Java Futures
Mid 2019 edition

Brian Goetz (@briangoetz)
Java Language Architect, Oracle

ORACLE

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.
It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

JAVA EVOLUTION

As Java approaches middle age…

- Java has been around for almost 25 years
  - … and been declared dead many times
- But, Java is still the world most popular programming platform
  - And we want Java to be vibrant and relevant for the next 25 years too
- We plan to (continue to) do this by
  - Staying relevant to the problems people want to solve
  - Staying relevant to the hardware people want to run on
  - Keeping the promises we've made to our users
  - Staying consistent to our principles
  - Co-evolve the JVM and the Java Language to work together

Keeping our promises

- The prime directive is: **Compatibility**
- Java is successful today because code written 25 years ago *just works*
  - Keeping our promises is how we keep our users
  - Even though this takes longer, costs more, and constrains our options
- Example: Generics
  - Codebases could gradually migrate to generics
    - A given class could be generifed now, later, or never
    - Graceful degradation at the boundary
    - No flag days
- Example: Lambdas
  - Existing libraries using single-method interfaces could automatically work with lambdas, without recompiling

First, do no harm

- Language features are forever
  - Each language feature interacts with every other (including future ones!)
  - Must pick very carefully
- If we don't know the right answer, the right answer is:
  - ***DO NOTHING (for now)***
  - Generics got this right – waited 10 years until we had the right story, rather than reaching for something like C++ templates
  - Lambdas too – we'd probably not be very happy if we'd implemented the proposals circa 2005-2006
- Whatever the feature, the goal is the same
  - *Make it easier to build and maintain reliable programs*

So, we're clearly not done

- All of this is to say: we're not done evolving Java
- Java needs to continue to evolve to
  - Adapt to new problems
  - Adapt to new hardware
  - Meet ever-rising developer expectations
- We'll never be done
  - But we're also mindful that a language can get "full"
  - Hence, we have to pick and choose features carefully

IN THE LAST YEAR (OR SO) …

Rapid Release Cadence

Feature releases every 6 months

- Moved to a 6-month, time-driven release cycle
  - Already delivered three (almost four!) releases under this plan!
- Earlier releases had been feature-driven, with releases every 2-4 years
  - Difficult to accurately plan release dates
  - Perception of slow progress
  - Small features “stuck” behind big ones
  - Significant release management overhead
- More agile, lower overhead
  - Features are late-bound to a release, when they are ready
  - OK to miss the train

New Release Cadence

9, 10, 11, 12 …

- Java 9 – released Sept 2017
  - 3 ½ years in the making
  - Over 90 JEPs
  - Somewhat disruptive release…
- Java 10 – March 2018 (new, and already old!)
  - 6 months in the making, 12 JEPs
- Java 11 – Sept 2018
  - First “LTS” release under new cadence, 17 JEPs
- Java 12 – March 2019
- Java 13 – already in “rampdown”
- Java 14 – already under development

Preview Features

Provisional status for language and platform features

- Because language features are forever, we have to get them right
  - With 2-4 year release cycles, that's a lot of time to get feedback
  - With 6mo release cycle, there's less time
  - We often get limited feedback on EA builds
- New risk-reduction mechanism: *Preview Features*
  - Complete, fully-specified features – not "experimental"
  - Opportunity to gather broader feedback, while there's still time to pull the emergency brake cord
  - Should "graduate" quickly to permanent features, or be withdrawn
  - Risky to use in production, as they may change (or even go away!)
- *Most language features will go through a round of Preview*

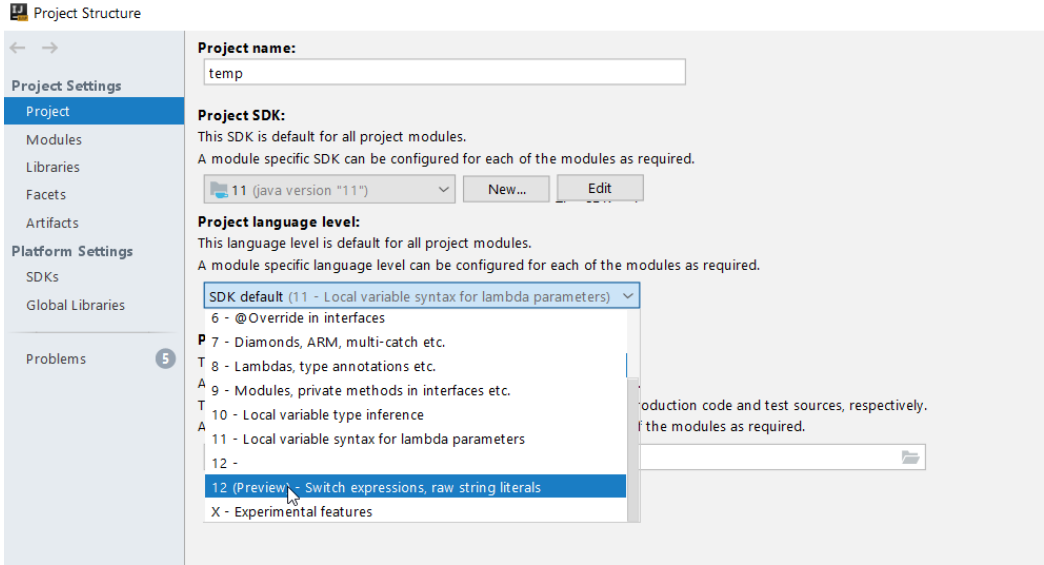
Preview Features

Provisional status for language and platform features

- Preview features described in JEP 12
  - Requires opt-in on compiler + launcher to turn them on

```
javac –enable-preview –release 12 Foo.java
java –enable-preview Foo
```

- IDE support too!



Current Initiatives

- Project Amber
  - Adapting to rising developer expectations
  - Right-sizing language ceremony
- Project Valhalla
  - Adapting to modern hardware
  - Value types, generic specialization
- Project Loom
  - Adapting to rising scale expectations
  - Fibers and continuations
- Project Panama
  - Better interop with native code and data
- … and more

Local Variable Type Inference

Added In Java 10 (three versions ago already!)

- Extend type inference to local var declaration (still just static typing)

```
URL url = new URL("http://www.oracle.com/");
URLConnection conn = url.openConnection();
Reader reader = new BufferedReader(
        new InputStreamReader(conn.getInputStream()));
```

- becomes

```
var url = new URL("http://www.oracle.com/");
var conn = url.openConnection();
var reader = new BufferedReader(
        new InputStreamReader(conn.getInputStream()));
```

- Gives users option to elide type information
  - Frequently, variable name is more useful than the type anyway

Local Variable Type Inference

- For local variables only (not fields or method return types)
- Not “just” syntactic sugar
  - Some types are not denotable
    - Intersection types, capture types, anonymous class types
  - These get special treatment
- `var c = this.getClass()`
  - `Class<cap<? extends ThisClass>>`
- `var nums = List.of(1, 2, “three”)`
  - `List<? extends Serializable & Comparable<...>>`

Local Variable Type Inference

- LVTI was one of the most commonly requested features
- …but there was also significant (and vocal) angst about it
  - "Will enable bad developers to write bad code"
  - "Just giving in to fashion"
- So far, seems to be working fine
  - Takes some time for people to get used to something new
  - Takes time to fully understand how a new feature will affect how we code
  - Takes some time for best practices to emerge
    - `http://openjdk.java.net/projects/amber/LVTIstyle.html`
    - `http://openjdk.java.net/projects/amber/LVTIFAQ.html`
  - We expect to publish similar documents for most new features

CLEARED FOR TAKE-OFF

Switch Enhancements

Preview feature in 12 + 13

- A significant fraction of `switch` statements want to be expressions
  - Assign to a common target in each arm
  - Unnecessarily indirect, error-prone
- The need to "break" on each case is irritating
  - And worse, error-prone
- Been exploring `switch` enhancements through *pattern matching*
- JEP 325 “sediments out” some of these enhancements
  - Expression form of `switch`
  - Single-consequence, fallthrough-free form of `case` for statement switches
  - Streamlining multiple case labels

Switch Enhancements

Typical use of switch to simulate an expression

```
int numLetters;
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
       numLetters = 6;
       break;
    case TUESDAY:
       numLetters = 7;
       break;
    case THURSDAY:
    case SATURDAY:
       numLetters = 9;
       break;
    case WEDNESDAY:
       numLetters = 10;
       break;
    default: throw new UnexpectedDayException(day);
}
```

Switch Enhancements

Expression switch, using simplified case labels

```
int numLetters
    = switch (day) {
        case MONDAY, FRIDAY, SUNDAY -> 6;
        case TUESDAY -> 7;
        case THURSDAY, SATURDAY -> 8;
        case WEDNESDAY -> 9;
        // no default needed
    };
```

Switch Enhancements

What's new?

- Two orthogonal enhancements to switch
  1. Can use `switch` as either expression or statement
     - Expression switches must be exhaustive
     - In expression switches, break takes a value (`break "foo"`)
  2. Streamlined case clauses (`case` label `->` consequence)
     - Single consequence (only one thing on RHS), but can be a block
     - No fallthrough allowed
     - Can specify multiple labels on one line
     - Break rarely needed (unless using blocks)
     - Works with both expression and statement forms
- (And ready for pattern matching, when it's ready)

Multi-line String Literals

Preview feature in 13

- Multi-line strings require quotes and concatenation on each line
  - Error-prone for snippets of JSON, SQL, HTML
  - Manual mangling → introduces errors, harder to read
- If we could just cut and paste in a snippet of JSON..
  - Easier to read, eliminates places for bugs to hide
- Originally was going to be a preview feature for 12
  - But was withdrawn and revised

Multi-line String Literals

```
String html = "<html>\n" +
              "  <body>\n" +
              "    <p>Hello World.</p>\n" +
              "  </body>\n" +
              "</html>\n";
```

Multi-line String Literals

```
String html = """
..............<html>
..............   <body>
..............     <p>Hello World.</p>
..............   </body>
..............</html>
              """;
```

ON THE BOARD

Pattern matching

Phase I: Type patterns in instanceof (JEP 305)

- We write test-and-extract code all the time

```
if (obj instanceof Integer) {
    int intValue = ((Integer) obj).intValue();
    // use intValue
}
```

- Type name is repeated in both instanceof test and cast
  - Irritating, error-prone
  - Obfuscates business logic
- Yes, we could do flow typing
  - But there's a better answer: *pattern matching*
  - A *pattern* fuses testing, conditional extraction, and binding to variables

Pattern matching

Type patterns in instanceof

- We can write a *type pattern* as a type name, plus a variable name:
  - `Integer intValue`
  - Looks like a variable declaration (not by accident)
- You can put a pattern on the RHS of `instanceof`
  - Now can rewrite our test-and-extract code to use type patterns

```
if (obj instanceof Integer intValue) {
    // use intValue
}
```

- Other kinds of patterns too, and other constructs that can use them
  - Type patterns in instanceof will be first
  - Nearly 100% of casts will just disappear

Pattern matching

Type patterns in instanceof

- Pattern matching works nicely with short-circuiting boolean AND (&&)
  - Almost all `equals()` methods can become a single expression
  - Simplifies control flow
  - Binding variables only in scope where they would be “definitely assigned”

```
public boolean equals(Object o) {
    return (o instanceof ThisClass t)
        && this.size == t.size
        && Objects.equals(this.name, t.name);
}
```

Pattern Matching

Phase II: Type patterns in switch

```
String formatted = "unknown";
if (constant instanceof Integer) {
    int i = (Integer) constant;
    formatted = String.format("int %d", i);
}
else if (constant instanceof Byte) {
    byte b = (Byte) constant;
    formatted = String.format("byte %d", b);
}
else if (constant instanceof Long) {
    long l = (Long) constant;
    formatted = String.format("long %d", l);
}
else if (constant instanceof Double) {
    Double d = (Double) constant;
    formatted = String.format("double %f", d);
}
// Short, Character, Float, Boolean
else if (constant instanceof String) {
    String s = (String) constant;
    formatted = String.format("String %s", s);
}
```

Pattern Matching

Type patterns in switch

```
String formatted;
switch (constant) {
    case Integer i:
        formatted = String.format("int %d", i);
        break;
    case Byte b:
        formatted = String.format(“byte %d", b);
        break;
    case Long l:
        formatted = String.format(“long %d", l);
        break;
    case Double d:
        formatted = String.format(“double %f", d);
        break;
    case String s:
        formatted = String.format("String %s", s);
         break;
    // Short, Character, Float, Boolean
    default:
        formatted = "unknown";
}
```

Pattern Matching

Type patterns in expression switch

```
String formatted =
    switch (constant) {
        case Integer i -> String.format("int %d", i);
        case Byte b    -> String.format(“byte %d", b);
        case Long l    -> String.format(“long %d", l);
        case Double d  -> String.format(“double %f", d);
        case String s  -> String.format("String %s", s);
        // Short, Character, Float, Boolean
        default -> "unknown";
    }
```

Records

“Plain old data” classes

- Many classes are just “dumb” aggregating wrappers for some data
  - Well-understood how to model data as objects
  - But, modeling overhead is high
  - Lots of tedious, error-prone boilerplate code
    - Constructors, accessors, Object methods
- Claim: this boilerplate is needed because we’re writing at the wrong level of abstraction
  - We’re writing classes, which are very general
  - We need a way of saying “this class is merely a container for this data”
  - By giving up the flexibility to decouple representation from interface contract, compiler can safely fill in the boilerplate
  - Because we’ve already communicated the semantics

Records

Many classes are just “dumb data holders”

```
class Point {
    final int x;
    final int y;
}
```

```
class Point {
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Point point = (Point) o;
        if (x != point.x) return false;
        return y == point.y;

    }

    @Override
    public int hashCode() {
        int result = x;
        result = 31 * result + y;
        return result;
    }

    @Override
    public String toString() {
        return "Point{x=" + x + ", y=" + y + '}';
    }
}
```

Records

- Records give us some pleasant boilerplate reduction
  - But are not *about* boilerplate
  - A record says, declaratively: “I am a simply a carrier for my data”
  - The boilerplate reduction derives from this semantic commitment
  - Allows compiler to infer state-driven methods – and more (stay tuned)
- Similar to the trade we make with enums
  - Give up control over instance creation
  - Gain a lot of functionality for free – because we made a semantic concession

Sealed Types

- Records give us one half of *algebraic data types* (product types)
- The other half (sum types) is useful too!
- A sum type is a discriminated union
  - Shape = Circle | Rect
- Allows compiler to reason about exhaustiveness
  - “A Shape is either a Circle or a Rect”
- A sealed type may only be extended by a limited set of types

Sealed Types

```
record Point(int x, int y) { }

sealed interface Shape { }

record Circle(Point center, int radius) implements Shape { }
record Rect(Point ll, Point ur) implements Shape { }
```

Pattern Matching, again

Phase III: Deconstruction patterns, nested patterns, and more

- Records and pattern matching go very well together
  - We can freely decompose a record into its data
- A *deconstruction pattern* is like the opposite of a constructor
  - Performs a type test, and if it passes, extracts and binds components

```
if (shape instanceof Circle(var center, var radius)) {
    // use center, radius
}
```

- Deconstruction patterns are class members, just like constructors
  - We're not magically guessing based on field names
  - Classes designed for deconstruction would provide a deconstructor
  - Records will get them for free

Pattern Matching

Deconstruction patterns

- Deconstruction patterns can work in switch too
  - And will work nicely with sealed classes

```
Shape s = ...
Float area = switch (s) {
    case Circle(var center, var r) -> PI * r * r;

    case Rect(Point p1, Point p2)
        -> (p2.x – p1.x) * (p2.y – p1.y);

    // Exhaustive switch over sealed type, no default needed!
    default -> ...
}
```

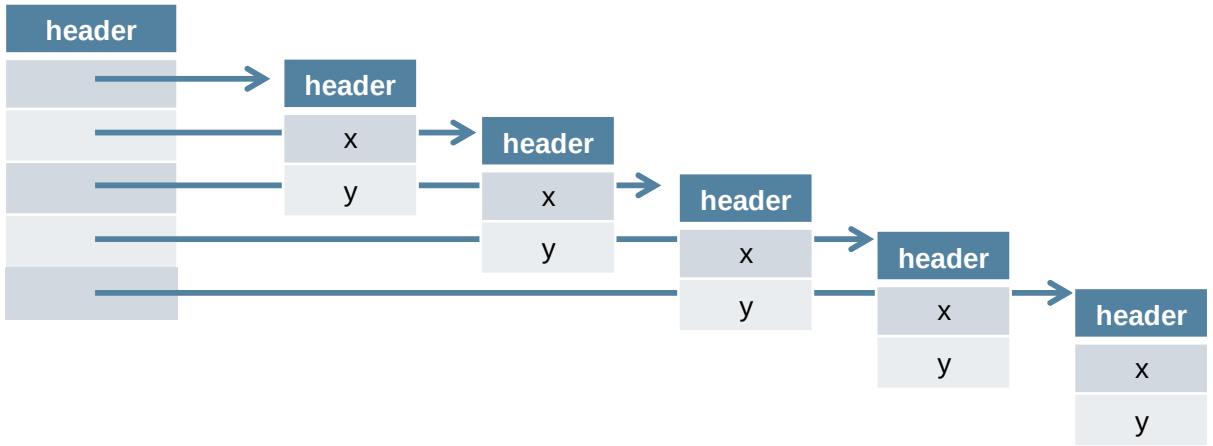
Project Valhalla

- Project Valhalla aims to *reboot the layout of data in memory*
- Why is this so important?
- Over the past 25 years, hardware has changed a lot
  - Relative cost of memory fetch vs arithmetic has increased by 200-1000x
  - Indirections (pointer fetches) are hazardous to performance
- Java data structures are so "pointery" because of *object identity*
  - Identity is needed for polymorphism, mutability, locking
  - Not all objects need that!
  - But all objects pay for it

Data Layout

The data layout we have

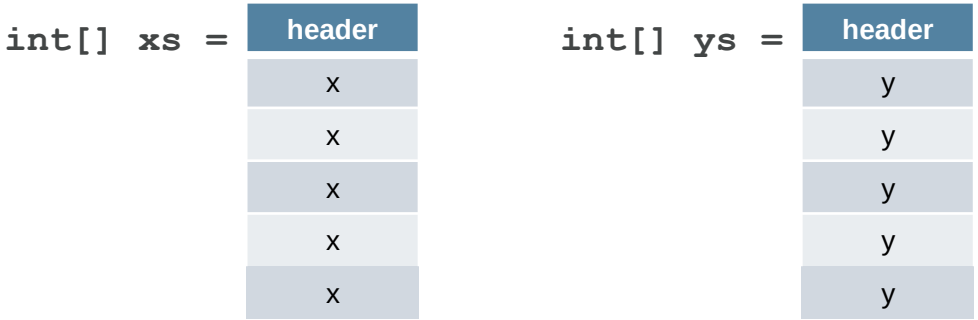
```
final class Point {
    final int x;
    final int y;
}

Point[] pts =
```



Data Layout

What we don't want people to do

- Some developers will do this …

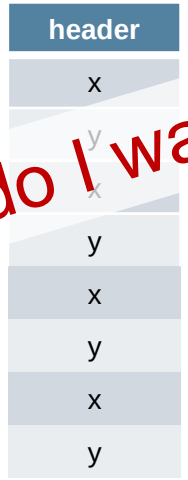


  - Less readable, more error-prone → less maintainable
- Stems from bad choice: "abstraction or performance, pick one"
- But, we can't automatically figure out that a data structure will never rely on identity
  - Need some help from the programmer

Data Layout

The data layout we want

```
inline class Point {
    int x;
    int y;
}

Point[] pts =
```



What code do I want to write to get this layout?

Value Types

Our starting point

- Value types are “pure data” aggregates
  - Just data, no identity
  - Equality comparison based on state (since there is no identity)
  - No representational polymorphism (superclasses or subclasses)
  - Not mutable
  - Not nullable*
- By giving up on identity, mutability, polymorphism, we get…
  - Values *routinely flattened into arrays, other values, objects*
  - No object header needed
  - Aggregates (with behavior) that have runtime behavior of primitives

*Some possible relaxation may be needed here for migration compatibility

Value Types

- But, unlike primitives
  - Can have methods, fields
  - Can implement interfaces
  - Can use encapsulation to hide representation
  - Can be generic
- General rubric for answering “how would it work” questions
  - “What Would Int Do”
- Could equally describe them as
  - Faster objects (with restrictions)
  - User-defined primitives (with fields, methods, encapsulation, etc)

Codes like a class, works like an int

Value Types

Who wants value types?

- Application writers
  - Can reason about locality and footprint of data-intensive code
- Library writers
  - Efficient and expressive implementations of smart pointers, numerics, cursors, value-wrappers like `Optional`
  - More efficient collections
- Compiler writers
  - Efficient substrate for language features like tuples, multiple return, built-in numeric types, wrapped native resources
- Everyone wants value types!

Project Valhalla

Current Status

- Been running more than four years
  - In that time, we've built five rounds of prototypes
  - Each aimed at investigating a particular aspect of the problem
- The current prototype ("LW1") has validated the VM underpinnings
  - Flattened layout
  - JIT optimizations
  - Enough language support
- The next prototype ("LW2") should be good enough for experimentation
  - Including erased generics over values
  - But not (yet) specialized generics over values
  - EA soon!

Project Valhalla

Sample results

- Consider matrix multiplication with complex elements
  - How do we model Complex? With a class

```
public class Complex {

    private final double re;
    private final double im;

    ...

    public Complex add(Complex that) {
        return new Complex(this.re + that.re, this.im + that.im);
    }

    public Complex mul(Complex that) {
        return new Complex(this.re * that.re - this.im * that.im,
                           this.re * that.im + this.im * that.re);
    }
}
```

Project Valhalla

Sample results

- We can multiply these in the obvious way
  - But with lots of allocation and indirection

```
public Complex[][] multiply() {
        int size = A.length;
        Complex[][] R = new Complex[size][size];
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                Complex s = new Complex(0, 0);
                for (int k = 0; k < size; k++) {
                    s = s.add(A[i][k].mul(B[k][j]));
                }
                R[i][j] = s;
            }
        }
        return R;
    }
}
```

Project Valhalla

Sample results

- JMH benchmark results (mainstream i7 system)
  - Take with appropriate skepticism

Metric	Boxed	Value	Factor
Time/op (ms)	3609	298	12.1
Allocation/op (MB)	3823	3.8	1006
Instructions	7.8G	2.5G	3.1
Instructions/cycle	1.02	2.63	2.6

Summary

Lots of stuff in the pipeline!

- Amber already delivering
  - LVTI, Expression Switch
- Lots more coming
  - Records, sealed types, pattern matching
- Valhalla starting to bear fruit
  - After years in the lab, getting solid results with real code
  - EA builds in the next year
- Panama and Loom also making tremendous strides
- Come back next year to gauge our progress!

Q & A