Kleis for Rust and Java Programmers

A Practical Guide for Engineers

Overview

Kleis is a statically typed, mathematical programming language that combines the precision of type theory with the algebraic expressiveness of modern functional languages. This guide explains Kleis concepts by drawing parallels with constructs familiar to experienced **Rust** and **Java** developers.

The goal is not to repeat the formal specification, but to give you *intuition*:

- If you know Rust traits or Java interfaces, you already understand Kleis `structure`.
- If you know Rust enums or Java sealed classes, you already understand Kleis algebraic data types.
- If you know Rust's `match` or Java's `switch` expressions, you already understand Kleis pattern matching.
- If you understand generics and type bounds, you already understand Kleis parametric kinds.

1 Structures = Traits / Interfaces

Rust:

```
trait Add {
    fn add(self, other: Self) -> Self;
}
```

Java:

```
interface Add<T> {
    T add(T other);
}
```

Kleis:

```
structure Add(T) {
    operation add : T → T → T
}
```

Key correspondences

Rust trait	Kleis structure
Java interface	Kleis structure
Trait method	Kleis operation
Trait bound	Kleis parameter kind / constraint

A Kleis `structure` is conceptually the same as a Rust `trait` or Java `interface`, but with mathematical intent: the operations are meant to satisfy axioms.

For example, a Monoid in Kleis:

```
structure Monoid(M) {
    operation (•) : M → M → M
    element e : M

    axiom associativity:
      ∀(x y z : M). (x • y) • z = x • (y • z)
}
```

Rust and Java cannot express axioms, but the structural analogy remains.

2 implements = impl blocks / interface implementation

Rust:

```
impl Add for i32 {
    fn add(self, other: i32) -> i32 { self + other }
}
```

Java:

```
class MyInt implements Add<Integer> {
    Integer add(Integer x, Integer y) { return x + y; }
}
```

Kleis:

```
implements Add(ℝ) {
    operation add = builtin_add
}
```

Key idea

An `implements` block in Kleis is exactly an `impl` block in Rust: it says "for this type, here is how the structure's operations are executed."

Kleis blends the clarity of Rust's "one impl per trait/type" with the mathematical rigor of specifying laws (axioms).

3 Algebraic Data Types = Rust enums / Java sealed classes

Rust:

```
enum Option<T> { None, Some(T) }
```

Java:

```
sealed interface Option<T>
    permits None<T>, Some<T> {}

record None<T>() implements Option<T> {}
record Some<T>(T value) implements Option<T> {}
```

Kleis:

```
data Option(T) = None | Some(T)
```

Key correspondences

Rust enum	Kleis algebraic data type (ADT)
Java sealed class	Kleis ADT
Enum variant	ADT constructor
Record fields	constructor fields

Kleis ADTs are *fully parametric* and integrate with its type inference system.

4 Pattern matching = Rust match / Java switch expressions

Rust:

```
match x {
    Some(v) => v,
    None => 0,
}
```

Java 21+:

```
switch(x) {
    case Some(var v) -> v;
    case None        -> 0;
}
```

Kleis:

```
match x {
    Some(v) => v
  | None    => 0
}
```

What Kleis adds

- Exhaustiveness checking (like Rust)
- Non-redundancy checking
- Nested pattern matching
- Constructor-level inference

Pattern matching is a central mechanism in Kleis for deconstructing ADTs and defining semantics purely through symbolic structure.

5 Generics and Kinds = Rust traits bounds / Java generics constraints

Rust:

```
fn norm<V: VectorSpace>(v: V) -> f64
```

Java:

```
<T extends VectorSpace<T>>
double norm(T v)
```

Kleis:

```
operation norm :
  ∀(V : Type) (F : Field).
    VectorSpace(V) over Field(F) ⇒ V → F
```

Key ideas

- Kleis quantifiers (∀) generalize Rust/Java generics.
- Kleis kinds classify type parameters (e.g. $Type, \mathbb{N}, Type \to Type$).
- Constraints correspond to Rust trait bounds.

Kleis type inference is Hindley–Milner–based, so type arguments are often omitted and recovered automatically.

6 Structures with "extends" = trait inheritance / interface inheritance

Rust has trait inheritance:

```
trait Group: Monoid {
    fn inv(&self) -> Self;
}
```

Java has interface inheritance:

```
interface Group extends Monoid {
    T invert(T x);
}
```

Kleis:

```
structure Group(G) extends Monoid(G) {
    operation inv : G → G
}
```

Key idea

`extends` forms *algebraic hierarchies*, not class hierarchies.
Examples:

$$\text{Semigroup} \subseteq \text{Monoid} \subseteq \text{Group} \subseteq \text{AbelianGroup}$$

Mathematically this is elegant, and for programmers the mental model is Rust's trait inheritance.

7 "over Field(F)" = associated traits / context parameters

Rust equivalent idea:

```
trait VectorSpace<F: Field> {
    type Vector;
}
```

Java:

```
interface VectorSpace<V,F extends Field<F>> { ... }
```

Kleis:

```
structure VectorSpace(V) over Field(F) {
    operation (+) : V → V → V
    operation (·) : F → V → V
}
```

Mental model for programmers

"`over Field(F)`" means:

¿ This structure depends on another structure instance (a field instance), ¿ just like a Rust trait might require another trait for its associated type.

8 Nested structures = inner traits / inner interfaces (but better)

Java:

```
interface Ring {
    interface Additive { ... }
    interface Multiplicative { ... }
}
```

Rust:

```
trait Ring {
    type Additive: AbelianGroup;
    type Multiplicative: Monoid;
}
```

Kleis:

```
structure Ring(R) {
    structure additive : AbelianGroup(R) { ... }
    structure multiplicative : Monoid(R) { ... }
}
```

Why Kleis is cleaner

Rust and Java approximate mathematical hierarchies; Kleis expresses them *exactly*. Nested structures model multi-operation objects (rings, fields, vector spaces) in their natural form.

9 Matrices as a motivating case

Rust:

```
struct Matrix<M,N,T> { data: [[T; N]; M] }
```

Java:

```
class Matrix<T> { T[][] data; }
```

Kleis:

```
data Matrix(m: Nat, n: Nat, T)
```

Kleis can express:

- dimension-safe multiplication,
- polymorphic matrix types,
- block matrices,
- vector spaces over fields.

Dimension correctness = no run-time errors

$$\texttt{operation multiply} : Matrix(m, n, T) \to Matrix(n, p, T) \to Matrix(m, p, T)$$

Kleis enforces this at compile time, unlike Rust or Java without custom type machinery.

10 Summary Table

Concept	Rust / Java analogue	Kleis feature
Trait / Interface	`trait`, `interface`	`structure`
Impl block	`impl`, `implements`	`implements`
Enum / Sealed class	`enum`, sealed types	`data` ADT
Pattern match	`match`, `switch`	Kleis `match`
Trait bounds	trait bounds, extends	constraints, kinds
Nested traits	inner interfaces	nested structures
Laws	none native	axioms
Generic types	generics	polymorphic types, kinds

Closing Remarks

Rust and Java programmers will find Kleis familiar at the surface level: structures behave like traits, implementations behave like `impl` blocks, and data types mirror enums or sealed classes.

But Kleis takes the next step:

- mathematics becomes first-class,
- algebraic laws are part of the type system,
- polymorphism is principled and expressive,
- pattern matching is exhaustive and safe,
- parametric kinds allow type-level mathematics.

Kleis is thus both *more general* than a conventional programming language and *more precise* than a typical proof assistant: it lets expert programmers write mathematics like software and software like an algebraic structure.