Pattern Matching in Java

Srikanth Sankaran,
Consulting Member of Technical Staff
Java Product Group, Oracle

August 2017

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.

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Pattern Matching - Motivation

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

- Test to see if an expression has some characteristic
- Convert
- Destructure and extract interesting state bits

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Pattern Matching - Motivation

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

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Pattern Matching – Type Test Patterns

- Pattern
  - A combination of a predicate applied to a target &
  - A set of binding variables extracted from the target if predicate applies to it.

```
if (x matches Integer i) {
    // can use i here
}
```


Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Pattern Matching – Type Test Patterns

```
String formatted = "unknown";
if (obj matches Integer i) {
    formatted = String.format("int %d", i);
} else if (obj matches Byte b) {
    formatted = String.format("byte %d", b);
} else if (obj matches Long l) {
    formatted = String.format("long %d", l);
} else if (obj matches Double d) {
    formatted = String.format(“double %f", d);
} else if (obj matches String s) {
    formatted = String.format("String %s", s);
} // ...
```

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Improved 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;
    // String, Short, Character, Float, Boolean
    default: formatted = "unknown";
}
```

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

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";
    };
```


Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Generalizing Switch: Constant Patterns

```
String s =
    exprswitch (num) {
        case 0 -> "zero";
        case 1 -> "one";
        case int i -> "some other Integer";
        default -> "not an Integer";
    };
```


Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Destructuring Patterns

```
if (node matches AddNode(Node x, Node y)) {  ...  }


int eval(Node n) {
    return exprswitch(n) {
        case IntNode(int i) -> i;
        case NegNode(Node n) -> -eval(n);
        case AddNode(Node left, Node right) -> eval(left) + eval(right);
        case MulNode(Node left, Node right) -> eval(left) * eval(right);
    };
}
```


Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Nested Patterns

```
if (node matches AddNode(Node x, Node y)) {  ...  }
```

- “Node x” may look like binding variable declaration
- It is actually a nested type test pattern !

The pattern AddNode(p1, p2), where p1 and p2 are patterns, matches a target if:

- the target is an AddNode;
- the left component of that AddNode matches p1;
- the right component of that AddNode matches p2.

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Evaluating Expressions: Var patterns

```
int eval(Node n) {
    return switch(n) {
        case IntNode(var i) -> i;
        case NegNode(var n) -> -eval(n);
        case AddNode(var left, var right) -> eval(left) + eval(right);
        case MulNode(var left, var right) -> eval(left) * eval(right);
        case ParenNode(var node) -> eval(node);
    };
}
```


Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Nesting Constant patterns

```
String formatted = exprswitch (anObject) {
    case Point(0, 0) -> "at origin";
    case Point(0, var y) -> "on x axis";
    case Point(var x, 0) -> "on y axis";
    case Point(var x, var y) -> String.format("[%d,%d]", x, y);
    default -> "not a point";
};
```


Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Example: Simplifying Expressions

- Suppose we want to simplify algebraic expressions
  - 0 + e == e
  - 1 * e == e
  - 0 * e == 0
  - etc
- We could do this with Visitors … yuck
  - Much easier with pattern matching
  - Less ceremony, easier composition

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Nesting nontrivial patterns

Nested Pattern!

```
Node simplify(Node n) {
    return switch(n) {
        case IntNode -> n;

        case NegNode(NegNode(var n)) -> simplify(n);
        case NegNode(var n) -> simplify(new NegNode(simplify(n)));

        case AddNode(IntNode(0), var right) -> simplify(right);
        case AddNode(var left, IntNode(0)) -> simplify(left);
        case AddNode(var left, var right)
            -> simplify(new AddNode(simplify(left), simplify(right)));

        case MulNode(IntNode(1), var right) -> simplify(right);
        case MulNode(var left, IntNode(1)) -> simplify(left);
        case MulNode(IntNode(0), var right) -> new IntNode(0);
        case MulNode(var left, IntNode(0)) -> new IntNode(0);
        case MulNode(var left, var right)
            -> simplify(new MulNode(simplify(left), simplify(right)));
    };
}
```


Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

The _ pattern

```
case MulNode(IntNode(0), _) -> new IntNode(0);
```


Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Patterns - Summary

```
Types of Patterns:
    - Type-test patterns, which bind the cast target to a binding variable;
    - Destructuring patterns, which destructure the target and recursively match
    - Constant patterns, which match on equality;
    - Var patterns, which match anything and bind their target;
    - The _ pattern, which matches anything.

Contexts:

    - Matches predicate
    - Enhanced switch statement
    - Expression switch
```

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

ORACLE®