Paths to support additional numeric types on the Java platform

Joseph D. Darcy (OpenJDK darcy, jddarcy, @jddarcy)
Java Platform Group, Oracle

JVM Language Summit
August 5, 2025

Starting Context

- Exploring adding new numeric types to the Java platform in anticipation of
  - Value classes
  - Type classes
  - …
- Give feedback on the design of those language features through trial usage.
- This talk represents what we've worked out so far.
  - Subject to change in response to future work.
  - More explorations expected to follow
- Not exhaustive, intend to convey a representative sampling of the considerations at play.

Safe Harbor Statement

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.

Outline

- Brief visit with abstract algebra
- Jobs for numerical types
- Platform implications of adding a primitive type
- *Growing a Language* and type classes
- `Float16` Retrospective and Prospective
- Current thoughts on modeling numerics
- Design considerations in other potential new numeric types: complex and imaginary numbers

Motivation: uses of additional numeric types

Better support in multiple areas

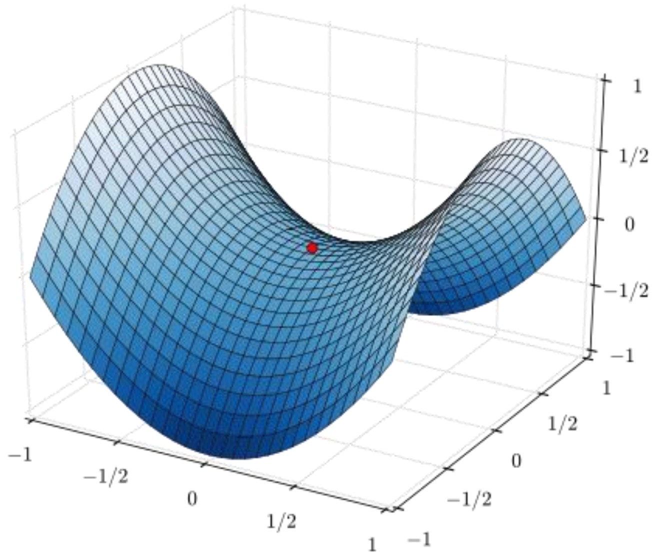
- Traditional scientific and engineering computation
- Education and “paving the on-ramp”
- Machine Learning/AI

Platform support for numeric types

- For a given, type *platform* support means using some combination of
  - Current and future features of the Java programming language
  - Current and future features of the Java virtual machine
  - Current and future features of the Java core libraries
- New numeric types may be added to the platform directly, but the intention is also to allow 3rd parties to develop and use new numeric types.
- In the components above, new features may be aimed at library authors rather than end users.
- Essence of the design exercise at the platform level is determining not only *what* to support, but *how* to support it using an allocation of responsibilities among these components.

Where does Java numerics sit?

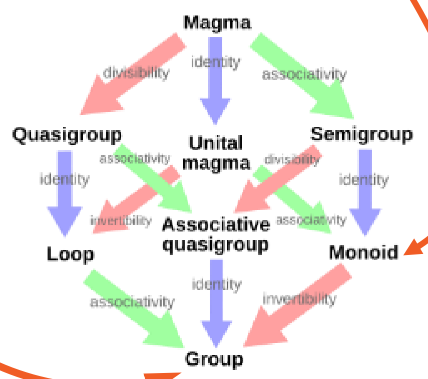
- Considerations from from the Java programming language, current and anticipated features.
- Considerations from mathematics
- Considerations from *computation*, separate from mathematics
- Explore the region around the saddle point of all these considerations



Brief visit with abstract algebra

Remember: ***"a monad is a monoid in the category of endofunctors, what's the problem?"***
—Phil Wadler (attrib.)

Fundamental abstract algebraic structures



- Monoid: ≈addition with an identity element
- Group: ≈addition and inverse elements (subtraction)
- Ring: ≈addition and multiplication

  (Rings that aren't fields can have a division operator defined over them, but don't have multiplicative inverses the way a field does.)

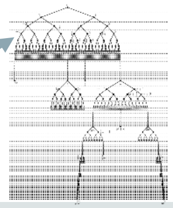
- Field: ≈addition, subtraction, multiplication, and division

  From Wikipedia: "a field has two commutative operations, called addition and multiplication; it is a group under addition with 0 as the additive identity; the nonzero elements form a group under multiplication with 1 as the multiplicative identity; and multiplication distributes over addition.
  Even more succinctly: a field is a commutative ring where 0 ≠ 1 and all nonzero elements are invertible under multiplication. "

More on fields

Various subcategories: ordered vs unordered, etc.

- Examples:
  - rational numbers ($\mathbb{Q}$), real numbers($\mathbb{R}$), complex numbers ($\mathbb{C}$)
  - algebraic numbers: superset of rational numbers, doesn’t include all real numbers (“Why don’t you just use rational numbers?”
    Rational numbers aren’t sufficient in general since they don’t contain their limits; in particular, numbers like $\sqrt{2}$ and π aren’t rational numbers)
- FYI, sometimes real numbers aren’t adequate for the desired tasks either:
  - hyperreal numbers
  - superreal numbers
  - surreal numbers



Different ways of adding infinite, trans-infinite, and infinitesimal values.

$\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, and other fields satisfy the field axioms:

Field axioms are the basis for many compiler transformations

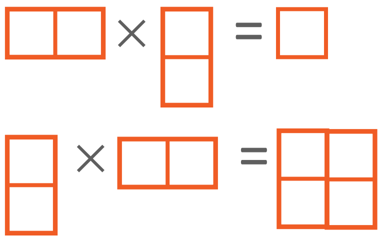
- Closed under addition
- Associative addition
  $(a + b) + c = a + (b + c)$
- Identity element for addition
  $a + 0 = 0 + a = a$
- Closed under multiplication
- Associative multiplication
  $(a * b) * c = a * (b * c)$
- Identity element for multiplication
  $a * 1 = 1 * a = a$
- Zero annihilator: $a * 0 = 0 * a = 0$
- Commutative addition
  $a + b = b + a$
- Commutative multiplication
  $a * b = b * a$
- Additive inverse
  $\forall a \, \exists b$, so that $a + b = b + a = 0$
- Multiplicative inverse
  $\forall a \mathrel{!=} 0 \, \exists b$, so that $a * b = b * a = 1$
- Distributivity:
  $a*(b + c) = a*b + a*c$, etc.

Aside: finite fields

- Smallest possible field GF(2) – Galois Field with two elements – just 0 and 1
  - addition is XOR
  - multiplication is AND
  - Proving the field axioms hold is left “as an exercise for the reader.”
- Other finite fields used in cryptography for elliptic curves and other algorithms.

Still more algebraic structure: vectors and matrices

- Vector space: built using elements of a Field; from Wikipedia *“a vector space is an abelian [that is, commutative. –ed.] group under addition, […] and defines a ring homomorphism from the field F into the endomorphism ring of this group.”*
- Matrices are related to vector spaces; matrices define operations of the form:
  - Matrix *op* Matrix and *scalar* * Matrix
  - Note: *two* kinds of multiplication defined: Matrix * Matrix, *scalar* * Matrix and
- Note: in general multiplication of matrices is associative, but *not* commutative; $A \times B$ and $B \times A$ can have different dimensions, etc.



Abstract algebra implications for platform design

- Extensive, rich set of relations between various algebraic structures
  - Beside top-level structures, additional properties of being ordered vs. unordered
  - Terminology not always consistent across textbooks
  - Multiple ways to extend a given structure, including multiple ways to extend the real numbers with infinity
- *Not* necessarily required (or desirable) for this full set of relations to be mirrored in a platform's numeric facilities
  - Numeric facilities should be *informed* by underlying mathematical structures.
  - Numeric facilities also informed by other design factors of the platform.

How do these algebraic structures relate to `int` and `long`?

- Two’s complement integer arithmetic forms a ring.
- Quick check: additive inverse *does* hold:

```
> Integer.MIN_VALUE + Integer.MIN_VALUE
$1 ==> 0
```

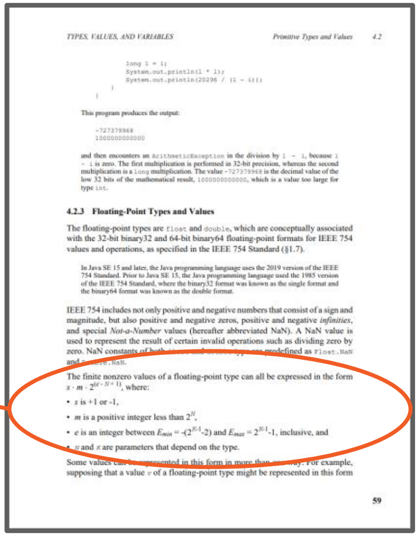
✓

How do the algebraic structures relate to `float` and `double`?

- The Java language and JVM use IEEE 754 floating-point.
- IEEE 754 arithmetic *specifically* approximates the extended reals:

Table 3.1—Relationships between different specification levels for a particular format

Level 1	$\{-\infty \ldots\ 0\ \ldots +\infty\}$	Extended real numbers.
many-to-one ↓	*rounding*	↑ projection (except for NaN)
Level 2	$\{-\infty \ldots -0\} \cup \{+0 \ldots +\infty\} \cup$ **NaN**	Floating-point data—an algebraically closed system.
one-to-many ↓	*representation specification*	↑ many-to-one
Level 3	(***sign***, ***exponent***, ***significand***) $\cup \{-\infty, +\infty\} \cup$ **qNaN** $\cup$ **sNaN**	Representations of floating-point data.
one-to-many ↓	*encoding for representations of floating-point data*	↑ many-to-one
Level 4	**0111000…**	Bit strings.



TYPES, VALUES, AND VARIABLES *Primitive Types and Values* 4.2

```
            long l = i;
            System.out.println(l * l);
            System.out.println(20296 / (l - i));
        }
    }
```

This program produces the output:

```
-727379968
1000000000000
```

and then encounters an `ArithmeticException` in the division by `l - i`, because `l - i` is zero. The first multiplication is performed in 32-bit precision, whereas the second multiplication is a `long` multiplication. The value `-727379968` is the decimal value of the low 32 bits of the mathematical result, `1000000000000`, which is a value too large for type `int`.

4.2.3 Floating-Point Types and Values

The floating-point types are `float` and `double`, which are conceptually associated with the 32-bit binary32 and 64-bit binary64 floating-point formats for IEEE 754 values and operations, as specified in the IEEE 754 Standard (§1.7).

In Java SE 15 and later, the Java programming language uses the 2019 version of the IEEE 754 Standard. Prior to Java SE 15, the Java programming language used the 1985 version of the IEEE 754 Standard, where the binary32 format was known as the single format and the binary64 format was known as the double format.

IEEE 754 includes not only positive and negative numbers that consist of a sign and magnitude, but also positive and negative zeros, positive and negative *infinities*, and special *Not-a-Number* values (hereafter abbreviated NaN). A NaN value is used to represent the result of certain invalid operations such as dividing zero by zero. NaN constants of both [illegible] are predefined as `Float.NaN` and [illegible]`.NaN`.

The finite nonzero values of a floating-point type can all be expressed in the form $s \cdot m \cdot 2^{(e - N + 1)}$, where:

- s is +1 or -1,
- m is a positive integer less than 2^N,
- e is an integer between $E_{min} = -(2^{K-1}-2)$ and $E_{max} = 2^{K-1}-1$, inclusive, and
- N and K are parameters that depend on the type.

Some values [illegible] this form in more than [illegible]. For example, supposing that a value v of a floating-point type might be represented in this form

59

JLS §4.2.3

- However, the extended reals are *none* of a group, ring, or field.
  - But, various useful algebraic properties do still hold
  - Infinities interfere with satisfying various axioms

Floating-point Equality, Equivalence, and Comparison

As discussed in `java.lang.Double`, need an equiv. relation to discuss field axioms

“Comparing numerical equality to various useful equivalence relations that can be defined over floating-point values:

***numerical equality* (== operator):** (*Not* an equivalence relation)
Two floating-point values represent the same extended real number. The extended real numbers are the real numbers augmented with positive infinity and negative infinity. *[Signed zero and NaN complications…]*

***bit-wise equivalence*:**
The bits of the two floating-point values are the same. […]

***representation equivalence*:** ✓
The two floating-point values represent the same IEEE 754 *datum*. In particular, for finite values, the sign, exponent, and significand components of the floating-point values are the same. Under this relation:

- `+0.0` and `-0.0` are distinguished from each other.
- every bit pattern encoding a NaN is considered equivalent to each other
- positive infinity is equivalent to positive infinity; negative infinity is equivalent to negative infinity. […]

Note that representation equivalence is often an appropriate notion of equivalence to test the behavior of math libraries. *[emphasis added]”*

Field axioms on floating-point arithmetic: Perils of round-off, non-finite values, and signed zero

- Closed under addition
- *Associative addition*
  $(a + b) + c = a + (b + c)$
- Identity element for addition†
  $a + 0 = 0 + a = a$
- Closed under multiplication
- *Associative multiplication*
  $(a * b) * c = a * (b * c)$
- Identity element for multiplication
  $a * 1 = 1 * a = a$

† *Using -0.0 rather than +0.0 and round to nearest.*

Hold if considering representation equivalence rather than IEEE 754 defined == relation.

- *Zero annihilator:* $a * 0 = 0 * a = 0$
- Commutative addition
  $a + b = b + a$
- Commutative multiplication
  $a * b = b * a$
- *Additive inverse*
  $\forall a \, \exists b$, so that $a + b = b + a = 0$
- *Multiplicative inverse*
  $\forall a \neq 0 \, \exists b$, so that $a * b = b * a = 1$
- *Distributivity*:
  $a*(b + c) = a*b + a*c$, etc.

Despite this situation, still desirable to use numerical operators on floating-point types.

Intermediate observation

Might have hoped for bright-line guidance about when operator usage was reasonable and tasteful based on abstract algebraic structures,
but
existing primitive types with operator support in the language have limited connections to abstract algebraic structures.

IMO, being a group/ring/field could be a considered a *sufficient* condition for a type to use operators, but *not* a necessary condition.

Therefore, we will need to develop criteria other than solely algebraic ones.

Intermediate leading question

Can there be an opt-in way to indicate a type *does* satisfy a particular subset of the field axioms?

End of abstract algebra visit

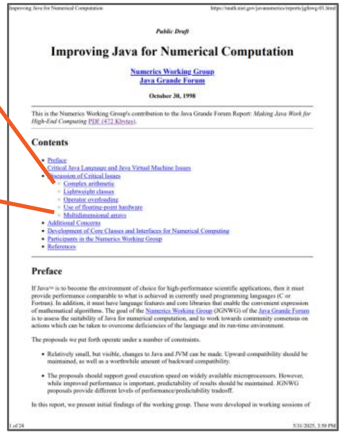
*Java is a blue collar language.
It’s not PhD thesis material but
a language for a job.*

James Gosling
The Feel of Java, Computer, June 1997.

Particular numerical types of interest for different jobs

- Complex and imaginary
- Vectors/matrices
- Quad (128-bit binary)
- Decimal types
  - IEEE 754 {32, 64, 128} bit
  - SQL-style, monetary
- Machine learning
  - Float16
  - bfloat16
  - Future 8-bit floating-point types?

Nvidia TensorFloat-32 is a 19-bit format with the significand *size of* `float16` *and the* exponent *size of* `bfloat16`. 😐



Improving Java for Numerical Computation https://math.nist.gov/javanumerics/reports/jgfnwg-01.html

Public Draft

Improving Java for Numerical Computation

Numerics Working Group
Java Grande Forum

October 30, 1998

This is the Numerics Working Group's contribution to the Java Grande Forum Report: *Making Java Work for High-End Computing* PDF (472 Kbytes).

Contents

- Preface
- Critical Java Language and Java Virtual Machine Issues
- Discussion of Critical Issues
  - Complex arithmetic
  - Lightweight classes
  - Operator overloading
  - Use of floating-point hardware
  - Multidimensional arrays
- Additional Concerns
- Development of Core Classes and Interfaces for Numerical Computing
- Participants in the Numerics Working Group
- References

Preface

If *Java*™ is to become the environment of choice for high-performance scientific applications, then it must provide performance comparable to what is achieved in currently used programming languages (C or Fortran). In addition, it must have language features and core libraries that enable the convenient expression of mathematical algorithms. The goal of the Numerics Working Group (JGNWG) of the Java Grande Forum is to assess the suitability of Java for numerical computation, and to work towards community consensus on actions which can be taken to overcome deficiencies of the language and its run-time environment.

The proposals we put forth operate under a number of constraints.

- Relatively small, but visible, changes to Java and JVM can be made. Upward compatibility should be maintained, as well as a worthwhile amount of backward compatibility.
- The proposals should support good execution speed on widely available microprocessors. However, while improved performance is important, predictability of results should be maintained. JGNWG proposals provide different levels of performance/predictability tradeoff.

In this report, we present initial findings of the working group. These were developed in working sessions of

1 of 24 5/31/2025, 3:50 PM

- Unsigned integers (including byte)
- Rational numbers/fractions
- Other entries from LISP-style numerical towers
- Niche applications
  - Quaternions ($i^2 = j^2 = k^2 = ijk = -1$)
  - Unums/posits
  - Interval arithmetics
  - Doubled-`double`
  - 80-bit binary floating-point (ABI usage)
  - …

Utility of operators on numerical types

- Many of these types would naturally benefit from at least of subset of the operators, such as the arithmetic operators {`+`, `-`, `*`, `/`}
- Many of these types satisfy the properties of an algebraic structure no better than IEEE 754 binary floating-point types like `float` and `double`.
- ∴ do ***not*** mandate strict satisfaction of the properties of a ring, field, vector space, etc. as a requirement for a type to be eligible to use operators.

Numeric types in the platform (`java.*`) as of JDK 25

- Most primitive types plus `java.math.{BigInteger`, `BigDecimal}`
- What does being a primitive type get you?
  - Language support
    - Operators
    - Literals
    - Compile-time constant-folding (JLS §15.29 *Constant Expressions*)
  - Works on every JVM and in all nooks and crannies of the platform
  - Potential for vectorization (see *Auto-Vectorization in HotSpot* later today)
  - Dense arrays – even important post-Project Lilliput's JEP 450 & JEP 519.
- Preferable for future numeric types to *act like* today's primitive types.

Speedrun through a thought experiment of what adding a new primitive type, including JVM support, would involve.

**Consider a new numeric type akin to `float` or `double`,
for concreteness keep in mind `float16` and `decimal128`.**

Abstractly, how to describe numeric support for a type?

- Create numeric values
  - Conversion from text/strings
  - Conversion from numeric values of other types
  - Conversion from binary data
  - …
- Operate on numeric values
  - Operations appropriate for that type: e.g. {+, –, *, /, ...}
  - Library support: sqrt, sin, cos, tan, …
- Output numeric values
  - Textual output, binary output
  - …

Where is that support *specified*?

The Java® Language Specification
Java SE 24 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley
Daniel Smith
Gavin Bierman

2025-02-07

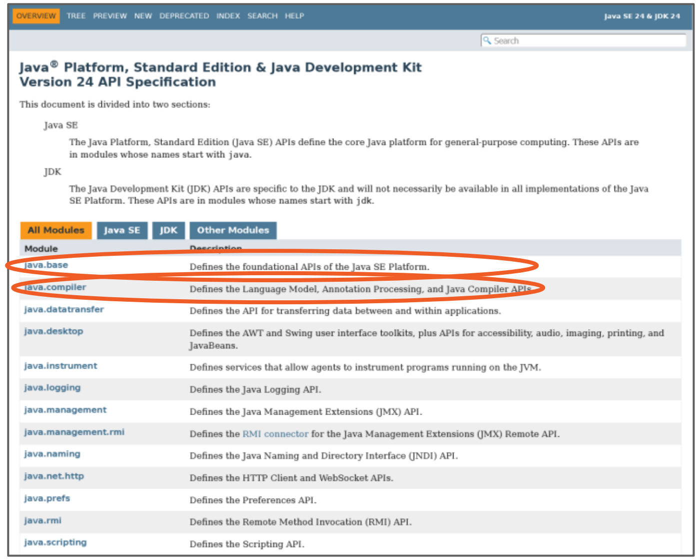
Possible future changes: pattern matching, etc.

The Java® Virtual Machine Specification
Java SE 24 Edition

Tim Lindholm
Frank Yellin
Gilad Bracha
Alex Buckley
Daniel Smith

2025-02-07

- Class libraries


OVERVIEW TREE PREVIEW NEW DEPRECATED INDEX SEARCH HELP Java SE 24 & JDK 24

Search

Java® Platform, Standard Edition & Java Development Kit Version 24 API Specification

This document is divided into two sections:

Java SE

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with java.

JDK

The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with jdk.

All Modules | Java SE | JDK | Other Modules

Module	Description
java.base	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
java.datatransfer	Defines the API for transferring data between and within applications.
java.desktop	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.
java.instrument	Defines services that allow agents to instrument programs running on the JVM.
java.logging	Defines the Java Logging API.
java.management	Defines the Java Management Extensions (JMX) API.
java.management.rmi	Defines the RMI connector for the Java Management Extensions (JMX) Remote API.
java.naming	Defines the Java Naming and Directory Interface (JNDI) API.
java.net.http	Defines the HTTP Client and WebSocket APIs.
java.prefs	Defines the Preferences API.
java.rmi	Defines the Remote Method Invocation (RMI) API.
java.scripting	Defines the Scripting API.

- JNI / FFM API
- Serialization/marshalling
- …

JLS sections

Table of Contents

1 Introduction 1

1.1 Organization of the Specification 2
1.2 Example Programs 6
1.3 Notation 6
1.4 Relationship to Predefined Classes and Interfaces 7
1.5 Preview Features 7
1.6 Feedback 10
1.7 References 10

2 Grammars 13

2.1 Context-Free Grammars 13
2.2 The Lexical Grammar 13
2.3 The Syntactic Grammar 14
2.4 Grammar Notation 14

3 Lexical Structure 19

3.1 Unicode 19
3.2 Lexical Translations 20
3.3 Unicode Escapes 21
3.4 Line Terminators 24
3.5 Input Elements and Tokens 24
3.6 White Space 26
3.7 Comments 26
3.8 Identifiers 28
3.9 Keywords 30
3.10 Literals 32
3.10.1 Integer Literals 33
3.10.2 Floating-Point Literals 40
3.10.3 Boolean Literals 43
3.10.4 Character Literals 44
3.10.5 String Literals 45
3.10.6 Text Blocks 47
3.10.7 Escape Sequences 53
3.10.8 The Null Literal 54
3.11 Separators 54
3.12 Operators 54

4 Types, Values, and Variables 55

4.1 The Kinds of Types and Values 55

4.2 Primitive Types and Values 56
4.2.1 Integral Types and Values 57
4.2.2 Integer Operations 57
4.2.3 Floating-Point Types and Values 59
4.2.4 Floating-Point Operations 61
4.2.5 The `boolean` Type and boolean Values 63
4.3 Reference Types and Values 64
4.3.1 Objects 66
4.3.2 The Class `Object` 68
4.3.3 The Class `String` 69
4.3.4 When Reference Types Are the Same 69
4.4 Type Variables 70
4.5 Parameterized Types 72
4.5.1 Type Arguments of Parameterized Types 73
4.5.2 Members and Constructors of Parameterized Types 76
4.6 Type Erasure 77
4.7 Reifiable Types 77
4.8 Raw Types 79
4.9 Intersection Types 83
4.10 Subtyping 84
4.10.1 Subtyping among Primitive Types 84
4.10.2 Subtyping among Class and Interface Types 84
4.10.3 Subtyping among Array Types 85
4.10.4 Least Upper Bound 86
4.10.5 Type Projections 89
4.11 Where Types Are Used 91
4.12 Variables 96
4.12.1 Variables of Primitive Type 96
4.12.2 Variables of Reference Type 97
4.12.3 Kinds of Variables 99
4.12.4 `final` Variables 101
4.12.5 Initial Values of Variables 103
4.12.6 Types, Classes, and Interfaces 104

5 Conversions and Contexts 109

5.1 Kinds of Conversion 112
5.1.1 Identity Conversion 112
5.1.2 Widening Primitive Conversion 113
5.1.3 Narrowing Primitive Conversion 114
5.1.4 Widening and Narrowing Primitive Conversion 117
5.1.5 Widening Reference Conversion 117
5.1.6 Narrowing Reference Conversion 118
5.1.6.1 Allowed Narrowing Reference Conversion 118
5.1.6.2 Checked and Unchecked Narrowing Reference Conversions 120
5.1.6.3 Narrowing Reference Conversions at Run Time 121
5.1.7 Boxing Conversion 124

JLS sections, cont.

The Java® Language Specification

5.1.8 Unboxing Conversion 125
5.1.9 Unchecked Conversion 127
5.1.10 Capture Conversion 127
5.1.11 String Conversion 129
5.1.12 Forbidden Conversions 130
5.2 Assignment Contexts 130
5.3 Invocation Contexts 135
5.4 String Contexts 137
5.5 Casting Contexts 137
5.6 Numeric Contexts 143
5.7 Testing Contexts 147

6 Names 149

6.1 Declarations 150
6.2 Names and Identifiers 158
6.3 Scope of a Declaration 160
6.3.1 Scope for Pattern Variables in Expressions 164
6.3.1.1 Conditional-And Operator `&&` 164
6.3.1.2 Conditional-Or Operator `||` 165
6.3.1.3 Logical Complement Operator `!` 166
6.3.1.4 Conditional Operator `? :` 166
6.3.1.5 Pattern Match Operator `instanceof` 167
6.3.1.6 `switch` Expressions 167
6.3.1.7 Parenthesized Expressions 167
6.3.2 Scope for Pattern Variables in Statements 167
6.3.2.1 Blocks 168
6.3.2.2 `if` Statements 168
6.3.2.3 `while` Statements 170
6.3.2.4 `do` Statements 170
6.3.2.5 `for` Statements 170
6.3.2.6 `switch` Statements 170
6.3.2.7 Labeled Statements 171
6.3.3 Scope for Pattern Variables in `case` Labels 171
6.4 Shadowing and Obscuring 171
6.4.1 Shadowing 175
6.4.2 Obscuring 178
6.5 Determining the Meaning of a Name 179
6.5.1 Syntactic Classification of a Name According to Context 180
6.5.2 Reclassification of Contextually Ambiguous Names 184
6.5.3 Meaning of Module Names and Package Names 186
6.5.3.1 Simple Package Names 186
6.5.3.2 Qualified Package Names 186
6.5.4 Meaning of *PackageOrTypeNames* 186
6.5.4.1 Simple *PackageOrTypeNames* 186
6.5.4.2 Qualified *PackageOrTypeNames* 187
6.5.5 Meaning of Type Names 187
6.5.5.1 Simple Type Names 187

v

CONVERSIONS AND CONTEXTS *Casting Contexts* 5.5

Table 5.5-A. Casting to primitive types

To → / **From ↓**	`byte`	`short`	`char`	`int`	`long`	`float`	`double`	`boolean`
`byte`	≈	ω	ωη	ω	ω	ω	ω	-
`short`	η	≈	η	ω	ω	ω	ω	-
`char`	η	η	≈	ω	ω	ω	ω	-
`int`	η	η	η	≈	ω	ω	ω	-
`long`	η	η	η	η	≈	ω	ω	-
`float`	η	η	η	η	η	≈	ω	-
`double`	η	η	η	η	η	η	≈	-
`boolean`	-	-	-	-	-	-	-	≈
`Byte`	⊗	⊗,ω	-	⊗,ω	⊗,ω	⊗,ω	⊗,ω	-
`Short`	-	⊗	-	⊗,ω	⊗,ω	⊗,ω	⊗,ω	-
`Character`	-	-	⊗	⊗,ω	⊗,ω	⊗,ω	⊗,ω	-
`Integer`	-	-	-	⊗	⊗,ω	⊗,ω	⊗,ω	-
`Long`	-	-	-	-	⊗	⊗,ω	⊗,ω	-
`Float`	-	-	-	-	-	⊗	⊗,ω	-
`Double`	-	-	-	-	-	-	⊗	-
`Boolean`	-	-	-	-	-	-	-	⊗
`Object`	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗

139

...

JLS sections, cont.

14.20.3.1 Basic `try`-with-resources 548
14.20.3.2 Extended `try`-with-resources 551
14.21 The `yield` Statement 552
14.22 Unreachable Statements 553
14.30 Patterns 559
14.30.1 Kinds of Patterns 560
14.30.2 Pattern Matching 563
14.30.3 Properties of Patterns 564

15 Expressions 567

15.1 Evaluation, Denotation, and Result 567
15.2 Forms of Expressions 568
15.3 Type of an Expression 569
15.4 Floating-point Expressions 570
15.5 Expressions and Run-Time Checks 574
15.6 Normal and Abrupt Completion of Evaluation 575
15.7 Evaluation Order 577
15.7.1 Evaluate Left-Hand Operand First 577
15.7.2 Evaluate Operands before Operation 579
15.7.3 Evaluation Respects Parentheses and Precedence 580
15.7.4 Argument Lists are Evaluated Left-to-Right 581
15.7.5 Evaluation Order for Other Expressions 582
15.8 Primary Expressions 583
15.8.1 Lexical Literals 584
15.8.2 Class Literals 584
15.8.3 `this` 585
15.8.4 Qualified `this` 586
15.8.5 Parenthesized Expressions 587
15.9 Class Instance Creation Expressions 587
15.9.1 Determining the Class being Instantiated 589
15.9.2 Determining Enclosing Instances 591
15.9.3 Choosing the Constructor and its Arguments 594
15.9.4 Run-Time Evaluation of Class Instance Creation Expressions 598
15.9.5 Anonymous Class Declarations 599
15.9.5.1 Anonymous Constructors 600
15.10 Array Creation and Access Expressions 602
15.10.1 Array Creation Expressions 602
15.10.2 Run-Time Evaluation of Array Creation Expressions 603
15.10.3 Array Access Expressions 606
15.10.4 Run-Time Evaluation of Array Access Expressions 607
15.11 Field Access Expressions 609
15.11.1 Field Access Using a Primary 610
15.11.2 Accessing Superclass Members using `super` 613
15.12 Method Invocation Expressions 614
15.12.1 Compile-Time Step 1: Determine Type to Search 616
15.12.2 Compile-Time Step 2: Determine Method Signature 618

15.12.2.1 Identify Potentially Applicable Methods 624
15.12.2.2 Phase 1: Identify Matching Arity Methods Applicable by Strict Invocation 627
15.12.2.3 Phase 2: Identify Matching Arity Methods Applicable by Loose Invocation 628
15.12.2.4 Phase 3: Identify Methods Applicable by Variable Arity Invocation 629
15.12.2.5 Choosing the Most Specific Method 629
15.12.2.6 Method Invocation Type 634
15.12.3 Compile-Time Step 3: Is the Chosen Method Appropriate? 635
15.12.4 Run-Time Evaluation of Method Invocation 638
15.12.4.1 Compute Target Reference (If Necessary) 638
15.12.4.2 Evaluate Arguments 640
15.12.4.3 Check Accessibility of Type and Method 641
15.12.4.4 Locate Method to Invoke 642
15.12.4.5 Create Frame, Synchronize, Transfer Control 647
15.13 Method Reference Expressions 648
15.13.1 Compile-Time Declaration of a Method Reference 651
15.13.2 Type of a Method Reference 658
15.13.3 Run-Time Evaluation of Method References 659
15.14 Postfix Expressions 662
15.14.1 Expression Names 663
15.14.2 Postfix Increment Operator `++` 663
15.14.3 Postfix Decrement Operator `--` 664
15.15 Unary Operators 664
15.15.1 Prefix Increment Operator `++` 666
15.15.2 Prefix Decrement Operator `--` 666
15.15.3 Unary Plus Operator `+` 667
15.15.4 Unary Minus Operator `-` 667
15.15.5 Bitwise Complement Operator `~` 668
15.15.6 Logical Complement Operator `!` 668
15.16 Cast Expressions 669
15.17 Multiplicative Operators 670
15.17.1 Multiplication Operator `*` 671
15.17.2 Division Operator `/` 671
15.17.3 Remainder Operator `%` 673
15.18 Additive Operators 675
15.18.1 String Concatenation Operator `+` 676
15.18.2 Additive Operators (`+` and `-`) for Numeric Types 678
15.19 Shift Operators 680
15.20 Relational Operators 681
15.20.1 Numerical Comparison Operators `<`, `<=`, `>`, and `>=` 681
15.20.2 The `instanceof` Operator 682
15.21 Equality Operators 685
15.21.1 Numerical Equality Operators `==` and `!=` 685
15.21.2 Boolean Equality Operators `==` and `!=` 686
15.21.3 Reference Equality Operators `==` and `!=` 687
15.22 Bitwise and Logical Operators 687

One effect of conversion and promotion policies

- Allows `javac` to determine how to compile the expression
  `a + b;`
  Based on the types of the expressions, after any necessary conversions, one of:
  - Compilation error
  - `dadd` of `a` and `b`
  - `iadd` of `a` and `b`
  - `fadd` of `a` and `b`
  - `ladd` of `a` and `b`
  - String concatenation sequence of `a` and `b`

Historically, could be `StringBuffer` or `StringBuilder` calls, as of JDK 9 with [JEP 280](): "Indify String Concatenation" use `invokedynamic` via `StringConcatFactory`.

JVMS sections

Table of Contents

1 Introduction 1

1.1 A Bit of History 1
1.2 The Java Virtual Machine 2
1.3 Organization of the Specification 3
1.4 Notation 4
1.5 Feedback 4

2 The Structure of the Java Virtual Machine 5

2.1 The `class` File Format 5
2.2 Data Types 6
2.3 Primitive Types and Values 6
- 2.3.1 Integral Types and Values 7
- 2.3.2 Floating-Point Types and Values 8
- 2.3.3 The `returnAddress` Type and Values 9
- 2.3.4 The `boolean` Type 10

2.4 Reference Types and Values 10
2.5 Run-Time Data Areas 11
- 2.5.1 The `pc` Register 11
- 2.5.2 Java Virtual Machine Stacks 11
- 2.5.3 Heap 12
- 2.5.4 Method Area 13
- 2.5.5 Run-Time Constant Pool 13
- 2.5.6 Native Method Stacks 14

2.6 Frames 15
- 2.6.1 Local Variables 15
- 2.6.2 Operand Stacks 16
- 2.6.3 Dynamic Linking 17
- 2.6.4 Normal Method Invocation Completion 17
- 2.6.5 Abrupt Method Invocation Completion 18

2.7 Representation of Objects 18
2.8 Floating-Point Arithmetic 18
2.9 Special Methods 22
- 2.9.1 Instance Initialization Methods 22
- 2.9.2 Class Initialization Methods 22
- 2.9.3 Signature Polymorphic Methods 23

2.10 Exceptions 24
2.11 Instruction Set Summary 26
- 2.11.1 Types and the Java Virtual Machine 27
- 2.11.2 Load and Store Instructions 29
- 2.11.3 Arithmetic Instructions 30

- 2.11.4 Type Conversion Instructions 31
- 2.11.5 Object Creation and Manipulation 33
- 2.11.6 Operand Stack Management Instructions 34
- 2.11.7 Control Transfer Instructions 34
- 2.11.8 Method Invocation and Return Instructions 34
- 2.11.9 Throwing Exceptions 35
- 2.11.10 Synchronization 35

2.12 Class Libraries 36
2.13 Public Design, Private Implementation 37

3 Compiling for the Java Virtual Machine 39

3.1 Format of Examples 39
3.2 Use of Constants, Local Variables, and Control Constructs 40
3.3 Arithmetic 43
3.4 Accessing the Run-Time Constant Pool 46
3.5 More Control Examples 47
3.6 Receiving Arguments 50
3.7 Invoking Methods 51
3.8 Working with Class Instances 53
3.9 Arrays 55
3.10 Compiling Switches 57
3.11 Operations on the Operand Stack 59
3.12 Throwing and Handling Exceptions 59
3.13 Compiling `finally` 63
3.14 Synchronization 66
3.15 Annotations 67
3.16 Modules 68

4 The `class` File Format 71

4.1 The `ClassFile` Structure 72
4.2 Names 79
- 4.2.1 Binary Class and Interface Names 79
- 4.2.2 Unqualified Names 79
- 4.2.3 Module and Package Names 79

4.3 Descriptors 80
- 4.3.1 Grammar Notation 80
- 4.3.2 Field Descriptors 81
- 4.3.3 Method Descriptors 82

4.4 The Constant Pool 83
- 4.4.1 The `CONSTANT_Class_info` Structure 86
- 4.4.2 The `CONSTANT_Fieldref_info`, `CONSTANT_Methodref_info`, and `CONSTANT_InterfaceMethodref_info` Structures 87
- 4.4.3 The `CONSTANT_String_info` Structure 88
- 4.4.4 The `CONSTANT_Integer_info` and `CONSTANT_Float_info` Structures 88
- 4.4.5 The `CONSTANT_Long_info` and `CONSTANT_Double_info` Structures 90

- 4.4.6 The `CONSTANT_NameAndType_info` Structure 91
- 4.4.7 The `CONSTANT_Utf8_info` Structure 92
- 4.4.8 The `CONSTANT_MethodHandle_info` Structure 94
- 4.4.9 The `CONSTANT_MethodType_info` Structure 96
- 4.4.10 The `CONSTANT_Dynamic_info` and `CONSTANT_InvokeDynamic_info` Structures 96
- 4.4.11 The `CONSTANT_Module_info` Structure 97
- 4.4.12 The `CONSTANT_Package_info` Structure 98

4.5 Fields 99
4.6 Methods 101
4.7 Attributes 105
- 4.7.1 Defining and Naming New Attributes 112
- 4.7.2 The `ConstantValue` Attribute 113
- 4.7.3 The `Code` Attribute 114
- 4.7.4 The `StackMapTable` Attribute 117
- 4.7.5 The `Exceptions` Attribute 125
- 4.7.6 The `InnerClasses` Attribute 126
- 4.7.7 The `EnclosingMethod` Attribute 128
- 4.7.8 The `Synthetic` Attribute 130
- 4.7.9 The `Signature` Attribute 131
  - 4.7.9.1 Signatures 132
- 4.7.10 The `SourceFile` Attribute 136
- 4.7.11 The `SourceDebugExtension` Attribute 137
- 4.7.12 The `LineNumberTable` Attribute 138
- 4.7.13 The `LocalVariableTable` Attribute 139
- 4.7.14 The `LocalVariableTypeTable` Attribute 141
- 4.7.15 The `Deprecated` Attribute 142
- 4.7.16 The `RuntimeVisibleAnnotations` Attribute 143
  - 4.7.16.1 The `element_value` structure 145
- 4.7.17 The `RuntimeInvisibleAnnotations` Attribute 148
- 4.7.18 The `RuntimeVisibleParameterAnnotations` Attribute 149
- 4.7.19 The `RuntimeInvisibleParameterAnnotations` Attribute 151
- 4.7.20 The `RuntimeVisibleTypeAnnotations` Attribute 152
  - 4.7.20.1 The `target_info` union 158
  - 4.7.20.2 The `type_path` structure 163
- 4.7.21 The `RuntimeInvisibleTypeAnnotations` Attribute 169
- 4.7.22 The `AnnotationDefault` Attribute 170
- 4.7.23 The `BootstrapMethods` Attribute 171
- 4.7.24 The `MethodParameters` Attribute 172
- 4.7.25 The `Module` Attribute 174
- 4.7.26 The `ModulePackages` Attribute 181
- 4.7.27 The `ModuleMainClass` Attribute 182
- 4.7.28 The `NestHost` Attribute 183
- 4.7.29 The `NestMembers` Attribute 183
- 4.7.30 The `Record` Attribute 185
- 4.7.31 The `PermittedSubclasses` Attribute 186

4.8 Format Checking 188
4.9 Constraints on Java Virtual Machine Code 188

JVMS sections, cont.

4.9.1 Static Constraints 189
4.9.2 Structural Constraints 192
4.10 Verification of `class` Files 196
4.10.1 Verification by Type Checking 198
4.10.1.1 Accessors for Java Virtual Machine Artifacts 200
4.10.1.2 Verification Type System 204
4.10.1.3 Instruction Representation 208
4.10.1.4 Stack Map Frames and Type Transitions 211
4.10.1.5 Type Checking Abstract and Native Methods 215
4.10.1.6 Type Checking Methods with Code 218
4.10.1.7 Type Checking Load and Store Instructions 226
4.10.1.8 Type Checking for `protected` Members 228
4.10.1.9 Type Checking Instructions 231
4.10.2 Verification by Type Inference 351
4.10.2.1 The Process of Verification by Type Inference 351
4.10.2.2 The Bytecode Verifier 351
4.10.2.3 Values of Types `long` and `double` 354
4.10.2.4 Instance Initialization Methods and Newly Created Objects 355
4.10.2.5 Exceptions and `finally` 356
4.11 Limitations of the Java Virtual Machine 358

5 Loading, Linking, and Initializing 361

5.1 The Run-Time Constant Pool 361
5.2 Java Virtual Machine Startup 364
5.3 Creation and Loading 365
5.3.1 Loading Using the Bootstrap Class Loader 367
5.3.2 Loading Using a User-defined Class Loader 368
5.3.3 Creating Array Classes 369
5.3.4 Loading Constraints 370
5.3.5 Deriving a Class from a `class` File Representation 371
5.3.6 Modules and Layers 375
5.4 Linking 377
5.4.1 Verification 378
5.4.2 Preparation 378
5.4.3 Resolution 379
5.4.3.1 Class and Interface Resolution 381
5.4.3.2 Field Resolution 381
5.4.3.3 Method Resolution 382
5.4.3.4 Interface Method Resolution 384
5.4.3.5 Method Type and Method Handle Resolution 386
5.4.3.6 Dynamically-Computed Constant and Call Site Resolution 390
5.4.4 Access Control 395
5.4.5 Method Overriding 397
5.4.6 Method Selection 398
5.5 Initialization 399

Eliding chapter 6, The Java Virtual Machine Instruction Set, sections.

JVM Class File Structures

- Expand *BaseType* for Field Descriptors for new type (JVMS §4.3.2)
- New constant pool tags (JVMS §4.4)
- Constant value attribute update (JVMS §4.7.2)
- Stack Map updates/verifier updates (JVMS §4.10)
  (How many stack slots to instructions operating on the new types use?)
- Annotation attribute updates, including `element_value` type tag
  (assorted sections in JVMS §4.7.{[16 – 22]}.*)
- …

`double`-related JVM instructions

- `dcmp`<*op*> (`dcmpg`, `dcmpl`)
- `dadd`
- `dsub`
- `dmul`
- `ddiv`
- `dneg`
- `drem` (*not* IEEE 754 remainder)
- `i2d`, `l2d`, `d2i`, `d2l`, `d2f`, `f2d`
- `daload`, `dastore`
- `dconst_`<*d*> (2 opcodes)
- `dload, dload_`<*n*> (4 opcodes)
- `dstore`, `dstore_`<*n*> (4 opcodes)
- `dreturn`
- `ldc2_w`

How many opcodes are used already?

- 3 reserved opcodes (254, 255, 202 (0xca) for breakpoints)
  - About 202 allocated (0 to 201)
  - Therefore, 256 – (202 + 3) = 51 opcodes available
- About 20 opcodes per type (if done like `float` or `double`) ⇒ not many types can be added in this way.
  - Could add modifiers to existing opcodes; NARROW to treat `float` instruction as working on `Float16`, etc.
  - Various tricks could be used to reduce opcodes allocated for a type like `Float16`: after a `Float16` to float conversion, do the actual arithmetic using `float` instructions, and convert back, to `Float16,` etc.
  - But, supporting a type like Decimal64 or Decimal128 directly as a primitive would require a higher opcode allocation

Class library support

- Reflective support
  - Core reflection: `Class.isPrimitive()`, *`FOO`*`.TYPE`, etc.
  - `javax.lang.model.type.PrimitiveType`
  - `java.lang.classfile` updates
- Direct support: base conversion and input/output, mathematical operations, etc.
  - Wrapper class like `java.lang.Double`
  - Other methods in `java.lang.`{`Math`, `StrictMath`}
  - l10n, etc.
- Operations on arrays of primitives, sorting, copying, etc.

Primitive type support overview

- In the JLS and `javac`, literals, operators, conversions, code generation.
- In the VM, various structures to hold constants (constant pool, annotations, etc.), as well as instructions.
- In the libraries, reflective support for the type, support for mathematical operations on the type's values, and arrays of those values.
- Combination: some amount of intrinsification of library functionality

Observations on using a JVM-centric approach

Many limitations

- The remaining easy bytecode space is much smaller than that needed to naturally accommodate the number of types of possible interest.
- Added types in this fashion is centralized to the JDK
  - Significant cross-component collaboration, `javac`, HotSpot, core libraries, etc.”
  - Some aspects of adding new types looks to be closer than $O(n^2)$ rather than $O(n)$; e.g. JLS conversions.
- Preferable to make JLS and JVMS changes approximately once for all types rather than *N* times for each of *N* types added.
- Wouldn’t it be preferable if Java SE could get some new numerics types while third parties could add their own numeric types too?

Growing a Language

Guy Steele's keynote at the 1998 ACM OOPSLA conference, video

Excerpt from *Growing a Language*

“I might say yes to [adding] each one of these [kinds of numbers to the Java programming language] , but it is clear I must say no to all of them.
[…]
To add them all to the Java programing language would be just too much.”
—Guy Steele

Language features recommended in *Growing a Language*

- To make the Java programming language more extensible, add:
  - Generics, done in Java SE 5.0 via JSR 14, GA 2004
  - Lightweight classes / value classes – coming with Valhalla (JEP 401)
  - Operator overloading

Type classes – a mechanism for operator overloading…

…and conversions … and …

- One technique from our functional friends: type classes
- Provide a way to allow opt-in mapping of an operator to a method on a class via a *witness*…
- To achieve the goal of make primitives and library-defined types inter-operate, multiple possible paths.

Numerical type support type classes

- From earlier, language features of built-in numeric types include:
  - Literals
  - Conversions between types
  - Operators
  - Constant folding
- Initial iterations of type classes likely to support
  - Conversions between types
  - Operators
- Later iterations, or may not, support other features:
  - Constant expressions (JLS §15.29) and constant folding at compile time
  - Constants stored in the class file, including for annotation elements

How full is the glass of numeric types with type classes?

- IMO, just with operators and conversions from other types, the glass would be *at least* 80% full.

What are Java's operators?

See JLS §15 Expressions

- Postfix increment and decrement operators {++, --}
- Unary operators: "The operators +, -, ++, --, ~, !, and the cast operator (JLS §15.16) are called the *unary operators*."
- Multiplicative operators:
  "The operators *, /, and % are called the *multiplicative operators*."
- Additive operators:
  "The operators + and - are called the *additive operators*."

What are Java’s operators?, cont.

- Relational operators: “The numerical comparison operators `<`, `>`, `<=`, and `>=`, and the `instanceof` operator, are called the *relational operators*.”
- Equality operators: “The operators `==` (equal to) and `!=` (not equal to) are called the *equality operators*.”
- Shift operators: “The operators `<<` (left shift), `>>` (signed right shift), and `>>>` (unsigned right shift) are called the *shift operators*.”
- Bitwise and Logical Operators: “The bitwise operators and logical operators include the AND operator &, exclusive OR operator ^, and inclusive OR operator `|`.”

What are Java's operators?, cont.

- Other miscellaneous operators: {&&, `||`, `?:`, `instanceof`, `=`, `+=`, `-=`, `*=`, `/=`, &=, `|=`, `^=`, %=, `<<=`, `>>=`, `>>>=`}

Type classes and operator overloading

Working design assumptions

- Assume homogenous typing
  $$\alpha \; op \; \alpha \rightarrow \alpha$$
  is sufficient for operator overloading
  - In other words, *do* not attempt (initially) to support mixed-type operators
  - Some of the effect of mixed-type operators can be implemented via conversions
- Assume only identity-less classes will be eligible for operator overloading
- Assume not all operators will be eligible for overloading. Some compound operators *may* at most be overloaded indirectly via a compiler-generated composite of a non-compound overload; e.g. += built out of +.

Implications of assumptions

- Homogeneity does not necessarily provide ideal support for
  - Complex arithmetic
    (would benefit from direct `double` *op* `complex`, `imaginary` *op* `complex` and
    `imaginary * imaginary` → `double`,
    `imaginary + double` → `complex`, etc.
    Possible, if costly, to implement these more precise checks inside a complex type.)
  - Matrices
    Matrices can be multiplied together *and* multiplied by a scalar of the type of the entities of the matrix. If only Matrix * Matrix multiply is supported, 1×1 matrix and a scalar are distinct concepts, but systems can work out conversions/coercions between the two.

Implications for `BigInteger` and `BigDecimal`

- The `BigInteger` and `BigDecimal` classes are non-final and have public constructors and there are known to be subclasses in use for understandable and valid reasons.
  "It was not widely understood that immutable classes had to be effectively final when `BigInteger` and `BigDecimal` were written…"
  Effective Java, 1st edition, Item 13: Favor Immutability, 2001
- Challenging to migrate these classes to *not* have identity within the JDK's existing compatibility policies and practices.
- *Acknowledged to be very desirable for the platform to have arbitrary precision integer and decimal arithmetic usable with operators.*

`Float16` Retrospective and Prospective

`Float16` Functionality in the JDK

`jdk.incubator.vector.Float16` as of JDK 24

- float16 is a 16-bit IEEE 754 floating-point format (called “binary16” in the standard); cf. bfloat16 from Google which is not covered by a standards body).
- `Float16` in the incubator provides a fully functional software implementation
  - Favored simple, easy-to-understand code that doesn’t perform excessively poorly
- HotSpot intrinsifications in mainline JDK 25 for x64, aarch64, and RISC-V; thanks to OpenJDK collaborators, especially Intel being on the vanguard of adding this support!

`Float16` by the numbers

- Work originally done in the base module on a Valhalla branch; ported to `jdk.incubator.vector` in JDK 24.
- Initial push:
  - ≈2,500 lines of code for `Float16` (including comments and specification)
  - ≈100 LOC of supporting `Float16Consts`
  - ≈1,000 LOC for regression tests
- Small, well-understood, refactoring would be appropriate for `Float16` to be promoted from an incubator to the base module.

Envisioned update allowing `Float16` operators

Picture copyright 2012 Joseph D. Darcy All rights reserved.

- Change the type declaration to a `value` class
- Add a witness declaration to the class:

```
public static witness Monoid<Float16> MONOID =
   new Monoid<>() {
       @Override
       public Float16 zero() { return Float16.valueOf(0.0f); }

       @Override
       public Float16 add(Float16 b1, Float16 b2) {
           return Float16.add(b1, b2); // static add method
       }};
```

- A witness for an interface with more methods would be longer, etc.
- Expect *instance* methods on the witness implementation to call *static* methods of the numeric type.

Source code to class file (prototype)

For source code like:

```
Float16 a = Float16.valueOf("1.0");
Float16 b = 2.0f // type class conversion
Float16 sum = a + b;
```

generated class file instructions for the + operation:

```
10: ldc              #13 // Dynamic #0:__witness:Ljava/lang/runtime/Monoid;
12: aload_1
13: aload_2
14: invokeinterface #17,  3
// InterfaceMethod java/lang/runtime/Monoid.add:(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
```

(If the feature was being done pre-indy/condy, would use a different translation.)

Functional parity with `Float/Double`

- Fields
  - POSITIVE_INFINITY
  - NEGATIVE_INFINITY
  - NaN
  - MAX_VALUE
  - MIN_NORMAL
  - MIN_VALUE
  - SIZE, PRECISION
  - MAX_EXPONENT
  - MIN_EXPONENT
  - BYTES
- Methods
  - toHexString()
  - isNaN()
  - isInfinite()
  - isFinite()
- But, *no* public constructors
  - use `valueOf()` factories instead

Other functionality

`Float16` self-hosts functionality found in `Math`/`StrictMath` for `float` and `double`

- External to the JDK numeric types can't add methods in `java.*`
- `Math`/`StrictMath` methods:
  - `min()`, `max()`, `sqrt()`, `fma()` i.e. ($a \cdot b + c$) with one rounding error, `abs()`, `getExponent()`, `ulp()`, `nextUp()`, `nextDown()`, `scalb()`, `copySign()`, `signum()`
  - Many of the above are IEEE 754 operations too
- New methods corresponding to operators:
  - `add()`, `subtract()`, `multiply()`, `divide()`
  - `negate()`
  - (remainder/IEEERemainder functionality tbd)

Did *not* add…

- Format-specific sin, cos, tan, …
  - For now, could use `double` version of those methods, e.g.
    `Float16 sinOf_f16 =`
    `Float16.valueOf(Math.sin(f16.doubleValue()))`
  - Do not have these methods specific for `float` either (unliked the C math library)

Easy things that went as expected

- Simple implementation of an arithmetic operation:

```
public static Float16 add(Float16 addend, Float16 augend) {
    return valueOf(addend.floatValue() + augend.floatValue());
}
```

- Potential hazards of *double-rounding* avoided when the $2p + 2$ property holds for two formats and the {+, –, *, /} and sqrt operations
  - Can implement operations in the narrower format by converting to the wider format, doing the arithmetic in the wider format, and converting back to the narrow format
  - For IEEE 754-style types, holds for both binary and decimal bases.
  - Citations of this property included in comments of `Float16.java`.

Small things that took longer than expected: `fma` saga

Method specification

```java
/**
 * Returns the fused multiply add of the three arguments; that is,
 * returns the exact product of the first two arguments summed
 * with the third argument and then rounded once to the nearest
 * {@code Float16}.
 *
 * The handling of zeros, NaN, infinities, and other special cases
 * by this method is analogous to the handling of those cases by
 * {@link Math#fma(float, float, float)}.
 *
 * @apiNote This method corresponds to the fusedMultiplyAdd
 * operation defined in IEEE 754.
 *
 * @param a a value
 * @param b a value
 * @param c a value
 *
 * @return (<i>a</i> &times; <i>b</i> + <i>c</i>)
 * computed, as if with unlimited range and precision, and rounded
 * once to the nearest {@code Float16} value
 *
 * @see Math#fma(float, float, float)
 * @see Math#fma(double, double, double)
 */
public static Float16 fma(Float16 a, Float16 b, Float16 c) {
   /*
    * The double format has sufficient precision that a Float16
    * fma can be computed by doing the arithmetic in double, with
    * one rounding error for the sum, and then a second rounding
    * error to round the product-sum to Float16. In pseudocode,
    * this method is equivalent to the following code, assuming
    * casting was defined between Float16 and double:
    *
    * double product = (double)a * (double)b;  // Always exact
    * double productSum = product + (double)c;
    * return (Float16)produdctSum;
    *
    * (Note that a similar relationship does *not* hold between
    * the double format and computing a float fma.)
    *
    * Below is a sketch of the proof that simple double
    * arithmetic can be used to implement a correctly rounded
    * Float16 fma.
    *
    * ----------------------
    *
    * As preliminaries, the handling of NaN and Infinity
    * arguments falls out as a consequence of general operation
    * of non-finite values by double * and +. Any NaN argument to
    * fma will lead to a NaN result, infinities will propagate or
    * get turned into NaN as appropriate, etc.
    *
    * One or more zero arguments are also handled correctly,
    * including the propagation of the sign of zero if all three
    * arguments are zero.
    *
    * The double format has 53 logical bits of precision and its
    * exponent range goes from -1022 to 1023. The Float16 format
    * has 11 bits of logical precision and its exponent range
    * goes from -14 to 15. Therefore, the individual powers of 2
    * representable in the Float16 format range from the
    * subnormal 2^(-24), MIN_VALUE, to 2^15, the leading bit
    * position of MAX_VALUE.
    *
    * In cases where the numerical value of (a * b) + c is
    * computed exactly in a double, after a single rounding to
    * Float16, the result is necessarily correct since the one
    * double -> Float16 conversion is the only source of
    * numerical error. The operation as implemented in those
    * cases would be equivalent to rounding the infinitely precise
    * value to the result format, etc.
```

Explanation of implementation

```java
    * However, for some inputs, the intermediate product-sum will
    * *not* be exact and additional analysis is needed to justify
    * not having any corrective computation to compensate for
    * intermediate rounding errors.
    *
    * The following analysis will rely on the range of bit
    * positions representable in the intermediate
    * product-sum.
    *
    * For the product a*b of Float16 inputs, the range of
    * exponents for nonzero finite results goes from 2^(-48)
    * (from MIN_VALUE squared) to 2^31 (from the exact value of
    * MAX_VALUE squared). This full range of exponent positions,
    * (31 -(-48) + 1 ) = 80 exceeds the precision of
    * double. However, only the product a*b can exceed the
    * exponent range of Float16. Therefore, there are three main
    * cases to consider:
    *
    * 1) Large exponent product, exponent > Float16.MAX_EXPONENT
    *
    * The magnitude of the overflow threshold for Float16 is:
    *
    * MAX_VALUE + 1/2 * ulp(MAX_VALUE) =  0x1.ffcp15 + 0x0.002p15 = 0x1.ffep15
    *
    * Therefore, for any product greater than or equal in
    * magnitude to (0x1.ffep15 + MAX_VALUE) = 0x1.ffdp16, the
    * final fma result will certainly overflow to infinity (under
    * round to nearest) since adding in c = -MAX_VALUE will still
    * be at or above the overflow threshold.
    *
    * If the exponent of the product is 15 or 16, the smallest
    * subnormal Float16 is 2^-24 and the ~40 bit wide range bit
    * positions would fit in a single double exactly.
    *
    * 2) Exponent of product is within the range of _normal_
    * Float16 values; Float16.MIN_EXPONENT <=  exponent <= Float16.MAX_EXPONENT
    *
    * The exact product has at most 22 contiguous bits in its
    * logical significand. The third number being added in has at
    * most 11 contiguous bits in its significand and the lowest
    * bit position that could be set is 2^(-24). Therefore, when
    * the product has the maximum in-range exponent, 2^15, a
    * single double has enough precision to hold down to the
    * smallest subnormal bit position, 15 - (-24) + 1 = 40 <
    * 53. If the product was large and rounded up, increasing the
    * exponent, when the third operand was added, this would
    * cause the exponent to go up to 16, which is within the
    * range of double, so the product-sum is exact and will be
    * correct when rounded to Float16.
    *
    * 3) Exponent of product is in the range of subnormal values or smaller,
    * exponent < Float16.MIN_EXPONENT
    *
    * The smallest exponent possible in a product is 2^(-48).
    * For moderately sized Float16 values added to the product,
    * with an exponent of about 4, the sum will not be
    * exact. Therefore, an analysis is needed to determine if the
    * double-rounding is benign or would lead to a different
    * final Float16 result. Double rounding can lead to a
    * different result in two cases:
    *
    * 1) The first rounding from the exact value to the extended
    * precision (here `double`) happens to be directed _toward_ 0
    * to a value exactly midway between two adjacent working
    * precision (here `Float16`) values, followed by a second
    * rounding from there which again happens to be directed
    * _toward_ 0 to one of these values (the one with lesser
    * magnitude).  A single rounding from the exact value to the
    * working precision, in contrast, rounds to the value with
    * larger magnitude.
    *
    * 2) Symmetrically, the first rounding to the extended
    * precision happens to be directed _away_ from 0 to a value
    * exactly midway between two adjacent working precision
    * values, followed by a second rounding from there which
    * again happens to be directed _away_ from 0 to one of these
    * values (the one with larger magnitude).  However, a single
    * rounding from the exact value to the working precision
    * rounds to the value with lesser magnitude.
    *
    * If the double rounding occurs in other cases, it is
    * innocuous, returning the same value as a single rounding to
    * the final format. Therefore, it is sufficient to show that
    * the first rounding to double does not occur at the midpoint
    * of two adjacent Float16 values:
    *
    * 1) If a, b and c have the same sign, the sum a*b + c has a
    * significand with a large gap of 20 or more 0s between the
    * bits of the significand of c to the left (at most 11 bits)
    * and those of the product a*b to the right (at most 22
    * bits).  The rounding bit for the final working precision of
    * `float16` is the leftmost 0 in the gap.
    *
    *   a) If rounding to `double` is directed toward 0, all the
    *   0s in the gap are preserved, thus the `Float16` rounding
    *   bit is unaffected and remains 0. This means that the
    *   `double` value is _not_ the midpoint of two adjacent
    *   `float16` values, so double rounding is harmless.
    *
    *   b) If rounding to `double` is directed away form 0, the
    *   rightmost 0 in the gap might be replaced by a 1, but the
    *   others are unaffected, including the `float16` rounding
    *   bit. Again, this shows that the `double` value is _not_
    *   the midpoint of two adjacent `float16` values, and double
    *   rounding is innocuous.
    *
    * 2) If a, b and c have opposite signs, in the sum a*b + c
    * the long gap of 0s above is replaced by a long gap of
    * 1s. The `float16` rounding bit is the leftmost 1 in the
    * gap, or the second leftmost 1 iff c is a power of 2. In
    * both cases, the rounding bit is followed by at least
    * another 1.
    *
    *   a) If rounding to `double` is directed toward 0, the
    *   `float16` rounding bit and its follower are preserved and
    *   both 1, so the `double` value is _not_ the midpoint of
    *   two adjacent `float16` values, and double rounding is
    *   harmless.
    *
    *   b) If rounding to `double` is directed away from 0, the
    *   `float16` rounding bit and its follower are either
    *   preserved (both 1), or both switch to 0. Either way, the
    *   `double` value is again _not_ the midpoint of two
    *   adjacent `float16` values, and double rounding is
    *   harmless.
    */
```

Implementation

```java
    // product is numerically exact in float before the cast to
    // double; not necessary to widen to double before the
    // multiply.
    double product = (double)(a.floatValue() * b.floatValue());
    return valueOf(product + c.doubleValue());
}
```

HT Raffaello Giulietti for helping to work out the explanation details.

A way to implement `Float16.fma` with less thinking

- Screen for NaN, infinity, and signed zero arguments, then for most Float16 values a, b, and c use something like
```
Float16.valueOf(new BigDecimal(a.floatValue()*b.floatValue()))
                                .add(c.floatValue()))
```
- The `BigDecimal` to `Float16` conversion is nontrivial since the 2*p*+2 property does not hold for that operation, meaning that a conversion chain
  `BigDecimal` → `float` → `Float16`
  is *not* equivalent to
  `BigDecimal` → `Float16`.
- (The above was not used as a transitory implementation because we didn't implement the `BigDecimal` → `Float16` conversion before implementing `fma`.)

Bigger-than-small items that took effort

- `BigDecimal` ↔ `Float16` conversions; many operations can have a correct-if-slow implementation operating in an “exact” type and then rounding once to the final precision
- Properly rounding base conversion
  - Shortest-string binary → decimal conversion is dependent not only on the numerical value but also on the precision of the *format*
  - Decimal → binary conversion dependent on precision too ($2p + 2$ doesn’t work in all cases for base conversion)

Consider implementing a narrow binary floating-point type

Such as `bfloat` or the logically 19-bit TensorFloat-32

- Basic arithmetic operators should be straightforward.
- Some operators like `fma` may need more effort.
- Correctly-rounded binary ↔ decimal base conversion can be tricky.

Possible platform infrastructure for adding numerical types

Including outside of the platform

- Candidate platform infrastructure to ease such writing numerical types:
  - A `BigBinary` class to host higher-precision values or ease analysis.
    Could be just even if only to support initial implementations/bootstrapping, it doesn't necessarily have to be super fast.
  - `BigDecimal` and `BigBinary` operations that round at *exponent* boundaries too.
  - Include `java.lang.runtime`-esque support for decimal ↔ binary conversion at different precisions i.e. don't expect users to have implement "shortest string" semantics themselves.
- Suggest authors start with conversion to/from Big{$BASE} as an early or bootstrapping step, gives a lot of functionality, at possibly slow performance, akin to the Zero interpreter in HotSpot.

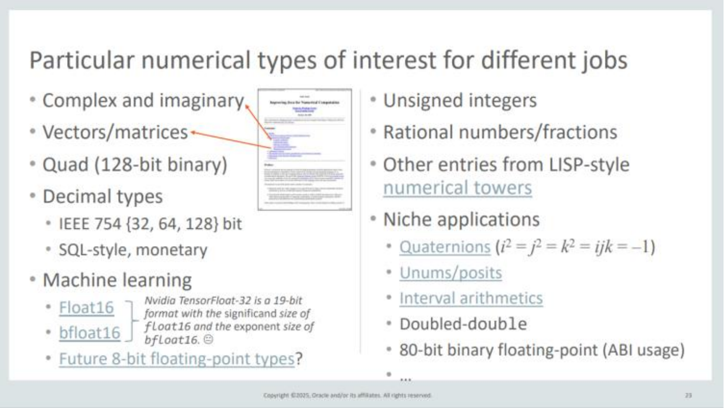
Current thoughts on modeling numerics

What about `java.lang.Number`?

- Very little “numeric” about `java.lang.Number`
- Operationally just means “can be converted to a primitive type”; doesn’t define any operations on the “number” type per se.
- `Number` specification doesn’t strictly require results of the different *`primitive`*`Value()` conversion methods to be consistent; e.g. `myNumber.intValue()` is *not* required to be equal to `(int)(myNumber.doubleValue())`).
- Would be better as an interface rather than an abstract class (slow interface dispatch on early JVM implementations?)
- *Not* a solid foundation for future work

Principle Components Analysis

- Looking over the numeric types of possible interest:
  - Something that acts *more or less* like an integer (`BigInteger`/bigger-than-`long` integer, unsigned integers, …)
  - Something that acts *more or less* like a real number (floating-point types, complex, decimal, …)



- (In a two-element approximation vectors/matrices are *closer* to real numbers than integers)
- Most, but not all, of these kinds of numbers are *ordered*. (Mathematically, the complex field $\mathbb{C}$ is *not* ordered.)

In terms of usage

- Types where it is reasonable to use {+, -, *, /}
- Types where it is reasonable to use {<, <=, >, >=}
- For Integral types
  - Shift operators: {<<, >> , >>>}
  - Bitwise operators: {~, &, ^, |}

Design point: unsigned integers and integers

What about operations like unary negation?

- One approach for unary negation, from JLS §15.15.4:
  "For integer values, negation is the same as subtraction from zero."
- *If* negation is defined for unsigned integers, should it be defined this way? How about unconditionally throwing an exception?
- If negation is *not* defined for unsigned integers, don't have to worry about this detail.
- Design philosophies: Lumpers vs Splitters (a tale as old as taxonomies)
  - Splitter solution: `UnsignedIntegral` a separate superinterface of `Integral`
  - Lumper solution: single `Integral` used by both signed and unsigned integral classes

Work-in-progress numeric hierarchy design

- High-level interfaces associated with operators:
  - `public interface Numerics<NT> extends java.lang.runtime.Monoid<NT> {/* +, -, *, / */}`
  - `public interface OrderedComparison<OC> { /* <, <=, >, >= */}`
  - `public interface Integral<IT> extends Numerics<IT>, OrderedComparison<IT> {…}`
  - ```
    public interface StandardFloatingPoint<SFP>
        extends Numerics<SFP>, OrderedComparison<SFP> { /* IEEE 754-style */}
    ```
  - …
- Particular uses
  - `UnsignedInt` with an `Integral` witness
  - Bigger-than-`long` integer with an `Integral` witness
  - Float16 and `Decimal128` with `StandardFloatingPoint` witnesses
  - …
    ```

Speculative indicators for algebraic properties

- Interfaces to declare various algebraic properties hold (placeholder names to minimize conflicts with existing types):
  - `AlebraicGroup<AG> /* extends Monoid<AG> */`
  - `AlebraicRing<AR>  /* extends Numerics<AG> */`
  - `AlebraicField<AF> /* extends Numerics<AF> */`
- Could be finer-grained separation for properties of add and multiply, etc.
- Could be used to enable transformations by code generators and VMs.

Design considerations in other potential new numeric types: complex and imaginary numbers

Why look at complex numbers?

- Useful for engineering calculations (electricity, etc.)
- Useful mathematically, complex numbers are needed to represent all the solutions to polynomials with real coefficients
  (i.e. fundamental theorem of algebra, a degree *n* polynomial will have *n* roots, counted with multiplicity, etc.)
- Commonly used an introductory example for abstract data types, *but* more advanced usage pushes on language and library support.

CS 101 design considerations for a complex number type

As an abstract data type

- Assume the textbook arithmetic formulas are in use
- How to model a complex number: `class`, `record`, `value class`?
  - `final` vs non-`final` on the type-level
  - Access level of methods, `static` vs instance declaration of particular methods
  - Constructors vs. factories, etc.
- Should the model be real and imaginary components or magnitude and phase/angle components?
- …

Different categories of design issues

Not quite independent categories of concern

- Language and platform support issues and touch points
  - Operator overloading:
    - Is an ordered imposed to allow comparison operators?
    - Is an operator defined for complex conjugation?
  - Conversions: is there a distinct imaginary type?
  - Literals
  - Constant folding
  - Pattern matching
  - Relation to complex support on other platforms, including C, C#, Swift, etc.

Different categories of design issues, cont.

Not quite independent categories of concern

- Just as there is real analysis for real numbers, there is complex analysis for complex numbers.
- Modeling issues
  - How are infinities added to complex numbers?
  - How is NaN represented?
  - Is the end goal `Complex`, or `Complex<double>` with separate `Complex<float>` and perhaps Complex`<StandardFloatingPoint>`?
- Numerical Issues
  - Extent of math library support (sin, cos, tan, …) and *branch cuts*
  - Defining error bounds and semantics for arithmetic operations

Example: complex multiply

- Minimal javadoc spec: "Returns the product of the complex arguments."
- Textbook formula: given $(a + b \cdot i)$ and $(c + d \cdot i)$, their product is:
  $(ac - bd) + (ad + bc) \cdot i$
  - *"There be (numerical) dragons."* Cancelation, overflow, etc., even without considering special value handling.
  - How accurate is the computed product in a norm-wise sense?
  - How accurate are the computed real and imaginary components of the product compared to the exact real and imaginary components?
  - Can users expect `(c1*c2)` and `(c2*c1)` to be equivalent?
- There are formulas with high per-component accuracy and commutative multiply, at the cost of additional floating-point ops, including `fma`.

Goal: users of the Java platform should have few occasions to need to understand the error behavior of complex arithmetic operations

Summary

- Many numeric types of interest for practical programming tasks.
- Working on platform features – spanning language, VM, and libraries – to enable additional easy-to-use, well-integrated numeric types in the JDK and broader Java ecosystem.

Q &A

Slides of this presentation:
https://github.com/jddarcy/SpeakingArchive/blob/master/JVMLS-2025-Numerics.pdf
`@jddarcy`