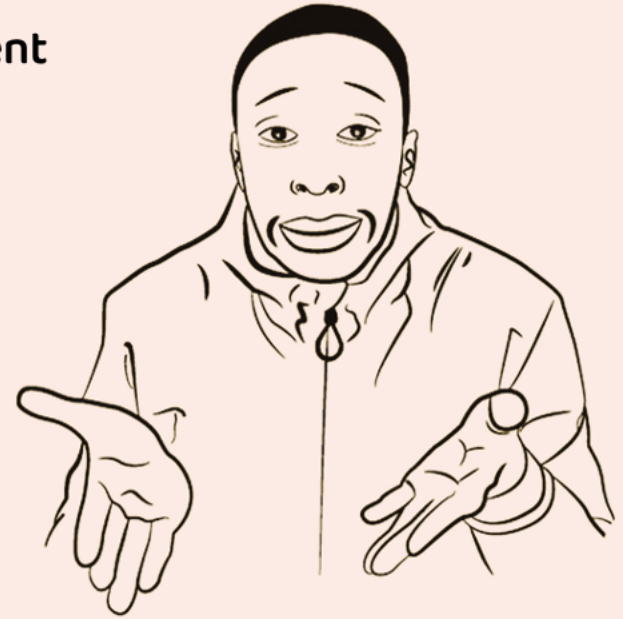
From Java 25

Module Import Declarations

eazy bytes

With this feature, you can import an entire module, like `java.base`, in one statement (e.g., `import module java.base`), which includes all packages it exports, such as `java.util` and `java.io`, eliminating the need for individual package imports.



```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

public class ImportModule {

    public static void main(String[] args) {...}

}
```

Equivalent import code. So simple....

```java
import module java.base;

public class ImportModule {

    public static void main(String[] args) {...}

}
```

From Java 25

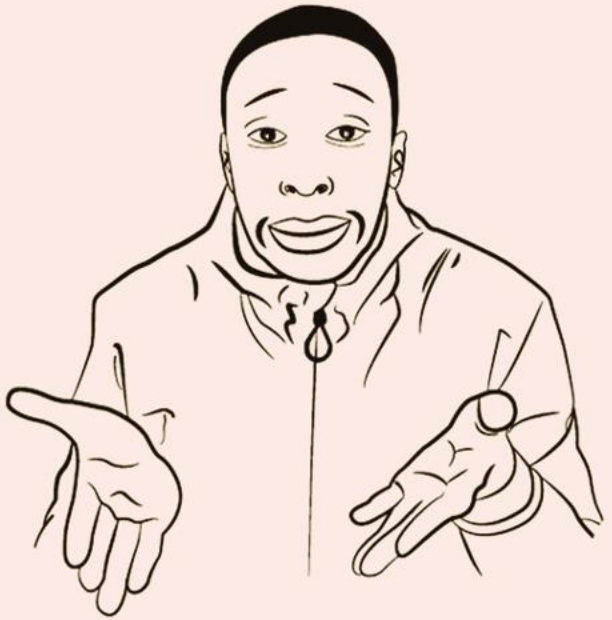
Module Import Declarations

eazy bytes

❖ Ambiguous imports

When importing a module, multiple packages may include classes with the same simple name, causing ambiguity and a compile-time error. For example,

```
import module java.base;  // exports java.util.Date
import module java.sql;   // exports java.sql.Date

Date d = ...              // Error - Ambiguous name!
```

To resolve this, use a single-type import:

```
import module java.base;  // exports java.util.Date
import module java.sql;   // exports java.sql.Date
import java.sql.Date;     // Resolves the ambiguity

Date d = ...              // Now refers to java.sql.Date
```

From Java 25

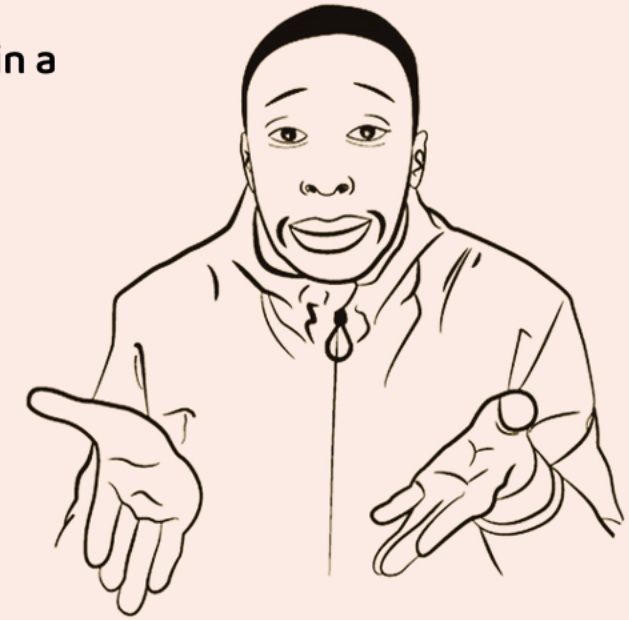
Module Import Declarations

eazy bytes

- **Performance impact** – Whether you import a specific class or an entire package or an entire module, only the classes you actually use in your code are loaded at runtime. So there will be no performance issues with importing entire module.

- **Compact Source file** - The java.base module is automatically imported on-demand in a compact source file.

- **Unnamed module** - The import module statement requires a module name, so packages from the unnamed module (i.e., the classpath) cannot be imported.



Module Import Declarations

eazy bytes

Grouping Import Declarations

Grouping import declarations improves readability of source files. Keeps related imports together. Order should follow shadowing behavior:

- Module imports (least specific)
- Package imports (on-demand *)
- Single-type imports (most specific)


```
// Module imports
import module java.sql;
import module java.logging;

// Package imports
import java.util.*;
import java.time.*;

// Single-type imports
import java.util.List;
import java.time.LocalDate;

class EmployeeApp { ... }
```

From Java 25

Compact Source Files and Instance Main Methods

eazy bytes

The Java team is making the language more beginner-friendly with features like Compact Source Files and Instance Main Methods.

Beginners can now start coding without first learning advanced Java concepts, while experienced programmers can quickly prototype ideas with less boilerplate. The code can then grow and evolve as skills and applications expand.

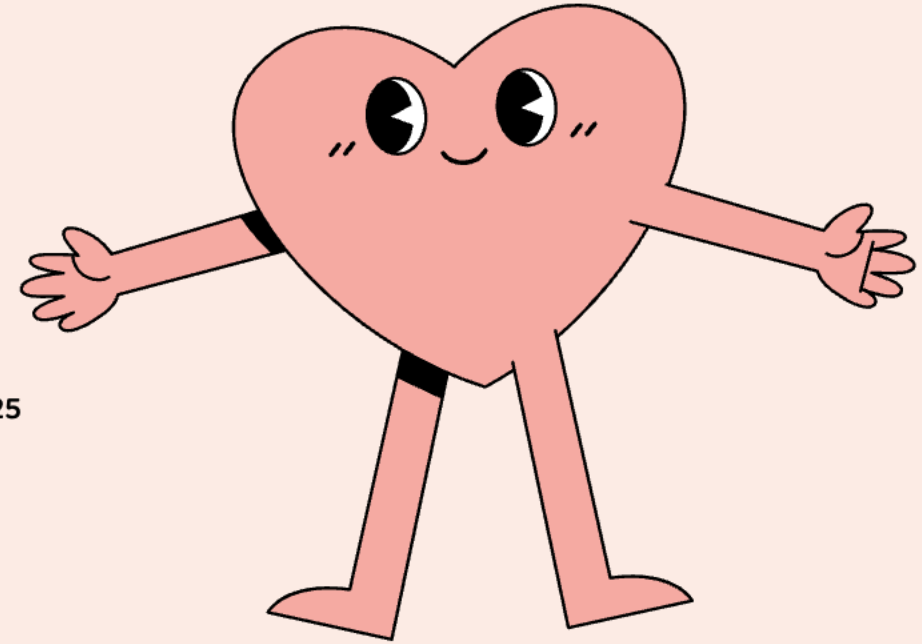
```java
public class HelloWorld {

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

Classic HelloWorld Program before Java 25

Equivalent Compact code from Java 25

```java
void main() {
    IO.println("Hello World!");
}
```

A key advantage of the minimal main() method without an explicit class is that the code stays concise and focused, introducing fewer concepts or keywords that might otherwise overwhelm beginners.



From Java 25

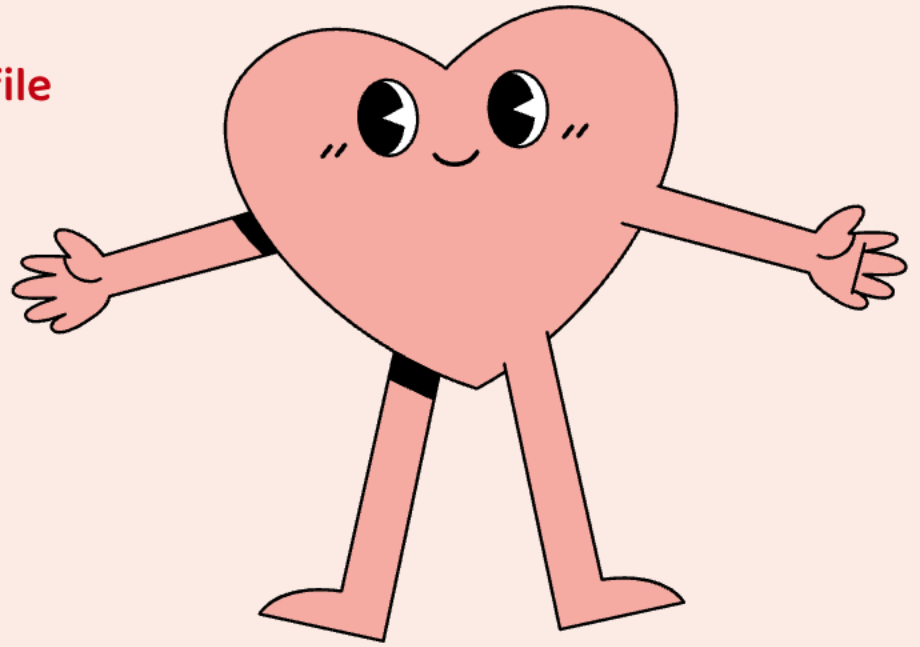
Compact Source Files and Instance Main Methods

eazy bytes

The term instance main() method refers to a main method declared without the static keyword. Traditionally, main was defined as static so it could run without creating an object of its class. In this new approach, main is treated as an instance method, meaning it no longer requires the static modifier.

Different Variations of the main method allowed in the Compact Source file

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

When a class has both a main(String[] args) method and a no-argument main() method, the JVM launcher gives preference to the main(String[] args) method as the program's entry point.

From Java 25

Compact Source Files and Instance Main Methods

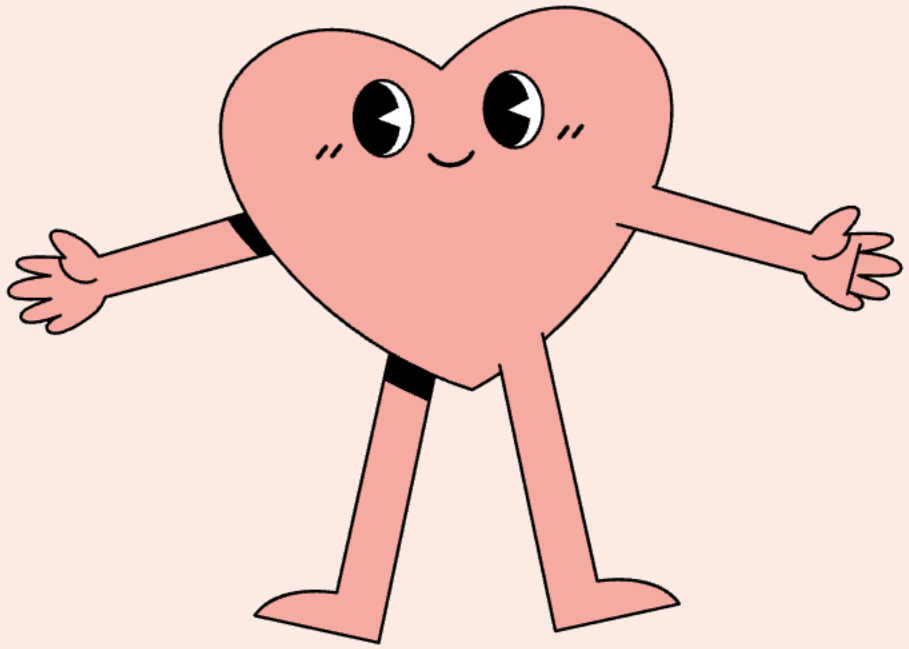
eazy bytes

Console interaction got simpler

Console interaction, like printing or reading messages, is now easier for new Java developers.
Compact Source files can use IO.println(), IO.print(), and IO.readln() for console output and input.

These methods are part of the new java.lang.IO class. Since the IO class resides in the java.lang package, it can be used without an import in any Java program. This applies to all programs, not just those in compact source files or those that declare instance main methods;

```
void main() {
    var name = IO.readln("Enter your name:");
    IO.println("Hello " + name);
}
```

From Java 25

Compact Source Files and Instance Main Methods

eazy bytes

Implicit Class in Compact Source Files

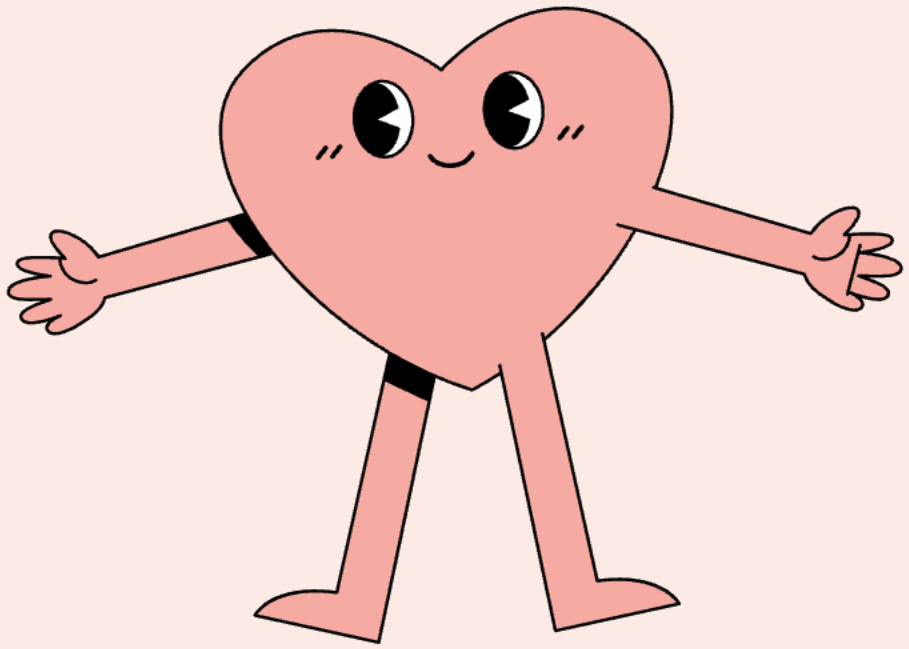
- An implicit class automatically created by the compiler. It will be a final top-level class in the unnamed package
- Extends java.lang.Object and implements no interfaces
- Has only a default no-argument constructor
- All fields and methods declared in the file become its members
- Must have a launchable main method → otherwise compile-time error

Using a Method

```
String greeting() { return "Hello, World!"; }

void main() {
    System.out.println(greeting());
}
```

Using a Field

```
String greeting = "Hello, World!";

void main() {
    System.out.println(greeting);
}
```



From Java 25

Compact Source Files and Instance Main Methods

eazy bytes

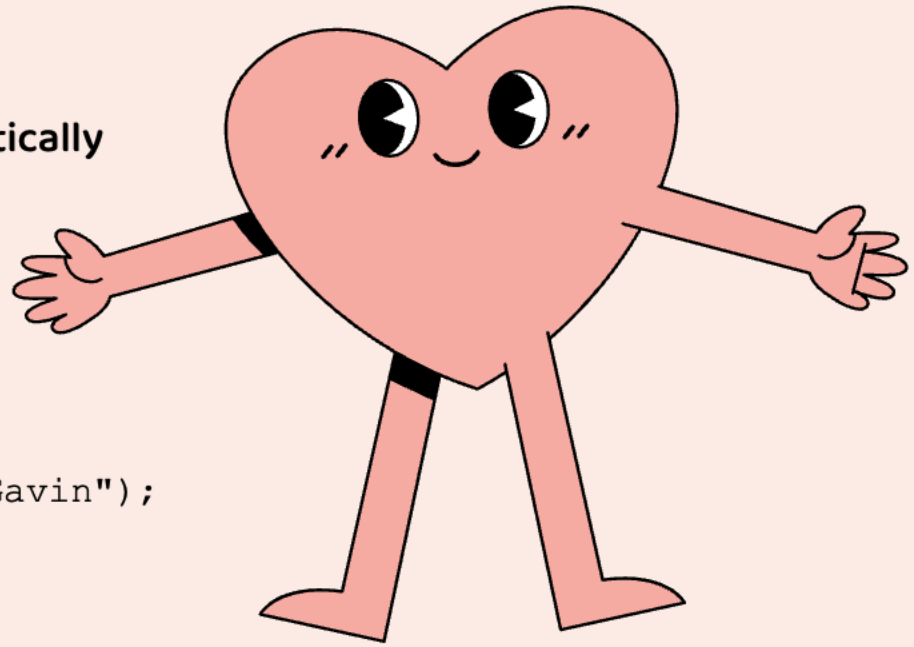
Automatic import of the java.base module

In traditional way, Imports must be written at the top. Beginners may find imports confusing and requires knowledge of package hierarchy

```
import java.util.List;
```

Simplification in Compact Source Files

All public top-level classes and interfaces from the java.base module are automatically available. Acts as if they were imported on demand. Common packages like:
`java.io, java.math, java.util` are immediately usable



```
// No need for: import java.util.List;

void main() {
    var authors = List.of("James", "Bill", "Guy", "Alex", "Dan", "Gavin");
    for (var name : authors) {
        IO.println(name + ": " + name.length());
    }
}
```

From Java 25

Flexible Constructor Bodies

eazy bytes

In Java, objects are built step by step, starting from the top of the class hierarchy. This means a superclass constructor always runs before a subclass constructor. To make this happen, every constructor must start by calling super(...) or this(...). If you don't write one, the compiler automatically adds a call to the no-argument superclass constructor (super()).

❖ **With Flexible Constructor Bodies,** Java is going to allow statements **before super() or this() invocations.**

Constructor Body:

```
{
        [ExplicitConstructorInvocation]
        [BlockStatements]
}
```

to:

```
{
        [BlockStatements]
        ExplicitConstructorInvocation
        [BlockStatements] }
}
```

an explicit constructor invocation is either super(..) or this(..)

From Java 25

Flexible Constructor Bodies

eazy bytes

❖ Problem 1

- Constructor chaining ensures safety but can feel too strict
- Developers often want to validate or compute arguments before super(...)
- Workarounds exist (helper methods), but they add extra complexity

```
class Vehicle {
    int speed;

    Vehicle(int speed) {
        if (speed < 0) throw new IllegalArgumentException("Invalid speed");
        this.speed = speed;
    }
}

class Car extends Vehicle {
    Car(int speed) {
        super(speed); // Forced first call
        if (speed > 200) {
            throw new IllegalArgumentException("Car too fast");
        }
    }
}
```


Here the superclass constructor runs before validation in Car
May perform unnecessary work before failing

From Java 25

Flexible Constructor Bodies

eazy bytes

❖ Problem 1

Workaround (Helper Method)

```
class Vehicle {
    int speed;

    Vehicle(int speed) {
        if (speed < 0) throw new IllegalArgumentException("Invalid speed");
        this.speed = speed;
    }
}

class Car extends Vehicle {
    private static int verifySpeed(int speed) {
        if (speed > 200) throw new IllegalArgumentException("Car too fast");
        return speed;
    }

    Car(int speed) {
        super(verifySpeed(speed));
    }
}
```


Validation happens before invoking Vehicle constructor
Still obeys Java's constructor chaining rules

From Java 25

Flexible Constructor Bodies

eazy bytes

❖ Problem 1

Solution with Flexible Constructor Bodies

```
class Vehicle {
    int speed;

    Vehicle(int speed) {
        this.speed = speed;
    }
}


public class Car extends Vehicle {

    Car(int speed) {
        if (speed < 0) throw new IllegalArgumentException("Invalid speed");
        if (speed > 200) throw new IllegalArgumentException("Car too fast");
        super(speed); // Super can be invoked after validations logic
    }

}
```

From Java 25

Flexible Constructor Bodies

eazy bytes

❖ Problem 2

Superclass Constructors Can Break Subclass Integrity

- Superclass constructor runs first (top-down initialization)
- Ensures superclass fields are initialized correctly
- BUT... it can accidentally access subclass fields before they're initialized

What happens?

```
new SavingsAccount("");
```

Throws NullPointerException because accountNumber is accessed before initialization.

```
class Account {
    public Account() {
        check();  // calls subclass's override
    }

    public void check() { }
}

class SavingsAccount extends Account {
    private String accountNumber;

    public SavingsAccount(String number) {
        super();              // Account() runs first
        this.accountNumber = number;
    }

    @Override
    public void check() {
        if (accountNumber.length() == 0) { // ✗ still null
            throw new RuntimeException("Invalid account");
        }
    }
}
```

Flexible Constructor Bodies

eazy bytes

❖ Problem 2

⚠️ Why This Violates Integrity

- Subclass (SavingsAccount) expects accountNumber to always be valid
- But superclass (Account) indirectly forces early access
- Even final fields can be seen in an uninitialized state
- This breaks the class's guarantee of internal consistency

From Java 25

Flexible Constructor Bodies

eazy bytes

❖ Problem 2

Solution with Flexible Constructor Bodies

```
class Account {

    public Account() {
        check();  // calls subclass's override
    }

    public void check() { }
}

public class SavingsAccount extends Account {

    private String accountNumber;

    public SavingsAccount(String number) {
        this.accountNumber = number;
        super();
    }

    @Override
    public void check() {
        if (accountNumber.length() == 0) {
            throw new RuntimeException("Invalid account number");
        }
    }
}
```

Flexible Constructor Bodies

eazy bytes

❖ Early Construction Context

This feature revises the previous restrictions by allowing code to run before calling super() or this(). It moves away from treating constructor arguments as static, and introduces the early construction context. With this approach, the constructor is divided into three parts:

1. **Prologue:** Code that runs before super() or this().
2. **Call of super() or this():** The required first statement in the constructor.
3. **Epilogue:** Code that follows the super() or this() call.

There are certain limitations on what we can write before the ExplicitConstructorInvocation.

Restrictions

❌ Cannot use this (explicitly or implicitly)
❌ Cannot access fields/methods of the current instance
❌ Cannot access fields/methods of the superclass (super)
✅ Allowed: assigning values to uninitialized fields declared in the class

From Java 25

Flexible Constructor Bodies

eazy bytes

❖ **Example 1 (Invalid Use)**

```
class Book extends Page {
    String title;
    int pages = 100;

    Book() {
        System.out.println(this);      // ❌ Not allowed
        var n = title;                 // ❌ Implicit this
        getPageCount();                // ❌ Implicit this

        title = "Java 25";             // ✅ OK (field had no initializer)
        pages = 200;                   // ❌ Error (already initialized)

        super();
}

    int getPageCount() { return pages; }
}
```

From Java 25

Flexible Constructor Bodies

eazy bytes

❖ **Example 2 (Superclass Access Not Allowed)**

```
class Shape {
    int sides;
    void draw() {}
}

class Triangle extends Shape {
    Triangle() {
        var x = super.sides;    // ✗ Not allowed
        super.draw();           // ✗ Not allowed
        super();
    }
}
```

From Java 25

Flexible Constructor Bodies

eazy bytes

❖ Executing statements before this()

You can run statements before this(...), as long as they don't access other instance members. Useful for argument validation and pre-processing.

```
public class Course {
    String courseName;
    int duration;

    // Main constructor
    public Course(String name, int duration) {
        this.courseName = Objects.requireNonNull(name);
        this.duration = duration;
    }

    // Overloaded constructor
    public Course(String prefix, String name, int duration) {
        Objects.requireNonNull(prefix);      // ✅ Allowed before this()
        Objects.requireNonNull(name);
        if (duration <= 0) throw new IllegalArgumentException("Invalid duration");

        this(prefix + " " + name, duration); // delegates safely
    }
}
```

From Java 25

Scoped Values

eazy bytes

❖ What Are Scoped Values?

A way to share immutable data between:

- A method and its callees (same thread) with out explicitly passing
- Parent and child threads
- Alternative to ThreadLocal → easier to reason about, faster, less memory

❖ The Problem - Parameter Passing Hell

Traditional Approach: Passing Context Everywhere

Problem: Every method needs the context parameter, even if it doesn't use it directly!


```
public class OrderService {
    public void processOrder(Order order, UserContext context) {
        validateOrder(order, context);
        calculatePrice(order, context);
        saveOrder(order, context);
    }

    private void validateOrder(Order order, UserContext context) {
        // Need context for user permissions
        checkUserPermissions(context);
    }

    private void calculatePrice(Order order, UserContext context) {
        // Need context for user discount
        applyUserDiscount(order, context);
    }
}
```

From Java 25

Scoped Values

eazy bytes

❖ The solution using ThreadLocal

Benefits: No parameter passing needed
Problems: Easy to forget cleanup, mutable, memory leaks

```
public class OrderService {

    private static final ThreadLocal<UserContext> CONTEXT = new ThreadLocal<>();

    public void processOrder(Order order, UserContext context) {
        CONTEXT.set(context);  // Set context
        try {
            validateOrder(order);
            calculatePrice(order);
            saveOrder(order);
        } finally {
            CONTEXT.remove();  // Clean up (often forgotten!)
        }
    }

    private void validateOrder(Order order) {
        UserContext context = CONTEXT.get();  // Get context
        checkUserPermissions(context);
    }
}
```

From Java 25

Scoped Values

eazy bytes

❖ ThreadLocal Problems

Three Major Issues with ThreadLocal:

Unconstrained Mutability

```
// Any code can change the value anytime
  CONTEXT.set(newValue);  // Dangerous!
```

Unbounded Lifetime

```
// Forgot to call remove() - memory leak!
CONTEXT.set(context);
// ... code ...
// CONTEXT.remove(); // Oops, forgot this!
```

Expensive Inheritance

```
// Child threads copy ALL ThreadLocal variables
// High memory overhead with many virtual threads
```


From Java 25

Scoped Values

eazy bytes

❖ ScopedValue - The Modern Solution

```
public class OrderService {

    private static final ScopedValue<UserContext> CONTEXT =
        ScopedValue.newInstance();

    public void processOrder(Order order, UserContext context) {
        ScopedValue.where(CONTEXT, context)
                   .run(() -> {
                       validateOrder(order);
                       calculatePrice(order);
                       saveOrder(order);
                   });
        // Context automatically destroyed here
    }

    private void validateOrder(Order order) {
        UserContext context = CONTEXT.get();  // Read context
        checkUserPermissions(context);
    }

}
```

From Java 25

Scoped Values

eazy bytes

❖ Simple ScopedValue Example

```
public class SimpleExample {

    private static final ScopedValue<String> USER_NAME =
        ScopedValue.newInstance();

    public void doWork() {
        ScopedValue.where(USER_NAME, "Alice")
                   .run(() -> {
                       processTask();  // Can access USER_NAME
                   });
        // USER_NAME is no longer accessible here
    }

    private void processTask() {
        String user = USER_NAME.get();  // Returns "Alice"
        System.out.println("Processing for: " + user);
        callAnotherMethod();
    }

    private void callAnotherMethod() {
        String user = USER_NAME.get();  // Still returns "Alice"
        System.out.println("Still processing for: " + user);
    }
}
```

From Java 25

Scoped Values

eazy bytes

❖ Dynamic Scope Concept

Dynamic Scope: Value is accessible during the execution lifetime of the run() method and all methods called from it.

```
public void methodA() {

    ScopedValue.where(NAME, "John").run(() -> {
        methodB();  // Can access NAME
    });
    // NAME not accessible here
}

public void methodB() {
    String name = NAME.get();  // Returns "John"
    methodC();  // NAME still accessible
}

public void methodC() {
    String name = NAME.get();  // Still returns "John"
}
```

From Java 25

Scoped Values

eazy bytes

❖ Rebinding (Nested Scopes)

Dynamic Scope: Value is accessible during the execution lifetime of the run() method and all methods called from it.

```
private static final ScopedValue<String> LEVEL = ScopedValue.newInstance();

public void outerMethod() {
    ScopedValue.where(LEVEL, "Level 1").run(() -> {
        System.out.println(LEVEL.get());  // Prints "Level 1"
        innerMethod();
        System.out.println(LEVEL.get());  // Still prints "Level 1"
    });
}

public void innerMethod() {
    System.out.println(LEVEL.get());  // Prints "Level 1"

    ScopedValue.where(LEVEL, "Level 2").run(() -> {
        System.out.println(LEVEL.get());  // Prints "Level 2"
        deepMethod();
    });

    System.out.println(LEVEL.get());  // Back to "Level 1"
}

public void deepMethod() {
    System.out.println(LEVEL.get());  // Prints "Level 2"
}
```

Performance improvements using AOT

eazy bytes

❖ What is Ahead-of-Time (AOT) in Java

Traditional Java Startup (Just-In-Time)

When you run: java MyApp

1. JVM starts
2. Reads JAR files from disk
3. Parses class files
4. Loads classes into memory
5. Links classes together (verifies, resolves references)
6. Runs static initializers
7. Finally starts your application

This happens EVERY TIME you start the app!

Ahead-of-Time (AOT) Approach: Do the work ONCE, reuse it MANY times

Training Run:

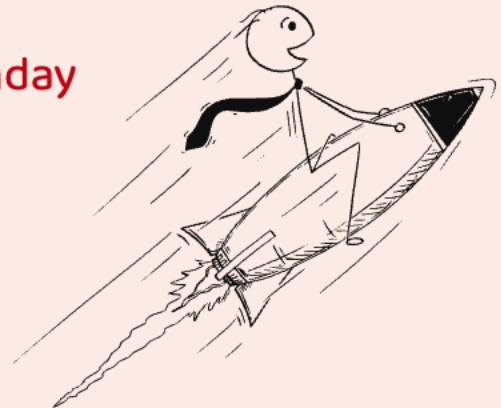
1. Run app once and record what classes are needed
2. Pre-load and pre-link all those classes
3. Save them in a cache file

Production Runs:

1. JVM starts
2. Loads pre-processed classes from cache
3. App starts immediately!

Result: Much faster startup!

Think of it like meal prep: Instead of cooking every day (JIT), you cook once on Sunday and reheat all week (AOT)!

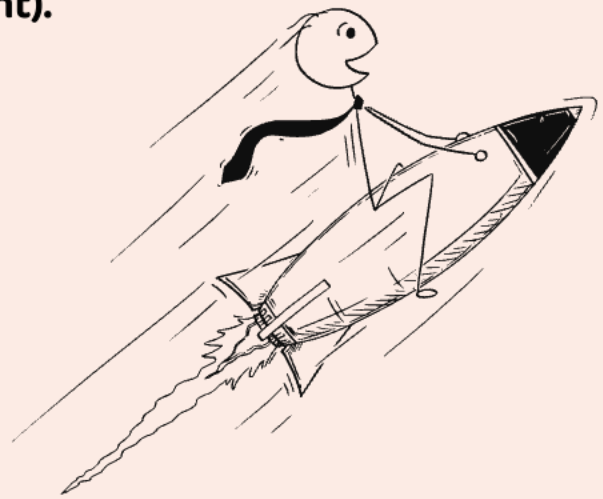


From Java 24

AOT Class Loading & Linking

eazy bytes

- **Instead of doing work just-in-time (during startup), do it ahead-of-time (before deployment).**



Three-Step Process (JDK 24):

Step 1: Training Run (Record)

Run your app once to record what classes it uses

```
java -XX:AOTMode=record -XX:AOTConfiguration=app.aotconf \
     -cp app.jar com.example.App
```

JVM watches your app run -> Records all classes loaded -> Saves configuration to app.aotconf

Step 2: Create Cache

Build the AOT cache from recorded configuration

```
java -XX:AOTMode=create -XX:AOTConfiguration=app.aotconf \
     -XX:AOTCache=app.aot -cp app.jar
```

Processes all classes ahead-of-time -> Saves pre-loaded/pre-linked classes to app.aot

From Java 24

AOT Class Loading & Linking

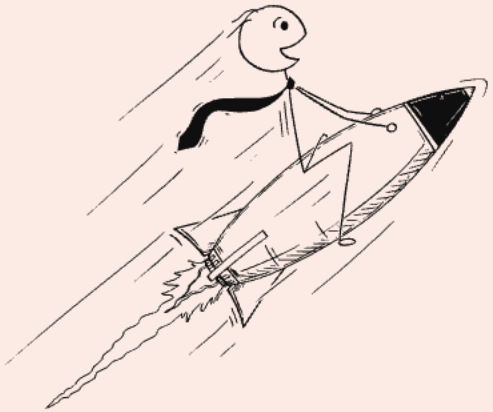
eazy bytes

Step 3: Production Use

Run with AOT cache for fast startup

```
java -XX:AOTCache=app.aot -cp app.jar com.example.App
```

Classes loaded instantly from cache -> No parsing, loading, or linking needed



From Java 25

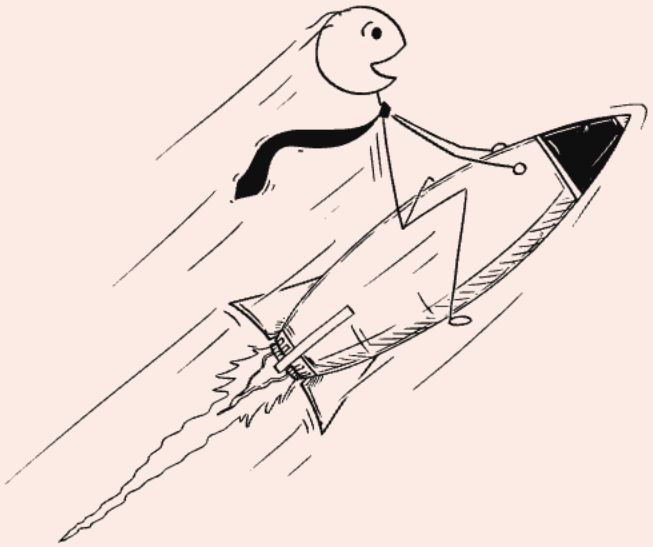
Ahead-of-Time Command-Line Ergonomics

eazy bytes

- **The Problem with JDK 24 is in total two separate commands need to run to create the AOT cache**

JDK 25 Solution: One-Step Cache Creation

```
# Single command does training + cache creation
java -XX:AOTCacheOutput=app.aot -cp app.jar com.example.App

# Then use as before
java -XX:AOTCache=app.aot -cp app.jar com.example.App
```

Benefits:

Simpler: One command instead of two
Cleaner: No leftover configuration files
Efficient: Perfect for automation and CI/CD

From Java 25

Ahead-of-Time Command-Line Ergonomics

eazy bytes

What Goes Into an AOT Cache?

Classes That CAN be Cached

- ✓ JDK Classes: String, ArrayList, HashMap, etc.
- ✓ Application Classes: Your main code from JARs
- ✓ Library Classes: Spring, Hibernate, etc. (from JARs)
- ✓ Classes loaded by built-in classloaders

Classes That CANNOT be Cached (Currently)

- User-defined classloader classes: Custom classloading logic
- Signed classes: Security restrictions
- Old bytecode: Requires old verifier
- Dynamic classes: Generated at runtime

What's Pre-computed?

- Parsed bytecode: Ready-to-use class structures
- Resolved references: Links between classes established
- Verified code: Bytecode verification completed
- Loaded metadata: Class, method, field information ready

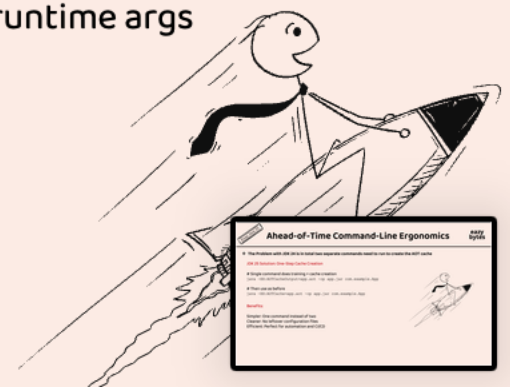
For AOT cache to work, training and production must match:

Must be Identical

- ✓ JDK Version: Same release number
- ✓ Architecture: x64, aarch64, etc.
- ✓ Operating System: Linux, Windows, macOS
- ✓ Main Classpath: Same JARs in same order
- ✓ Module Configuration: Same module options

Can be Different

- Garbage Collector: G1, Parallel, etc.
- Main Class: Training vs. production entry points
- Extra Classpath: Can append additional JARs
- Heap Size: -Xmx can vary
- Application Arguments: Different runtime args



From Java 25

Using AOT in Spring Boot App

eazy bytes

1. Build your Spring Boot application

```
mvn clean package
```

2. Create AOT cache (JDK 25 style - one command)

```
java -XX:AOTCacheOutput=myapp.aot -jar target/myapp.jar
```

3. Deploy and run with AOT cache

```
java -XX:AOTCache=myapp.aot -jar target/myapp.jar
```