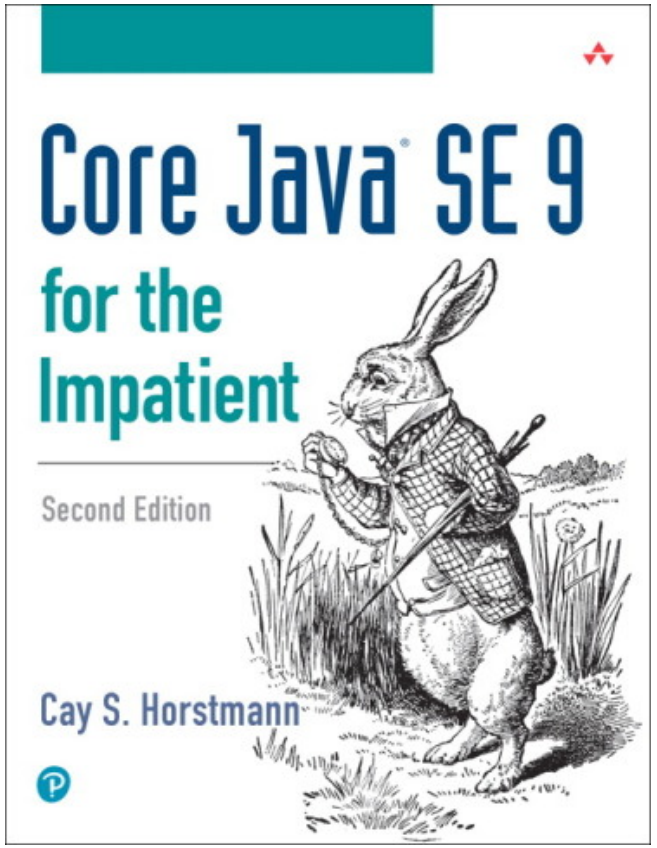
Java 9: The Good Parts (Not Modules)



- Cay Horstmann
- Author of Core Java (10 editions since 1996)

Outline

- Collection literals (almost)
- Elvis operator (almost)
- New sightings of streams...
- ...and completable futures
- The interactive shell
- Improvements in input, regex, processes, optionals, streams, big numbers
- Miscellaneous trivia
- Bonus: Pop quizzes
- Nothing depressing (i.e. not modules)

Collection Literals (Almost)

- In 2009, Josh Bloch proposed collection literals for Java 7 as part of Project Coin:

```
List<Integer> piDigits = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 9];
Set<Integer> primes = { 2, 7, 31, 127, 8191, 131071, 524287 };
Map<Integer, String> platonicSolids = { 4 : "tetrahedron", 6 : "cube",
   8 : "octahedron", 12 : "dodecahedron", 20 : "icosahedron" };
```

- This is what we are getting for Java 9:

```
List<Integer> piDigits = List.of(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 9);
Set<Integer> primes = Set.of(2, 7, 31, 127, 8191, 131071, 524287);
Map<Integer, String> platonicSolids = Map.of(4, "tetrahedron", 6, "cube",
   8, "octahedron", 12, "dodecahedron", 20, "icosahedron");
```

or

```
import static java.util.Map.*;
platonicSolids = ofEntries(entry(4, "tetrahedron"), entry(6, "cube"), ...);
```

Pop Quiz #1

What does this code print?

```
Map<String, Integer> months = Map.of("Jan", 1, "Feb", 2, "Mar", 3, "Apr",
4,   "May", 5, "Jun", 6, "Jul", 7, "Aug", 8, "Sep", 9, "Oct", 10, "Nov", 11, "Dec", 12);
System.out.println(map.values());
```

1. {Jan=1, Feb=2, Mar=3, ..., Dec=12}
2. {Jan=1, May=5, Aug=8, ..., Feb=2}
3. {Apr=4, Aug=8, Dec=12, ... Sep=9}
4. Nothing—the code does not compile.

Pop Quiz #1 Solution

- How are these `of` methods implemented?
- `static <E> List<E> of(E... elements)`
- But that doesn't work for `Map<K, V>.of`
- You can't have varargs with intermingled types `K V ...`
- 11 methods

```
Map.of()
Map.of(K, V)
Map.of(K, V, K, V)
...
```

- If the map has ≥ 11 entries, must use `Map.ofEntries(Map.Entry ...)`
- By the way, there are also 11 methods `List.of()`, `List.of(E)`, `List.of(E, E)`, ...
- To avoid the cost of packing the arguments into a varargs array.

Pop Quiz #2

```
void printSortedWithJava(String[] strings)
{
   List<String> list = Arrays.asList(strings); // Version 1
                     = List.of(strings); // Version 2
   if (!list.contains("Java")) list.add("Java");
   Collections.sort(list);
   System.out.println(list);
}
```

Which of the following is true?

1. Neither version runs correctly for any array
2. Version 1 runs correctly for all arrays containing "Java" but version 2 doesn't.
3. Version 2 runs correctly for all arrays containing "Java" but version 1 doesn't.
4. Both versions run correctly for all arrays

Pop Quiz #2 Solution

```
void printSortedWithJava(String[] strings)
{
   List<String> list = Arrays.asList(strings); // Version 1
                     = List.of(strings); // Version 2
   if (!list.contains("Java")) list.add("Java");
   Collections.sort(list);
   System.out.println(list);
}
```

- `List.of(...)` is immutable.
- So, clearly it can't work—you can't sort it.
- Throws an `UnsupportedOperationException`
- What if the list is already sorted?
- `Collections.sort` calls `List.sort` (added in Java 8), and `AbstractImmutableList.sort` throws an UOE.
- What about `Arrays.asList(...)`?
- You *can* sort those!
- But they are “fixed size”—you can't resize them.
- In 1998, immutability was still a strange concept.

Bonus Feature–Pairs

- Java has no generic `Pair<T, S>` type
- ...because we are waiting for Project Valhalla
- Pairs and tuples are more efficient as value types
- What about `Map.Entry<T, S>` as the poor man's pair?

```
Map.Entry<String, String> minmax(String[] values)
```

- It's an interface (since Java 2)
- Java 6 adds implementing classes `AbstractMap.SimpleEntry`, `AbstractMap.SimpleImmutableEntry`

```
return new AbstractMap.SimpleImmutableEntry<>(lowest, highest); // Ugh
```

- Hooray for Java 9!

```
return Map.entry(lowest, highest)
```

- Surely nobody would use a `Map.Entry` for a pair, right?
- These folks do...

Elvis Operator (Almost)

- Groovy:

```
name = person.name ?: "John Q. Public";
```

- Java 9:

```
name = Objects.requireNonNullElse(person.name, "John Q. Public");
```

- But that's not all, folks!
- Can compute alternative lazily:

```
name = Objects.requireNonNullElseGet(person.name,
   () -> System.getProperty("default.name"));
```

- This is all done to make `Optional` look good:

```
Optional<String> maybeName = person.name(); // Returns Optional
name = maybeName.orElse("John Q. Public");
```

Optional **News**

- New API methods yielding optional values:
  - `HttpClient.authenticator/cookieManager/proxy/sslParameters, ServiceLoader.findFirst, Runtime.Version.build/pre/post, ProcessHandle.of, ProcessHandle.Info.command/commandLine/arguments/startInstant/totalCpuDuration/user`

- New methods:

```
Optional<T>       or(Supplier<? extends Optional<? extends T>> supplier)
```

- ```
  maybeName = maybeName.or(() ->
      johnQPublic.name()); // Another Optional
  ```
  - Is `maybeName` nonempty? If so, `or` returns it.
  - Otherwise, call the supplier to get another `Optional`
- ```
  maybeName.ifPresentOrElse(System.out::println, // Consumer
     () -> Runtime.exec("rm -rf /home")); // Runnable
  ```
- ```
  Stream<T> stream()
  ```

  Yields a stream of length 0 or 1.

- Use with `flatMap` to drop empty results:

```
Stream<User> users = people.map(Person::name)
    .flatMap(Optional::stream);
```
```

Bonus–Optional Cheat Sheet

- Grab the data:

```
data = opt.orElse(defaultValue);
data = opt.orElseGet(() -> aValue);
data = opt.orElseThrow(SomeException::new);
```

- If/then/else with lambdas:

```
opt.ifPresent(value -> ...);
opt.ifPresentOrElse(value -> ..., () -> ...);
```

- Transform/substitute:

```
anotherOpt = opt.filter(value -> ...).map(value -> ...).or(() -> ...);
```

Stream News

- New API methods yielding streams:
  - `Scanner.tokens`, `Matcher.results`, `ServiceLoader.stream`, `LocalDate.datesUntil`, `StackWalker.walk`, `ClassLoader.resources`, `Process.children/descendants`, `Catalog.catalogs`, `DriverManager.drivers`
  - Common theme: laziness
- `Files.lines` now uses a memory mapped file
  - Efficient with parallel stream
- New stream methods `takeWhile`, `dropWhile`
- Finite stream iterator:

```
BigInteger limit = new BigInteger("10000000");
Stream<BigInteger> integers
    = Stream.iterate(BigInteger.ZERO,
        n -> n.compareTo(limit) < 0,
        n -> n.add(BigInteger.ONE));
```

- New collectors `flatMapping`, `filtering`:

```
Map<String, Set<City>> largeCitiesByState
    = cities.collect(
        groupingBy(City::getState,
            filtering(c -> c.getPopulation() > 500000,
                toSet())));
```

  - Why not just `cities.filter(...).collect(...)`?
  - Gives empty sets for states without large cities.

I/O and Regex Improvements

- After all these years, there is finally a method to read all bytes from an input stream:

```
byte[] bytes = Files.newInputStream(path).readAllBytes();
```

There is also `readNBytes`.

- `InputStream.transferTo(OutputStream)` transfers all bytes from an input stream to an output stream.
- `Scanner.tokens` gets a stream of tokens, similar to `Pattern.splitAsStream` from Java 8:

```
Stream<String> tokens = new Scanner(path).useDelimiter("\\s*,\\s*").tokens();
```

- `Matcher.stream` and `Scanner.findAll` gets a stream of match results:

```
Pattern pattern = Pattern.compile("[^,]");
Stream<String> matches = pattern.match(str).stream().map(MatchResult::group);
matches = new Scanner(path).findAll(pattern).map(MatchResult::group);
```

- `Matcher.replaceFirst/replaceAll` now have a version with a replacement function:

```
String result = Pattern.compile("\\pL{4,}")
    .matcher("Mary had a little lamb")
    .replaceAll(m -> m.group().toUpperCase());
    // Yields "MARY had a LITTLE LAMB"
```

Process Handles

- Get `ProcessHandle` as
  - `process.toHandle()` from a `Process process`
  - `ProcessHandle.of(pid)` from an OS PID
  - `ProcessHandle.current()`
  - `ProcessHandle.allProcesses()`
- Get OS process information:

```
long pid = handle.pid();
Optional<ProcessHandle> parent = handle.parent();
Stream<ProcessHandle> children = handle.children();
Stream<ProcessHandle> descendants = handle.descendants();
ProcessHandle.Info info = handle.info();
```

- `ProcessHandle.info` yields `command/commandLine/arguments/startInstant/totalCpuDuration/user` if available
- The `onExit` method yields a `CompletableFuture<ProcessHandle>` that completes when the process is completed:

```
Process process = new ProcessBuilder("/bin/ls").start();
process.onExit().thenAccept(h -> System.out.println(h + " all done"));
```

HttpClient

- Intended to replace fussy `URLConnection/HttpURLConnection`
- In `jdk.incubator` package
  - Run with `--add-modules jdk.incubator.httpclient`
- Build a client:

```
HttpClient client = HttpClient.newBuilder()
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build();
```

- Build a request:

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI("http://horstmann.com"))
    .GET()
    .build();
```

- Get and handle response:

```
HttpResponse<String> response = client.send(request, HttpResponse.BodyHandler.asString());
```

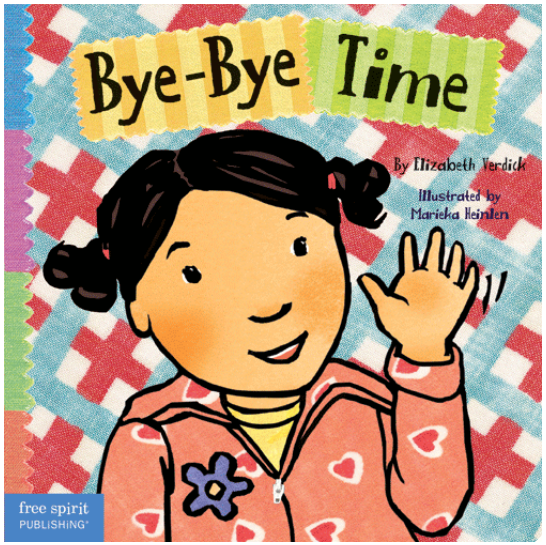
- Asynchronous processing:

```
client.sendAsync(request, HttpResponse.BodyHandler.asString())
   .completeOnTimeout(errorResponse, 10, TimeUnit.SECONDS)
   .thenAccept(response -> ...);
```

- Still work in progress—no support for application/x-www-form-urlencoded, multipart/form-data, JSON body handler, etc.
- BTW, `CompletableFuture.completeOnTimeout` is new in Java 9

Deprecation

- `@Deprecated` has `since` and `forRemoval` attributes:
- Deprecated `java.util` classes `Observable`, `Observer` (but not `Vector`, `Hashtable`)
- `Object.finalize`, `Runtime.runFinalizersOnExit` (!)
- `Class.newInstance`
  - Because it rethrows checked constructor exceptions without declaring them (!)
  - Use `Constructor.newInstance` which wraps them in an `InvocationTargetException`
- Applets
- Modules `java.activation`, `java.corba`, `java.transaction`, `java.xml.bind`, `java.xml.ws`
- Run `jdeprscan` to get deprecated uses in your code



Minor Language Changes

- `_` is not a valid variable name.
- `try`-with-resources can be used with an effectively final variable:

```
void print(PrintWriter out, String[] lines) {
    try (out) { // Effectively final variable
        for (String line : lines)
            out.println(line.toLowerCase());
    }
}
```

- Interfaces can have concrete `private` and `private static` methods.

Propeller Head Stuff

- Can use `@SafeVarargs` on `private` methods:

```
@SafeVarargs private <T> void process(T... values)
```

  - Unsafe vararg can create array of generic type
  - Previously, annotation was restricted to `static` and `final` methods and constructors

- Can use diamond with anonymous classes provided the inferred type is “denotable”:

```
List<String> alwaysJava = new ArrayList<>() {
   public String get(int n) { return "Java"; }
};
```

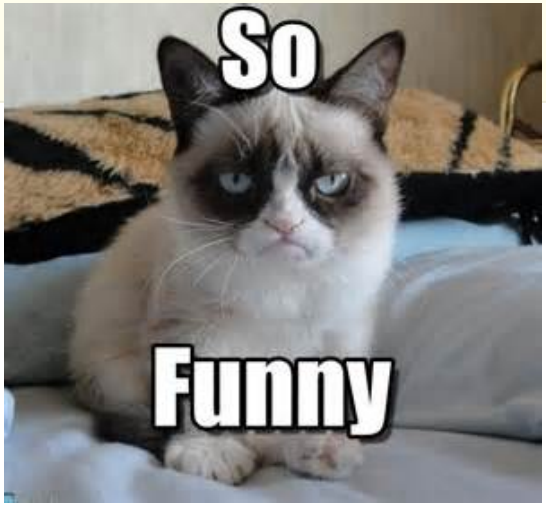
- A good number of obscure compiler errors are fixed. For example, this should compile and now does:

```
<X> int keyValueMatches(Map<X, ? extends X> arg) {
   int count = 0;
   for (X f : arg.keySet())
      if (f.equals(arg.get(f))) count++;
   return count;
}

abstract class Table<T> implements Map<T,T> {}
void matches(Table<?> arg) { keyValueMatches(arg); }
```

Pop Quiz #3

```
@interface Funny {
    int value(Funny this);
}
```

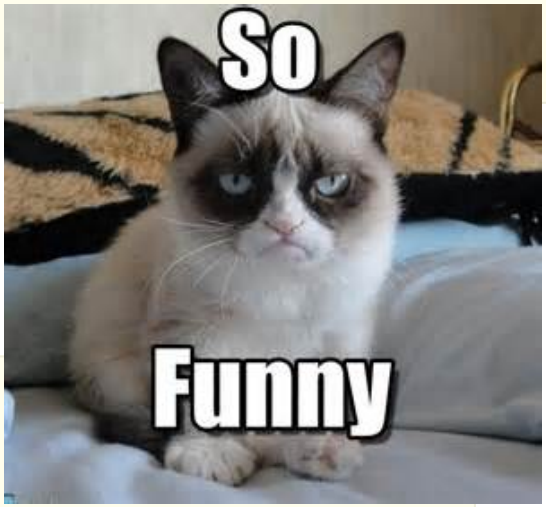


What's funny about this?

1. The parameter should be named something other than `this`
2. The method should be named something other than `value`
3. Nothing—it's legal Java
4. Java 8 accepted this and Java 9 doesn't

Pop Quiz #3 Solution

```
@interface Funny {
    int value(Funny this);
}
```



- Can you have method parameters called `this`?
- Yes, since Java 8!
- Rarely used feature for annotating the implicit parameter:

```
public class Person implements Comparable<Person> {
   public boolean compareTo(@Alive Person this, @Alive Person other) { ... }
   ...
}
```

- Can you have annotation methods with parameters?
- No—annotation methods yield the annotation values
- Code should have never compiled in Java 8
- But it did
- Fixed in Java 9

Pop Quiz #4

```
Object obj = Arrays.asList(1, 2, 3).toArray();
```

What is the type of `obj`?

1. `int[]`
2. `Integer[]`
3. `Object[]`
4. `Integer[]` in Java 8, `Object[]` in Java 9

Pop Quiz #4 Solution

- `Arrays.asList(1, 2, 3)` is a `List<Integer>`
- So, surely the answer is `Integer[]`
- But `List<E>.toArray()` is declared to return an `Object[]`
- Up to Java 8, `Arrays.asList(1, 2, 3).toArray()` did return an `Integer[]` array
- It's not wrong—`Integer[]` is a subtype of `Object[]`
- But what if someone understood the spec to mean that the following should be possible?

```
Object[] result = Arrays.asList(1, 2, 3).toArray();
result[0] = new Watermelon();
```

- Now they can. But the following now fails:

```
Integer[] result = (Integer[]) Arrays.asList(1, 2, 3).toArray();
```

- Remedy:

```
Integer[] result = Arrays.asList(1, 2, 3).toArray(new Integer[0]);
   // Maybe toArray(Integer[]::new) in the future?
```

Version Stuff

- Shocking news: `System.getProperty("java.version")` returns `"9"`, not `"1.9.0"`
- `javac -release 8` compiles for Java 8
  - Previously required `-source`, `-target`, `-bootclasspath`
- Multi-Release JARs


```
A.class
B.class
C.class
META-INF
   ├─ MANIFEST.MF (with line Multi-Release: true)
   ├─ versions
   ├─ 9
   │   ├─ A.class
   │   └─ B.class
   └─ 10
       └─ B.class
```

Pop Quiz #5

What will be the version number for the next version of Java?

1. 1.10.0
2. 10
3. X
4. 18.03

Command-Line Argument Cleanup

- Unix tools usually have short single-dash options (`ls -h`) and long double-dash options (`ls --human-readable`)
- Java tools have single-dash options (`java -classpath`)
- Except for `jar` with has archaic TAR options (`jar cvf`)
- Even command-line help uses a mixture of `-?`, `-help`, `--help`
- JEP 293 says “enough is enough”
- In the future, options will have a double-dash and a long name and may have a single-dash single-character shortcut
- The value of an option can be specified as


```
--example-option value
--example-option=value
-e value
-evalue
```

- Single character valueless options can be grouped; `-ef` is the same as `-e -f`
- `--module-path` or `-p`
- `--class-path` or `-cp`
- Of course `-classpath` still works for backward compatibility

Pop Quiz #6

Which of the following works in Java 9?

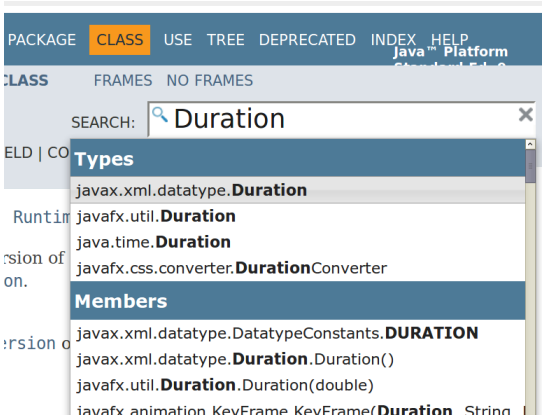
1. `java --jar MyApp.jar`
2. `jar -c -f=MyApp.jar *.class`
3. `jar -cv -f MyApp.jar *.class`
4. `jar cvf MyApp.jar *.class`

Pop Quiz #6 Solution

- `java --jar MyApp.jar`
- Nope. It's still `java -jar`.
- `jar -c -f=MyApp.jar *.class`
- Nope. Option not recognized: `-f=MyApp.jar`
- `jar -cv -f MyApp.jar *.class`
- Nope. `-f`: File or directory not found
- `jar cvf MyApp.jar *.class`
- Sure. And `jar -cvf MyApp.jar *.class` also works.

My Favorite Features

- You've seen JShell throughout this talk.
- REPL (read-eval-print loop) for Java
- My favorite Java 9 feature
- Tip #1: Shift+Tab and V key fills in variable declaration
  - Try it: `new Random()`, then Shift+Tab, then the V key (unshifted)
- Tip #2: Shift+Tab and I key fills in import declaration
  - Try it: `Duration`, then Shift+Tab, then the I key
- My second-favorite Java 9 feature: API Doc Search



A Final Hooray

- Resource files are now UTF-8
  - No more `file.preferences=Pr\u00e9f\u00e9rences`
- `javac` creates output directories, specified with the -d, -s, -h options, if they do not already exist
- The JDK no longer ships with JavaDB
- `EXIT_ON_CLOSE` is now removed from `JFrame` and inherited from `WindowConstants`
- The default “illegal access” is

```
--illegal-access=permit
```

- There is now `BigInteger.TWO`