the concatenation of two streams
- `distinct()` - remove duplicates.
- `filter(Predicate)` - remove those returning false.
- `flatMap(Function)` - map returning element_S_. More later.
- `limit(maxSize)` - do not return more than maxSize elements.
- `iterate(seed, function)` - element(n+1) = function(element(n))
- `map(function)` - apply/map function
- `of(...)` - stream of the zero or more values provided.
- `peek(...)` - debug stream. More later.
- `skip(n)` - discard the first n elements of the stream.
- `sorted()` - sort all elements.

https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

```
import static java.util.stream.Collectors.toList;

Stream.concat(Stream.of(1,2),Stream.of(3,4))
      .collect(toList()) // [1, 2, 3, 4]

Stream.of(1,1,2).distinct().collect(toList()) // [1, 2]

Stream.of(1,2).filter(e->e!=1).collect(toList())); // [2]
```

peek(...) - debug tool

```
Stream.of(2, 1, 2)
   .peek(e -> System.out.println("before " + e))
   .filter(e -> e != 1)
   .peek(e -> System.out.println("after " + e))
   .collect(Collectors.toList());  // [2, 2]
```

```
before 2
after 2
before 1
before 2
after 2
```

Lazy evaluation cause one element at a time to "flow through" the stream as needed. Only execute those steps needed to get a result!

flatMap() - zero or more return values

`map` only allows one result. `flatMap` allows zero or more, all of which are put into the result stream.

```
Arrays.asList(
  Arrays.asList("e", "d", "a"), Arrays.asList("c", "b")
) // [[e, d, a], [c, b]]
.stream()
.flatMap(
    e -> e.stream().sorted()        // [a, d, e] and [b, c]
).collect(Collectors.toList())      // [a, d, e, b, c]
```

Inspired by http://www.adam-bien.com/roller/abien/entry/java_8_flatmap_example

Stream termination to get result:

- `.reduce(...)` - hardcore functional programming approach
- `.collect(collector)` - heaps of syntactic sugar on top of `reduce`
- `findAny/findFirst/max/min` - a single element
- `count()/`
- `forEach(consumer)` - apply consumer to each element
- `allMatch(...)/anyMatch(...)/noneMatch(...)` - find characteristic
- `toArray()` - return array of all elements in stream

Remember, streams cannot be reused.

Selecting a single element from a Stream:

- `findAny()` - any element, we don't care which
- `findFirst()` - first element
- `max(comparator)` - largest element according to comparator.
- `min(comparator)` - smallest element according to comparator.

Returns `Optional<T>` as the selection may fail.

```
Stream.of(1, 2, 3).findFirst().orElseGet(() -> 0) // 1
Stream.of().findFirst().orElseGet(() -> 0) // 0
```

Slightly different methods for IntStream/LongStream/DoubleStream.

Collector

- Goes inside `.collect(..)`
- Many helper methods in java.util.stream.Collectors.
- Most of the time you want `Collectors.toList()`.

```
List<String> la = Arrays.asList("ad", "bc", "ba", "ac");

la.stream().collect(Collectors.toList())
// [ad, bc, ba, ac]

la.stream().collect(Collectors.joining("/"))
// ad/bc/ba/ac

la.stream().collect(
    Collectors.toCollection(() -> new TreeSet<String>())
) // [ac, ad, ba, bc]
```

Collect into a Map:

```
la.stream().collect(
  Collectors.groupingBy(e -> e.substring(0, 1))
) // {a=[ad, ac], b=[bc, ba]}

Map<String, String> m = la.stream().collect(
  Collectors.groupingBy(
      e -> e.substring(0,1), Collectors.joining("+")
  )
); // {a="ad+ac", b="bc+ba"}

Arrays.toString(m.entrySet().toArray())
// [a=ad+ac, b=bc+ba]
```

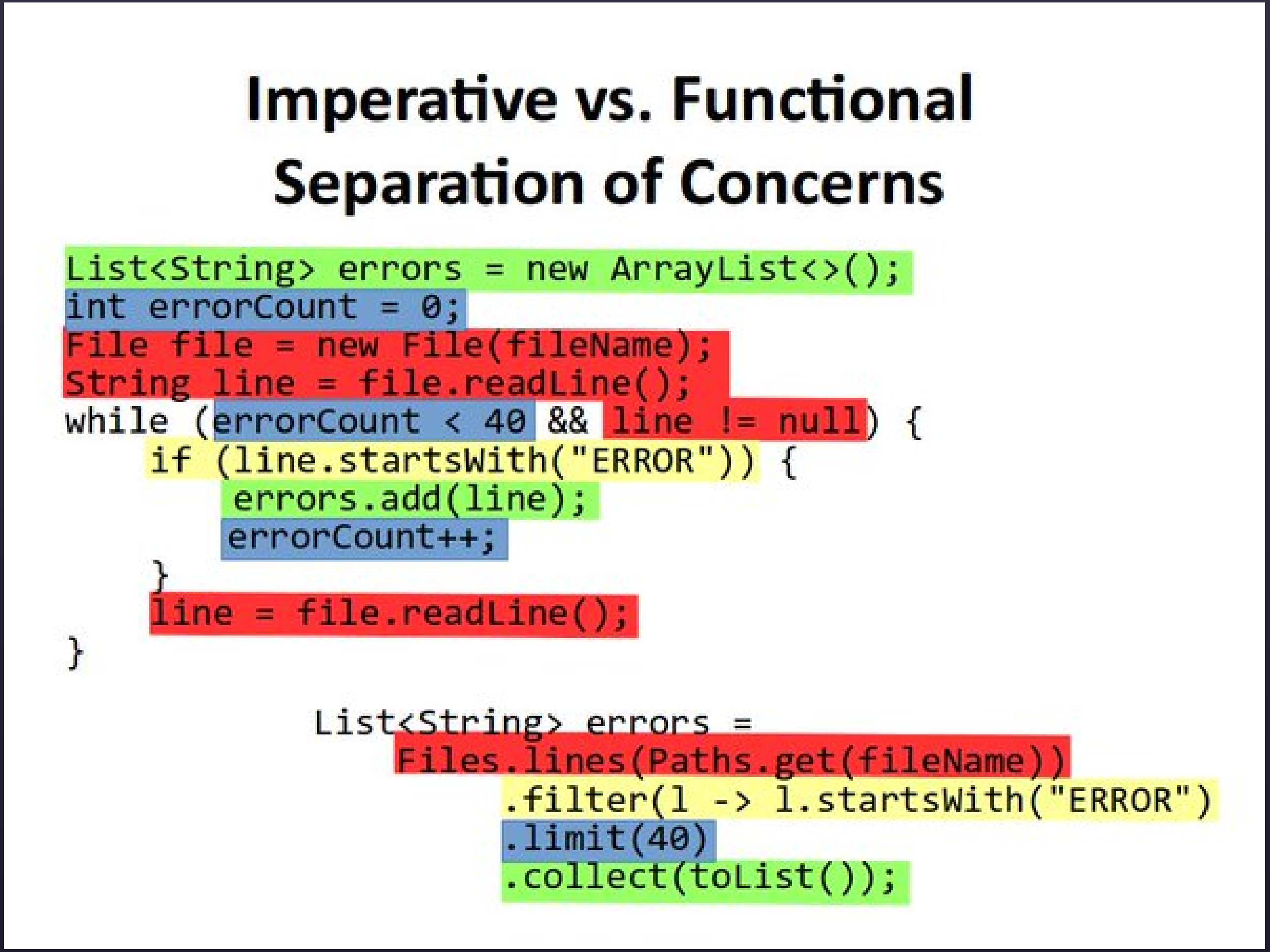
The last line is a nice trick to pretty-print a map.

Reading files as streams:

```
new BufferedReader(...).lines()
```

```
Files.lines(Path.get("..."), Charset.defaultCharSet())
```

Remember, the method in the functional interfaces used by streams do not throw exceptions. For files you will typically need to catch IOExceptions and wrap them in a RuntimeException.



Imperative vs. Functional
Separation of Concerns

```
List<String> errors = new ArrayList<>();
int errorCount = 0;
File file = new File(fileName);
String line = file.readLine();
while (errorCount < 40 && line != null) {
    if (line.startsWith("ERROR")) {
        errors.add(line);
        errorCount++;
    }
    line = file.readLine();
}
```

```
            List<String> errors =
                Files.lines(Paths.get(fileName))
                    .filter(l -> l.startsWith("ERROR")
                    .limit(40)
                    .collect(toList());
```

With a functional approach code tends to group together more.

http://blog.jooq.org/2015/08/13/common-sql-clauses-and-their-equivalents-in-java-8-streams/
(rant, but it has this nice figure)

Parallel streams:

Use `.parallelStream()` instead of `.stream()` to enable parallel processing in current JVM. Very simple optimization. Unfortunately has some overhead. Do not use uncritically!

Uses global ForkJoinPool:

- Avoid long processing in multithreaded applications.
- Untunable thread pool.

Use unordered streams if possible.

Rants: https://dzone.com/articles/think-twice-using-java-8 https://dzone.com/articles/whats-wrong-java-8-part-iii

Functional programming?

- No branching in stream. (no zip/unzip)
- Java only allow single return value (no tuples)
- More than two arguments require writing more `@FunctionalInterfaces`, and currying is tedious.
- No pattern matching :)

Essentially Java 8 now have an API for applying multiple operations to collection elements lazily one at a time with the loop operations in library code, instead of explicitly having to write them in user code. And just that. It is a start though!

TED
IDEA.
...WHAT DOES IT DO?
THAT'S THE PROBLEM. WE HAVEN'T FIGURED OUT HOW TO MAKE IT DO WHAT WE WANT.
DON
MON
RES

Credits:

Presentation written in Markdown and rendered with https://github.com/jsakamoto/MarkdownPresenter.

- Use line with `!` for "New slide"
- Small, stand-alone webserver.
- While editing, press Space to reload slide content.
- Instant feedback with IntelliJ markdown plugin
- No code syntax coloring. May be very simple to add.

See https://github.com/ravn/java8-presentation

The End!