Project Amber Update

Brian Goetz (@briangoetz)
Java Language Architect, Oracle

Project Amber

- Most OpenJDK projects (e.g., Panama, Valhalla, Loom) aim towards a fixed set of deliverables, and the project eventually “finishes”
- Project Amber is ongoing, is an umbrella for multiple feature streams
  - Marketing slogan: “Right-sizing language ceremony”
- Most Amber features are standalone improvements that make code clearer, more concise, or less error-prone
  - Some are bigger features arcs that are delivered over time

Amber features

- (JDK 10) Local variable type inference (“var”)
- (JDK 12) Switch expressions
- (JDK 13) Text blocks (two-dimensional string literals)
- (JDK 14) Records (nominal product types)
- (JDK 15) Sealed types (sum types)
- (JDK 14, 17, 19, more in progress) Pattern matching
- (in progress) String templates
- (in progress) “Paving the on ramp”

Records

Delivered in JDK 14

- Records appeal to the desire to model data with less boilerplate

```
record Name(String firstName, String lastName) { }
```

- Shallowly immutable class with API and implementation derived from the state description
  - Fields, constructors, getters, equals, hashCode, toString, deconstruction patterns
  - User can explicitly declare members if they want a different implementation
  - They are classes, so can have supertypes, methods, etc
  - Constructors can perform validation, argument normalization
  - Can use a streamlined form for explicit default constructor

```
record Range(int lo, int hi) {
    Range {
        if (lo > hi)
            throw new IllegalArgumentException();
    }
}
```


Records

- Many people thought (or still think) they wanted structural tuples
  - But Java has a strong commitment to nominality
    - Because names matter
    - NameAndScore is more descriptive (and safer) than `(String, int)`
  - And, nominal and structural types mix poorly
- Records are “nominal product types”
  - We played a similar trick as with functional interfaces in Lambda
    - Functional interfaces are “nominal function types”, defined with ordinary interfaces

Records

- Most developers will *think of* records as being a "syntax generator"
  - Akin to code generators like Lombok, AutoValue, etc
- Records are actually a *semantic* feature
  - "The data, the whole data, and nothing but the data"
  - API cannot diverge from that implied by state description
  - Can't have extraneous state
  - Strong state contract: `new R(r.c0(), r.c1(), ...)` must be equal to `r`
    - A record forms an *embedding-projection pair* with its product space
  - Frameworks can therefore manipulate records with confidence
    - Serialization already treats records specially and more safely

Sealed Classes

Delivered in JDK 15

- Classes and interfaces that limit which classes can extend them

```
sealed interface Shape
    permits Circle, Rectangle { ... }
```

  - Permits clause can be inferred if all subclasses are co-declared
  - Subclasses can be explicitly unsealed to enable controlled extension
  - *Sealed classes are nominal sum types*
- Good for security – you can use interfaces to cleanly define and evolve APIs and be confident you won’t get malicious subtypes
- Provides language with better exhaustiveness information
  - Better type checking for exhaustive switches, can omit `default` clause

Pattern matching

- Pattern matching is a natural fit for algebraic data types
  - Delivered separately from records + sealed types, but designed to work together
- Has rolled out in phases
  - Type patterns in instanceof (JDK 14)
  - Type patterns in switch (JDK 17)
  - Record patterns and nested patterns (JDK 19)
  - More to come…
- Each of these has had to drag big updates to some other feature(s) along with it
  - Variable scoping, switch, exhaustiveness checking

Pattern matching

Type patterns (JDK 14)

- A type pattern looks like a variable declaration

```
if (x instanceof String) {
    String s = (String) x;
    // use s
}
```

- Becomes

```
if (x instanceof String s) {
    // use s
}
```

- Users' first impression is probably "casts go away"
  - Removing casts is removing places for bugs to hide
  - There's way more to it, but you have to start somewhere

Pattern matching

Patterns in switch (JDK 17)

- Because it was copied too literally from C, the switch statement in Java is both weak and complex
  - Can only switch over a limited set of types, can only compare for equality with constants, statement-only (no expressions)
  - Generalized switch to accept patterns as case labels, support all types, use exhaustiveness information from sealed types, add switch expressions

```
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);
        default -> "unknown";
    }
```


Pattern matching

Record patterns (JDK 19)

- Because we can derive the API of records from their state description, records can provide destructuring for free as well as aggregation

```
record Circle(Point center, int radius) { }

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

- And patterns can be composed by nesting

```
if (shape instanceof Circle(Point(var x, var y), var radius))
{
    // use x, y, radius
}
```


Putting it together

Did you get some Haskell in my Java?

```
data Expr =
    SumExpr Expr Expr
    | ProdExpr Expr Expr
    | NegExpr Expr
    | ConstExpr Integer

eval :: Expr -> Integer

eval SumExpr a b = (eval a) + (eval b)
eval ProdExpr a b = (eval a) * (eval b)
eval NegExpr a = - (eval a)
eval ConstExpr i = i
```

 | |

Putting it together

Did you get some Haskell in my Java?

```
sealed interface Expr {
    record SumExpr(Expr left, Expr right)  implements Expr { }
    record ProdExpr(Expr left, Expr right) implements Expr { }
    record NegExpr(Expr e)                 implements Expr { }
    record ConstExpr(int c)                implements Expr { }
}

static int eval(Expr e) {
    return switch (e) {
        case SumExpr(var a, var b) -> eval(a) + eval(b);
        case ProdExpr(var a, var b) -> eval(a) * eval(b);
        case NegExpr(var a) -> -eval(a);
        case ConstExpr(var i) -> i;
    }
}
```

Inferred permits clause

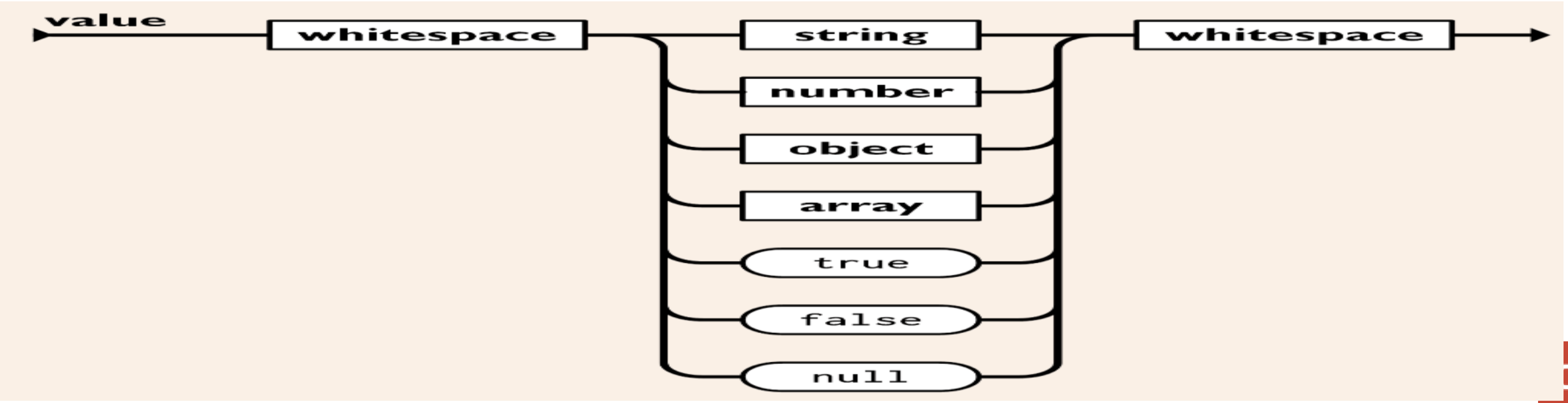
Switch expression (Java 12)

Record pattern with type inference

No default needed

Digression: JSON

- If you read the JSON spec, you'll see JSON is really just an ADT too
  - Normally we think of API design as a highly creative activity, but sometimes we should let the data do the designing
  - ADTs have a normalizing effect on API design



Digression: JSON

- If you read the JSON spec, you'll see JSON is really just an ADT
  - Normally we think of API design as a highly creative activity, but sometimes we should let the data do the designing
  - ADTs have a normalizing effect on API design

```
sealed interface JsonValue {
    record JsonString(String s)                         implements JsonValue { }
    record JsonNumber(double d)                         implements JsonValue { }
    record JsonNull()                                   implements JsonValue { }
    record JsonBoolean(boolean b)                       implements JsonValue { }
    record JsonArray(List<JsonValue> values)            implements JsonValue { }
    record JsonObject(Map<String, JsonValue> pairs)     implements JsonValue { }
}
```

Digression: JSON

- If we modeled JSON as an ADT with records and sealed types (not actually suggesting this), we could match

```
{ "name": "John", "age": 30, "city": "New York" }
```

with

```
if (j instanceof JsonObject(var pairs)
        && pairs.get("name") instanceof JsonString(String name)
        && pairs.get("age") instanceof JsonNumber(double age)
        && pairs.get("city") instanceof JsonString(String city)) {
    // use name, age, city
}
```

- Takes a messy, untyped blob of data, expresses constraints we need extracts the bits we want in the form we needed, *all in one go*
  - Without a million error-handling paths

Data Oriented Programming

- Why did we pick these features (records, sealed types, pattern matching)?
  - Sure, they solve common pain points
  - Sure, developers love them (developers REALLY love records)
- But, they also move us towards an approach that is better suited to today's application development: *data-oriented programming*

Towards Data Oriented Programming

- OOP is well suited to modeling complex entities and processes
  - Encapsulation separates implementation from interface
  - Encourages polymorphism
  - Behavior travels with state
  - Supports modular reasoning
- At its best when defining and defending *boundaries* (internal or external)
  - Maintenance, versioning, compilation, security, encapsulation boundaries…
- Modeling pure data with OOP is cumbersome
  - We tolerated this when data was just “the degenerate form of objects”

Towards Data Oriented Programming

Shifting practices in application development

- Program units are getting smaller
  - Smaller services can be maintained by a single team or developer, so don't need internal boundaries for managing complexity
- And coupled via less strongly typed schema
  - Boundaries between services defined by JSON, not Java objects
  - Much of what is exchanged is *pure data*
- Java should be good at this as well!
  - Untyped data is the new boundary
- Pattern matching is a great fit for defining the "new boundaries"
  - Where untyped data enters the service and becomes Java data
  - Concise specification of what input you expect and how to extract the parts you want, at the boundary of your program
  - Inside the boundary, it's all just (immutable) Java objects

Data Oriented Programming

- Data Oriented Programming encourages us to *model data as data*
  - Data should be *immutable*
  - Data should be *strongly typed*
  - Data should be *consistent* (Ideally, invalid states are *unrepresentable)*
  - Data should be easily convertible to and from the wire / file system
  - Data should be separate from nontrivial behavior on that data
  - These conspire to reduce the need for internal boundaries
- But still using natural idioms for the language
  - A service may take its input as JSON, but we want to quickly convert to data types that make more sense for Java
  - No “stringly typed” programming
- As a bonus, generally renders programs more testable
  - Specifically, more amenable to *generative testing* (testing with randomly generated domain-conformant test data)

Next steps in pattern matching

Deconstruction patterns for all classes

- Records got deconstruction for free because their API and implementation are automatically derived from the state description
  - How will regular classes express deconstruction?
  - With *deconstruction patterns,* which are the dual of constructors
- Deconstruction patterns will be declarable as class members
  - Can “return” multiple values, and some patterns will be conditional
  - Language’s flow analysis tracks pattern success or failure
- In general, for every object creation idiom, there should be a corresponding pattern dual, with similar syntax
  - Static patterns are the dual of static factories
  - Because this is how we make destructuring as composable as creation

Deconstruction patterns

Coming soon!

- Classes can declare deconstruction patterns (which are unconditional)
  - Look like a constructor in reverse (precise syntax TBD)

```
public class Point {
    int x, y;

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

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

Matched Pair

String templates

Coming soon!

- Most common feature request: “string interpolation, please”
- String interpolation is convenient but dangerous
  – Breeding ground for SQL/HTML injection attacks
- The alternatives we give users today aren’t great, though
  – String concatenation – just as unsafe, and less readable
  – String::format – harder to read, more error-prone
  – StringBuilder – yuck
- Most languages treat this as another form of string literal
  – Convenient shortcut, but limited in power
  – May lead to combinatorial explosion of string literal forms

String templates

- We solved this with “another level of indirection”
- A string template expression is a combination of literal text and embedded expressions
  - Plus a *template processor*

```
String greeting = STR."Hello \{name}"
```

- Template processor takes a template and produces *something*
  - `STR` is a predefined processor that does interpolation
  - But, processors can also perform arbitrary validation and transformation
    - Don’t even have to result in a String
  - Templates work with both single-line string literals and text blocks

String templates

- Writing more sophisticated template processors is easy

```
String line = FMT."Name: %-12s\{name}; size: %7.2f\{size}"
```

  - Formats using traditional String::format specifiers, preceding the embedded expressions

```
TemplateProcessor<ResultSet> db = new QueryProcessor(connection);
ResultSet rs
    = db."SELECT * FROM Person p WHERE p.last_name = \{name}";
```

  - DB processor validates SQL string for quote hygiene, escapes embedded expressions, creates prepared statement, and executes query

- Other applications include message localization, creating JSON objects without transiting through intermediate String format, etc
- Subversion: we snuck in validation and transformation when users thought they were just getting interpolation

Paving the on-ramp

Making Java easier for beginners

- Our first program is often “Hello World”

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

- This is full of boilerplate that makes people think “Java is hard”
- Worse, it is full of object-oriented *concepts* students are not ready for
  - Requires a lot of “you’ll understand that later”
  - Forces distortions in how we teach Java
- Value of these things comes much later, in organizing larger programs

Paving the on-ramp

Making Java easier for beginners

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(“Hello!”);
    }
}
```

Access control for encapsulation

Classes for modeling and organization

Static vs instance behavior

Command line interaction, arrays

Access control, again

Magic method name

Static fields

Paving the on-ramp

Making Java easier for beginners

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(“Hello!”);
    }
}
```

Paving the on-ramp

Making Java easier for beginners

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(“Hello!”);
    }
}
```

Paving the on-ramp

Making Java easier for beginners

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(“Hello!”);
    }
}
```

Paving the on-ramp

Making Java easier for beginners

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(“Hello!”);
    }
}
```

Paving the on-ramp

Making Java easier for beginners

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(“Hello!”);
    }
}
```

Paving the on-ramp

Making Java easier for beginners

~~`public class HelloWorld {`~~

    ~~`public`~~ ~~`static`~~ `void main(`~~`String[] args`~~`) {`

        ~~`System.out.`~~`println(“Hello!”);`

    ~~`}`~~

~~`}`~~

Paving the on-ramp

Making Java easier for beginners

```
void main() {
    println(“Hello!”);
}
```

Paving the on-ramp

Making Java easier for beginners

- This may appear to be merely syntax and boilerplate, but isn't really
  - Simple programs should be simple
  - Start with simple methods, build up to classes at your own rate
  - Also useful for writing scripts in Java
- More importantly, this removes the last linchpin supporting a suboptimal education approach – "early objects"
  - OO makes more sense after you've written some bad imperative programs
- *Educators can now teach Java the way they teach Python, without guilt*
  - OO concepts can be added in later, when they directly add value
- See "Paving the On Ramp"
  - https://openjdk.org/projects/amber/design-notes/on-ramp

Summary

- Externally, Amber means steady improvement in the language, and the "small", "productivity-oriented" features developers crave
  - New language features in most JDK releases
- Internally, Amber represents a new way of evolving the language
  - Break big features down into smaller pieces, but connect the pieces so they are part of a larger story arc
  - Some deceptively big things can emerge from seemingly "small" features!
    - E.g., safer serialization and withers emerging from deconstruction
    - There's a reason we have picked these features in this order
- These features are not "mere syntax"!
  - Making data-oriented programming more natural
  - Enabling new ways for educators to teach Java

