# JDK 22 FEATURES TO GET EXCITED ABOUT

Daniel Hinojosa

# STREAM GATHERERS

# STREAM GATHERERS

- Enhancement of Java Streams that performs collections as an *intermediate operation*
- At this time if you wish to aggregate or manage state, you would need to run a `collect`, and then call `.stream()` to perform extra work

Source: JEP 461 Stream Gatherers

# MANY REQUESTS FOR INTERMEDIATE OPERATIONS

- Over the years, many new intermediate operations have been suggested for the Stream API.
- Most of them make sense when considered in isolation, but adding all of them would make the (already large) Stream API more difficult to learn because its operations would be less discoverable.

Source: JEP 461 Stream Gatherers

# `StreamGatherer`

- `Stream::gather(Gatherer)` is a new intermediate stream operation that processes the elements of a stream by applying a user-defined entity called a gatherer.
- With the gather operation we can build efficient, parallel-ready streams that implement almost any intermediate operation.
- `Stream::gather(Gatherer)` is to intermediate operations what `Stream::collect(Collector)` is to terminal operations.

Source: JEP 461 Stream Gatherers

# TYPES OF GATHERERS

- Gatherers can transform elements in a one-to-one, one-to-many, many-to-one, or many-to-many fashion.
- They can track previously seen elements in order to influence the transformation of later elements
- They can short-circuit in order to transform infinite streams to finite ones, and they can enable parallel execution.

Source: JEP 461 Stream Gatherers

# FOUR FUNCTIONS THAT BECOME A `Gatherer`

1. The `initializer` - provides an object that maintains private state while processing stream elements
2. The `integrator` - integrates a new element from the input stream, possibly inspecting the private state object and possibly emitting elements to the output stream
3. The `combiner` - used to evaluate the gatherer in parallel when the input stream is marked as parallel. If a gatherer is not parallel-capable then it can still be part of a parallel stream pipeline, but it is evaluated sequentially
4. The `finisher` - invoked when there are no more input elements to consume. This function can inspect the private state object and, possibly, emit additional output elements.

Source: JEP 461 Stream Gatherers

# THE STEPS OF A `Gatherer`

1. Create a `Downstream` object which, when given an element of the gatherer's output type, passes it to the next stage in the pipeline.
2. Obtain the gatherer's private state object by invoking the get() method of its `initializer`.
3. Obtain the gatherer's `integrator` by invoking its `integrator()` method.
4. While there are more input elements, invoke the integrator's `integrate(…)` method, passing it the state object, the next element, and the downstream object. Terminate if that method returns `false`.
5. Obtain the gatherer's finisher and invoke it with the state and downstream objects.

Source: JEP 461 Stream Gatherers

# `Gatherers` ARE MEANT TO BE COMPOSED

- Gatherers support composition via the `andThen(Gatherer)` method, which joins two gatherers where the first produces elements that the second can consume
- This enables the creation of sophisticated gatherers by composing simpler ones:

```
source.gather(a).gather(b).gather(c).collect(...)
```

Can be applied as such:

```
source.gather(a.andThen(b).andThen(c)).collect(...)
```

# DEMO: VIEWING A GATHERER

In the *jdk22-features-excited/src/main/java* directory, navigate to the `com.evolutionnext.streamgatherers` package and run and analyze the following in order: `SimpleGatherer`, `GroupByGatherer`, and `ZipperGatherer`

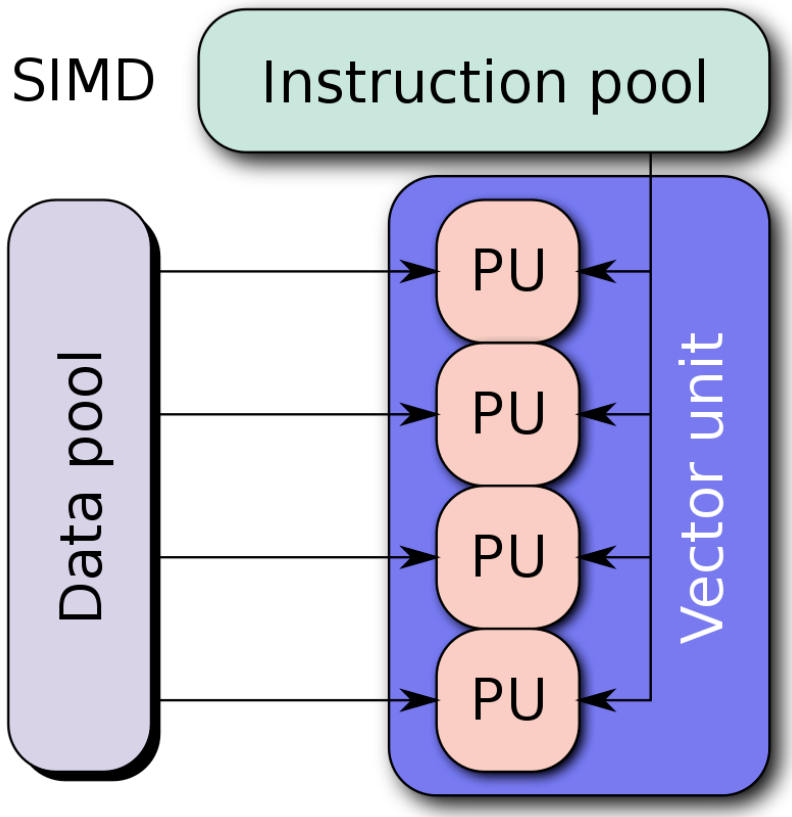
# VECTOR API

# SIMD

- Single Instruction/Multiple Data
- Such machines exploit data level parallelism, but not concurrency
- Not all machines offer the ability to process SIMD processes
- This is not new, and have been used with supercomputers in the 60s and 70s

Source: Wikipedia Single Instruction/Multiple Data

# SIMD PROCESSES




Source: Wikipedia Single Instruction/Multiple Data

# WHAT MODERN MACHINERY HAS SIMD?

- All x86 processors
- ARM processor with Neon Instruction Set
- Other processors are available, although it is not a goal to support vector instructions on CPU architectures other than x86 and AArch64

Source: Wikipedia Single Instruction/Multiple Data

# USE CASES

- Fast Computation of Arrays
- Machine Learning and Ai
- Gaming Arithmetic
- Imaging and Audio
- Cryptography

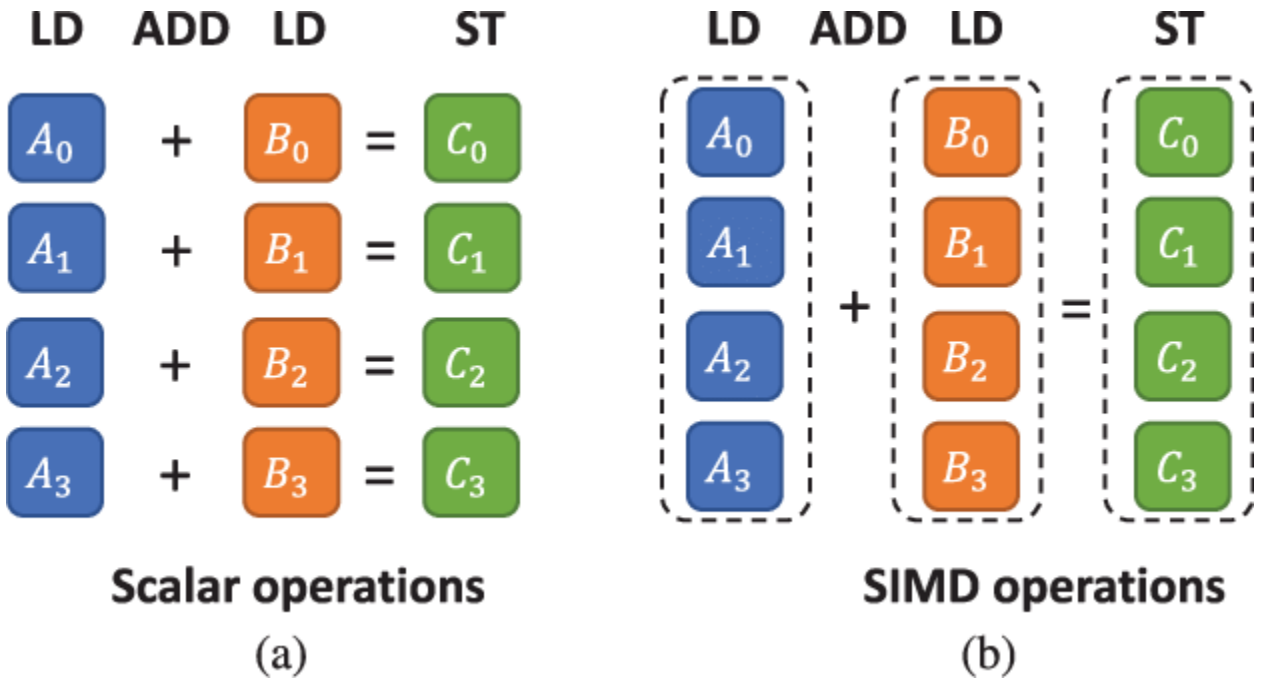
# CONCURRENT PROCESSING NOW WITH JAVA

- Distributed Process: Break up data into small chunks and calculate
- Carried through with Threads

# BENEFITS OF VECTOR API

- Fast *Parallel* Computation not Concurrent
- SIMD *do not use threads*
- Uses large amount of operating units, executing the same operation *at the same CPU cycle* at the exact same time
- This is tethered to Valhalla, particularly to the use of value objects
  - See more on value objects in this JEP
- This also requires the C2 Compiler to perform
  - Note: the C2 Compiler is the optimized "server-compiler"

# IMAGE OF SIMD VS SCALAR




Development of Computationally efficient voice conversion system on mobile phones

# WHAT DO WE HAVE TO DO?

- Structure your data as an array of numbers
- If you have 16 computation units, you can handle vectors of 16 components, and you will go sixteen times faster
- Create degradation logic if the Vector API is unable to perform with classic loop

# SPECIES

- Know the types that we need to use: `byte`, `short`, `int`, `long`, `float`, and `double`
- Know the size of what can be handled, anywhere from 64 bits to 512 bits, *for now*
- Depending on the type that used is called the *species*

# VECTORS

- A vector computation consists of a sequence of operations on vectors
- A vector comprises a (usually) fixed sequence of scalar values, where the scalar values correspond to the number of hardware-defined vector lanes
- A binary operation applied to two vectors with the same number of lanes would, for each lane, apply the equivalent scalar operation on the corresponding two scalar values from each vector.

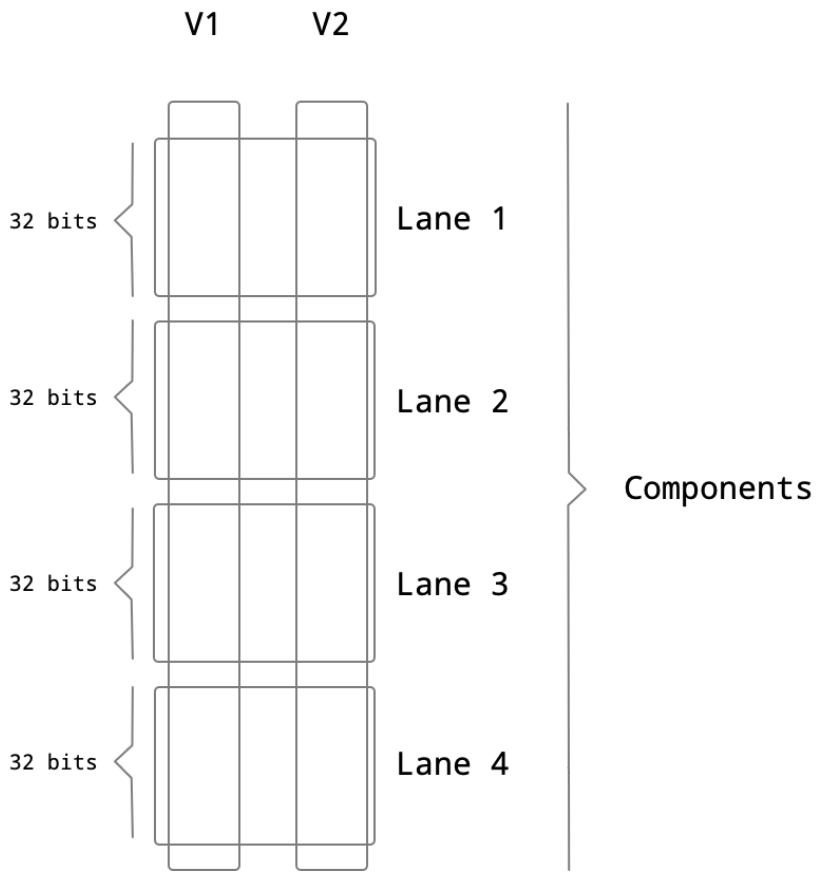
# HOW TO DETERMINE WHAT WE CAN PERFORM?

- We want: 256 bits and we wish to store a vector of integers
- We know: Integers are 32 bits (4 x 8 byte)
- We can calculate 256 bits / 32 bits is 8 components
- *Each component is called a lane*

# SHAPE

- The shape of the vector is the bit-wise size or the number of bits of a vector
- A vector with a shape of 512 bits will have 16 lanes and can operate on 16 `int`s at a time, while a 64-bit one will have only 2.

# VECTORS AND LANES



# `Vector<E>`

- The Vector<E> class has six abstract subclasses for each of the six supporting types: `ByteVector`, `ShortVector`, `IntVector`, `LongVector`, `FloatVector`, and `DoubleVector`.
- Specific implementations are important with SIMD machines, which is why shape-specific subclasses further extend these classes for each type.
- `Vector<E>` is mapped to a hardware vector register when vector computations are compiled by the HotSpot C2 compiler.
- For example `Int128Vector`, `Int512Vector`, etc. These will be abstracted.

Source: JEP 469

# DEMO: VECTOR API

In the *jdk22-features-excited/src/main/java* directory, navigate to the `com.evolutionnext.vector` package and run `AddStandardVector`

# MASKS

- To support control flow, some vector operations optionally accept masks represented by the public abstract class `VectorMask<E>`
- Each element in a mask is a boolean value corresponding to a vector lane.
- A mask selects the lanes to which an operation is applied: It is applied if the mask element for the lane is `true`, and some alternative action is taken if the mask is `false`.
- In other words, it's like `filter`

Source: JEP 469

# APPLYING MASKS

- Similar to vectors, instances of `VectorMask<E>` are instances of non-public concrete subclasses defined for each element type and length combination.
- The instance of `VectorMask<E>` used in an operation should have the same type and length as the vector instances involved in the operation.
- Vector comparison operations produce masks, which can then be used as input to other operations to selectively operate on certain lanes and thereby emulate flow control.
- Masks can also be created using static factory methods in the `VectorMask<E>` class.

Source: JEP 469

# EFFICIENT APPLICATION OF `Mask<T>`

- On such platforms an instance of `VectorMask<E>` is mapped to a predicate register, and a mask-accepting operation is compiled to a predicate-register-accepting vector instruction.
- On platforms that don’t support predicate registers, a less efficient approach is applied: An instance of VectorMask<E>is mapped, where possible, to a compatible vector register, and in general a mask-accepting operation is composed of the equivalent unmasked operation and a blend operation.

Source: JEP 469

# DEMO: VECTOR MASKS

In the *jdk22-features-excited/src/main/java* directory, navigate to the `com.evolutionnext.vector` package and run and analyze `AddStandardVectorLoop`, then move to `AddStandardVectorLoopWithMask`, and finally `AddStandardVectorLoopWithoutMask`

# VECTOR OPERATIONS

**There are two types of vector operations**

- Lane-wise operations
- Cross-lane operations

# LANE WISE OPERATIONS

- Performs a scalar operation, like addition, on a single lane on one or more vectors at a time.
- These operations can combine one lane of a vector with a lane of a second vector, for instance, during an add operation.
- Operations are further classified as unary, binary, ternary, test, or conversion operations

Source: JEP 469

# CROSS-LANE OPERATIONS

- Compute or modify data from different lanes of an entire vector.
- Sorting the components of a vector is an example of a cross-lane operation.
- Cross-lane operations can produce scalars or vectors of different shapes from the source vectors.
- Cross-lane operations can be further classified into permutation and reduction operations.

Source: JEP 469

# CROSS LANE PERMUTATION

- To support cross-lane permutation operations, some vector operations accept shuffles represented by the public abstract class `VectorShuffle<E>`.
- Each element in a shuffle is an int value corresponding to a lane index.
- A shuffle is a mapping of lane indexes, describing the movement of lane elements from a given vector to a result vector.
- Similar to vectors and masks, instances of `VectorShuffle<E>` are instances of non-public concrete subclasses defined for each element type and length combination.
- The instance of `VectorShuffle<E>` used in an operation should have the same type and length as the vector instances involved in the operation.

Source: JEP 469

# DEMO: VECTOR SHUFFLE

In the *jdk22-features-excited/src/main/java* directory, navigate to the `com.evolutionnext.vector` package and run and analyze `CompareVector`, and then `ReduceVector`

# STATEMENTS BEFORE super

# HOW WE DEAL WITH `super` NOW?

- Prior to this JEP, when calling a super `class`, Java *must* call the super class before we test invariants
- Even if we test the invariant and don't want to go through the construction, we still have to initialize the super class, thus causing an expense.
- The initial values of fields declared in the subclass can depend upon the initial values of fields declared in the superclass, so it is *critical to initialize fields of the superclass first*, before fields of the subclass.

# WHAT IS THIS NEW FEATURE?

- Now called Flexible Constructor Bodies (Second Preview)
- In constructors in the Java programming language, we can now allow statements that do not reference the instance being created to appear before an explicit constructor invocation, like `this` and `super`

Source: Flexible Constructor Bodies (Second Preview)

# STATEMENTS BEFORE `super`

**Now with this latest feature, we can use the `super` call further down in a constructor given the following rules**

1. Any unqualified `this` expression is disallowed in a pre-construction context
2. Any field access, method invocation, or method reference qualified by `super` is disallowed in a pre-construction context
3. No implicit calls to `this` or `super` keyword
4. You can access symbols of an outer class from within an inner class

Source: Flexible Constructor Bodies (Second Preview)

# DEMO: TRY STATEMENTS BEFORE `super`

In the *jdk22-features-excited/src/main/java* directory, and in the `com.evolutionnext.statementsbeforesuper` package let's try to refactor an application with validation before `super()`

# IMPLICIT DECLARED CLASSES AND ENHANCED `main` METHODS

# IMPLICIT DECLARED CLASSES AND ENHANCED `main` METHODS

- Evolve the Java programming language so that students can write their first programs without needing to understand language features designed for large programs.
- Far from using a separate dialect of the language, students can write streamlined declarations for single-class programs and then seamlessly expand their programs to use more advanced features as their skills grow.

JEP 463: Implicitly Declared Classes and Instance Main Methods

# DEMO: LEARN JAVA AGAIN, DIFFERENTLY

In the *jdk22-features-excited/implicit-main* directory, and let’s perform a "Hello World" application

# DECLUTTERING THE BEGINNER CODE

1. The class declaration and the mandatory `public` access modifier are programming-in-the-large constructs. They are useful when encapsulating a code unit with a well-defined interface to external components, but pointless in this little example.

2. The `String[] args` parameter also exists to interface the code with an external component, in this case the operating system's shell. It is mysterious and unhelpful here, especially since it is not used in simple programs like `HelloWorld`.

3. The `static` modifier is part of the language's class-and-object model. For the novice, static is not just mysterious but harmful: To add more methods or fields that main can call and use the student must either declare them all as static — thereby propagating an idiom which is neither common nor a good habit — or else confront the difference between static and instance members and learn how to instantiate an object.

# RUNNING THE CODE

**To try the examples below in JDK 22 you must enable preview features as follows**

- Compile the program with `javac --release 22 --enable-preview Main.java` and run it with `java --enable-preview Main`; or,
- When using the source code launcher from JEP 330 introduced in JDK 11, run the program with `java --source 22 --enable-preview Main.java`

# FOREIGN FUNCTION MEMORY API

# FOREIGN FUNCTION MEMORY API

- The Foreign Function and Memory (FFM) API enables Java programs to interoperate with code and data outside the Java runtime.
- This API enables Java programs to call native libraries and process native data without the brittleness and danger of JNI.
- The API invokes foreign functions, code outside the JVM, and safely accesses foreign memory, memory not managed by the JVM.

Source: Oracle Foreign Function and Memory API Documentation

# FOREIGN FUNCTION MEMORY API FEATURES

- Memory Segment and Arenas
- Calling C Libraries
- Pointer Handling
- Memory Layouts and Structured Access
- Checking Native Errors
- Slicing Allocators and Slicing Memory
- Restricted Methods
- `jextract`

Source: Oracle Foreign Function and Memory API Documentation

# MEMORY SEGMENTS

- Heap segment: This is a memory segment backed by a region of memory inside the Java heap, an on-heap region.
- Native segment: This is a memory segment backed by a region of memory outside the Java heap, an off-heap region. The examples in this chapter demonstrate how to allocate and access native segments.

Source: Oracle Foreign Function and Memory API Documentation

# ARENAS

- Arenas control the lifecycle of native memory segments.
- Each arena has a scope, which specifies when the region of memory that backs the memory segment will be deallocated and is no longer valid.
- A memory segment can only be accessed if the scope associated with it is still valid or alive.

Source: Oracle Foreign Function and Memory API Documentation

# ARENAS

## There are four kinds of arenas

- A confined arena
- A shared arena
- An automatic arena
- A global arena

Source: Oracle Foreign Function and Memory API Documentation

# CONFINED ARENA

- Created with `Arena::ofConfined`
- A confined arena provides a bounded and deterministic lifetime
- Its scope is alive from when it's created to when it's closed.
- A confined arena has an owner thread, and is typically the thread that created it.
- **Only the owner thread can access the memory segments allocated in a confined arena.**
- You'll get an exception if you try to close a confined arena with a thread other than the owner thread.

Source: Oracle Foreign Function and Memory API Documentation

# SHARED ARENA

- Created with `Arena::ofShared`
- It has no owner thread.
- Multiple threads may access the memory segments allocated in a shared arena.
- Any thread may close a shared arena, and the closure is guaranteed to be safe and atomic.

Source: Oracle Foreign Function and Memory API Documentation

# AUTOMATIC ARENA

- Created with `Arena::ofAuto`.
- This is an area that's managed, automatically, by the garbage collector. Any thread can access memory segments allocated by an automatic arena.
- If you call `Arena::close` on an automatic arena, you'll get a `UnsupportedOperationException`.

Source: Oracle Foreign Function and Memory API Documentation

# GLOBAL ARENA

- Created with `Arena::global`.
- Any thread can access memory segments allocated with this arena.
- In addition, the region of memory of these memory segments is never deallocated
- If you call `Arena::close` on a global arena, you'll get a `UnsupportedOperationException`.

Source: Oracle Foreign Function and Memory API Documentation

# ARENA SUMMARY

| *Type* | Lifetime | Thread Access |
|---|---|---|
| *Global* | Unbounded Lifetime | Multithreaded Access |
| *Automatic* | Automatic Lifetime | Multithreaded Access |
| *Confined* | Explicit Bounded Lifetime | Single Threaded Access |
| *Shared* | Explicit Bounded Lifetime | Multithreaded Access |

# DEMO: SETTING UP AN `Arena`

In the *jdk22-features-excited/src/main/java* directory, navigate to the `com.evolutionnext.foreignfunctionmemory` package and run and analyze `InvokeStrLen` and see how an `Arena` of `Confined` is set up.

# `SegmentAllocator` INTERFACE

- The `Arena` interface extends the SegmentAllocator interface, which contains methods that both allocate off-heap memory and copy Java data into it.

# STORING AN OBJECT IN NATIVE MEMORY USING A CONFINED `Arena`

- `SegmentAllocator.allocateFrom(String)` or `SegmentAllocator.allocateFrom(String, Charset)` allocates a memory segment with an arena, converts a string into a UTF-8 encoded, null-terminated C string, and then stores the string into the memory segment

# OTHER ALLOCATIONS ARE AVAILABLE

- Primitives (`byte`, `short`, `int`, `long`, `float`, `double`)
- Strings with `Charset`
- Address to Object
- ValueLayout

# PRINTING THE CONTENT OF THE OFF-HEAP MEMORY

- The `MemorySegment` interface contains various access methods that enable you to read from or write to memory segments.
- Each access method takes as an argument a value layout, which models the memory layout associated with values of basic data types such as primitives.
- A value layout encodes the size, the endianness or byte order, the bit alignment of the piece of memory to be accessed, and the Java type to be used for the access operation.

# RETRIEVING OFF-HEAP MEMORY

**`MemoryLayout.get(ValueLayout.OfByte,address)` takes as an argument `ValueLayout.JAVA_BYTE`. This value layout has the following characteristics**

- The same size as a Java byte
- `Byte` alignment set to 1: This means that the memory layout is stored at a memory address that's a multiple of 8 bits.
- Byte order set to `ByteOrder.nativeOrder()`: A system can order the bytes of a multibyte value from most significant to least significant (big-endian) or from least significant to most significant (little-endian).

# LITTLE ENDIAN - BIG ENDIAN

*In one of the more obscure satirical analogies in English literature, Jonathan Swift wrote about the ideological tussle between two factions of Lilliputians in Gulliver's Travels (1726). The Big-Endians liked to break their eggs at the big end, while the Little-Endians preferred the pointier option. Chaos ensued.*

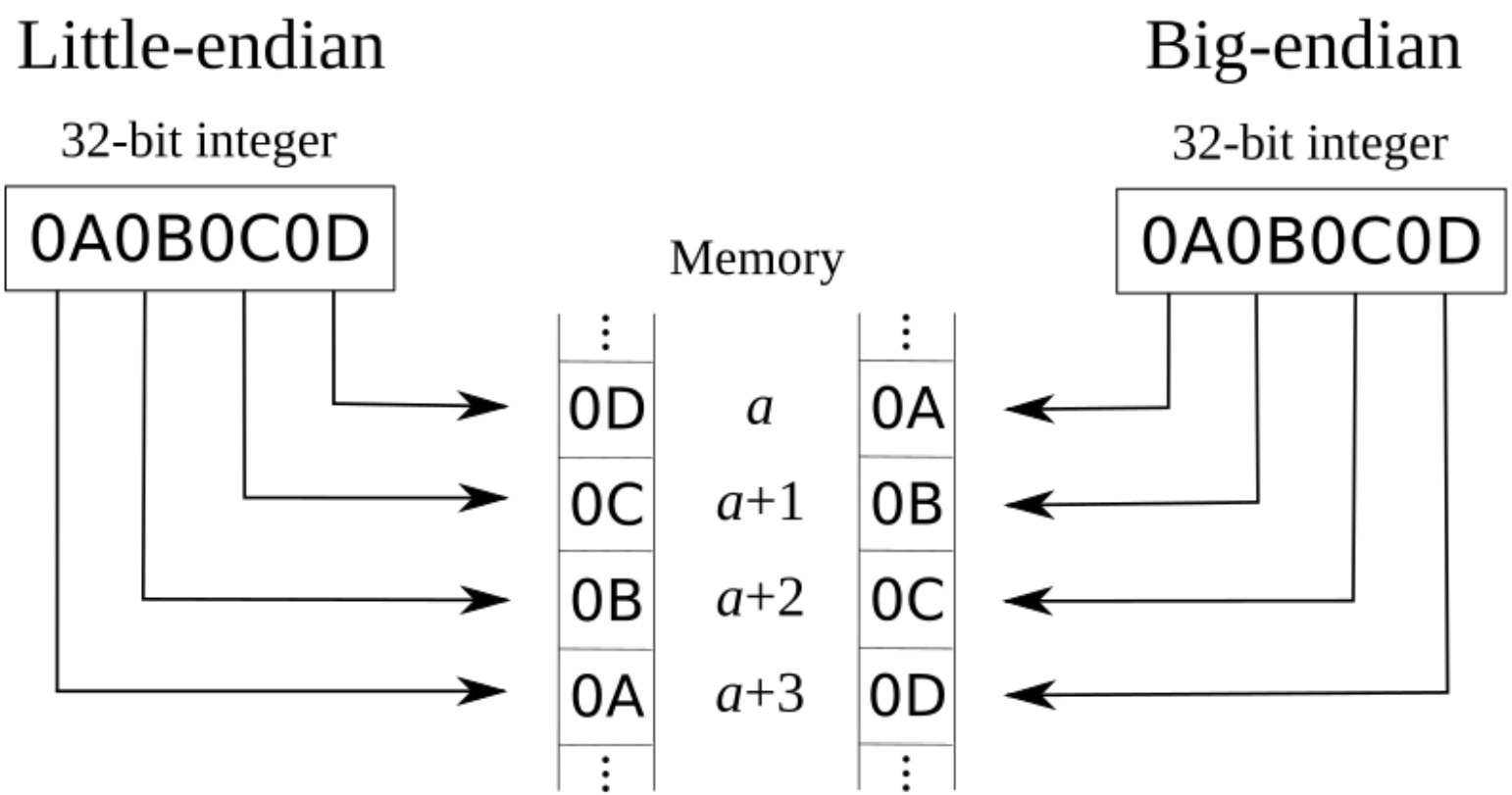
Matt Hall - Little Endian is Legal

# LITTLE ENDIAN - BIG ENDIAN

- Two hundred and fifty years later, Danny Cohen borrowed the terminology in his 1 April 1980 paper, *On Holy Wars and a Plea for Peace*
- He positioned the Big-Endians, preferring to store the big bytes first in memory, against the Little-Endians, who naturally prefer to store the little ones first.
- x86, ARM processors are little endian, RISC are big endian

Matt Hall - Little Endian is Legal

# LITTLE ENDIAN - BIG ENDIAN




Matt Hall - Little Endian is Legal

# DEMO: MEMORY ALLOCATION AND RETRIEVAL

In the *jdk22-features-excited/src/main/java* directory, navigate to the `com.evolutionnext.foreignfunctionmemory` package and run and analyze `InvokeStrLen` and see how we can set up and query memory from the native space

# CLOSING THE ARENA

- Arenas are typically closed with `try-with-resources` blocks
- All memory segments associated with its scope are invalidated, and the memory regions backing them are deallocated
- If you try to access a memory segment associated with an arena scope that's closed, you'll get an `IllegalStateException`

# DEMO: CLOSING THE ARENA

In the *jdk22-features-excited/src/main/java* directory, navigate to the `com.evolutionnext.foreignfunctionmemory` package and run and analyze `InvokeStrLen` and we can view how we close the `Arena`

# CALLING C LIBRARIES

**Calling a C library or anything that was compiled to a C library requires**

1. Obtaining an Instance to the Native Linker
2. Locating the Address of the C Function
3. Describing the C Function Signature
4. Creating a Downcall Handle to the C Function
5. Calling the C Function Directly from Java

# OBTAINING AN INSTANCE TO THE NATIVE LINKER

- Native Linker provides access to the libraries that adhere to the calling conventions of the platform in which the Java runtime is running.
- These libraries are referred to as "native" libraries.

# LOCATING THE ADDRESS OF THE C FUNCTION

- To call a native method, you need a *downcall* method handle, which is a `MethodHandle` instance that points to a native function.
- This instance requires the native function's address
- To obtain this address, you use a *symbol lookup*, which enables you to retrieve the address of a symbol in one or more libraries.

# LOCATING THE ADDRESS OF A LIBRARY C FUNCTION FROM A LIBRARY

- This method loads the specified library and associates it with an arena, which controls the symbol lookup's lifetime
- The following example specifies *libc.so.6*, which is the name of the C standard library for many Linux systems.

```
SymbolLookup stdLib = SymbolLookup.libraryLookup("libc.so.6", arena);
MemorySegment strlen_addr = stdLib.find("strlen").get();
```

# DEMO: MAPPING A C FUNCTION

In the *jdk22-features-excited/src/main/java* directory, navigate to the `com.evolutionnext.foreignfunctionmemory` package and run and analyze `InvokeStrLen` and we can view how we can map the C function

# DESCRIBING THE C FUNCTION SIGNATURE

- A downcall method handle also requires a description of the native function's signature, which is represented by a `FunctionDescriptor` instance
- A *function descriptor* describes the layouts of the native function's arguments and its return value, if any
- Each layout in a function descriptor maps to a Java type, which is the type that should be used when invoking the resulting downcall method handle.
- Once the downcall handle is obtained, you can invoke the call on your C library.

# DEMO: OBTAINING THE FUNCTION SIGNATURE

In the *jdk22-features-excited/src/main/java* directory, navigate to the `com.evolutionnext.foreignfunctionmemory` package and run and analyze `InvokeStrLen` and we finally make our call to `strlen` in a C library

# UPCALLS: PASSING JAVA CODE AS A FUNCTION

- An upcall is a call from native code back to Java code.
- An upcall stub enables you to pass *Java code* as a function pointer to a foreign function.

# USING `qsort`

- `qsort` C-library takes four arguments
  - `base`: Pointer to the first element of the array to be sorted
  - `nbemb`: Number of elements in the array
  - `size`: Size, in bytes, of each element in the array
  - `compar`: Pointer to the function that compares two elements

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

# THE FOLLOWING SECTIONS DESCRIBES MAKING AN UPCALL:

1. Defining the Java Method That Compares Two Elements
2. Creating a Downcall Method Handle for the Function
3. Creating a Method Handle to Represent a Java Method
4. Creating a Function Pointer from the Method Handle
5. Allocating Off-Heap Memory
6. Calling the C Function
7. Copying the Result Off-Heap to On-Heap Memory

# DEMO: MAKING AN UPCALL

In the *jdk22-features-excited/src/main/java* directory, navigate to the `com.evolutionnext.foreignfunctionmemory` package and run and analyze `InvokeQSort`

# SUMMARY: DOWNCALLS VS. UPCALLS

| Type | Description |
| --- | --- |
| *Downcall* | Calling a method to C from Java |
| *Upcall* | Calling a method to Java from C |

# FOREIGN FUNCTIONS THAT RETURN POINTERS

### Recall the memory allocation calls in C

- `malloc` - allocates requested memory, and return a pointer to it
- `free` - frees up that memory

### Dealing with pointers in Java

- When you invoke a native function that returns a pointer, like `malloc`, the Java runtime has no insight into the size or the lifetime of the memory segment the pointer points to
- Consequently, the FFM API uses a zero-length memory segment to represent this kind of pointer.

# FFM AND ZERO-LENGTH MEMORY SEGMENTS

**The FFM API uses zero-length memory segments to represent the following**

- Pointers returned from a foreign function
- Pointers passed by a foreign function to an upcall
- Pointers read from a memory segment

# DEMO: RETURN AND DEALING WITH POINTER

In the *jdk22-features-excited/src/main/java* directory, navigate to the `com.evolutionnext.foreignfunctionmemory` package and run and analyze `InvokeReturningPointers`

# MEMORY LAYOUTS AND STRUCTURED ACCESS

- Accessing structured data using only basic operations can lead to hard-to-read code that's difficult to maintain.
- Instead, you can use memory layouts to more efficiently initialize and access more complicated native data types such as C `structures`.

# DEMO: RETURN AND DEALING WITH POINTER

In the *jdk22-features-excited/src/main/java* directory, navigate to the `com.evolutionnext.foreignfunctionmemory` package and run and analyze `InvokeStructure`

# CHECKING `errno` ERRORS

- Some C standard library functions indicate errors by setting the value of the C standard library macro `errno`.
- You can access this value with a FFM API linker option.
- The `Linker::downcallHandle` method contains a varargs parameter that enables you to specify additional linker options.
- These parameters are of type `Linker.Option`.

# `captureCallState`

- One linker option is Linker.Option.captureCallState(String…), which you use to save portions of the execution state immediately after calling a foreign function associated with a downcall method handle.
- You can use it to capture certain thread-local variables. When used with the "errno" string, it captures the `errno` value as defined by the C standard library.
- Specify this linker option (with the "errno" string) when creating a downcall handle for a native function that sets errno.

# DEMO: RETURN AND DEALING WITH POINTER

In the *jdk22-features-excited/src/main/java* directory, navigate to the `com.evolutionnext.foreignfunctionmemory` package and run and analyze `HandlingErrors`

# `jextract`

- The `jextract` tool mechanically generates Java bindings from a native library header file.
- The bindings that this tool generates depend on the Foreign Function and Memory (FFM) API.
- With this tool, you don't have to create *downcall* and *upcall* handles for functions you want to invoke; the `jextract` tool generates code that does this for you.

# GETTING `jextract`

Obtain the tool from the following site:

https://jdk.java.net/jextract/

# RUNNING jextract

```
jextract -l :<shared-libraries> \
  --output <directory containing code generated by jextract> \
  -I <dependency header-files> \
  -t org.python <location of .h files>
```

# DEMO: RUNNING `jextract`

In the *jdk22-features-excited/src/main/java* directory, run and analyze *jextract_run.sh*. Then look at *gensrc* and view the java files that were created.

# LAUNCH MULTI-FILE SOURCE-CODE PROGRAMS

# LAUNCH MULTI-FILE SOURCE-CODE PROGRAMS

- Enhance the java application launcher to be able to run a program supplied as multiple files of Java source code.
- This will make the transition from small programs to larger ones more gradual, enabling developers to choose whether and when to go to the trouble of configuring a build tool.

# HOW DOES IT WORK?

- If a main class is invoked, the launcher will find other class that it is referring
- Some code may be compiled before the program starts executing while other code may be compiled lazily, on the fly.
- Multiple classes can be declared in one *.java* file, and are all compiled together.
- Classes co-declared in a *.java* file are preferred to classes declared in other .java files and have higher-precedence

# DEMO: TRY A MULTI-FILE

In the *jdk22-features-excited/multi-file/simple* directory, and let's try a multi-file source application

# NO DUPLICATE CLASSES ALLOWED

Duplicate classes in source-code programs are prohibited.

# DEMO: TRY A MULTI-FILE

In the *jdk22-features-excited/multi-file/invalid* directory, and let's prove that you cannot have conflicting names.

# USING PRECOMPILED CLASSES

- Programs that depend on libraries on the class path or the module path can also be launched from source files.
- You can run the programs by running:

```
$ java --class-path '*' <file>.java
```

- Here the '*' argument to the `--class-path`, or `-cp` option puts all the JAR files in the directory on the class path; the asterisk is quoted to avoid expansion by the shell.
- If you wish to use a *libs* directory use the directory name

```
$ java --class-path 'libs/*' <file>.java
```

# DEMO: TRY A LIST OF DEPENDENCIES

In the *jdk22-features-excited/multi-file/dependencies* directory, and let's integrate other dependencies into our project

# ENHANCED `switch` EXPRESSIONS

# SWITCH EXPRESSIONS

- Previously in Java, since 1.0, there have always been `switch` *statements*
- With the advancement of JEP 361 we now have `switch` _expressions

# DEMO: SWITCH EXPRESSIONS

In the *jdk22-features-excited/src/test/java* directory, navigate to the `com.evolutionnext.enhancedswitch` package, and open the `EnhancedSwitchTest`

# TRACK ALL THE ADVANCEMENTS WITH `switch`

- Once it is all brought in you will find that all the following are now related
  - Pattern Matching
  - Enhanced `switch` expressions
  - `records`
  - `sealed` classes

# ENHANCED `switch` AND PATTERN MATCHING

- Called JEP 441 pattern matching for `switch`
- This fuses the two notions of *pattern matching* and *switch expressions*

# PATTERN MATCHING ON `record`S

- JEP 405 introduced us to pattern matching on records
- This gives us the ability to break apart or destructure a `record`

# DEMO: SWITCH EXPRESSIONS WITH PATTERN MATCHING

In the *jdk22-features-excited/src/test/java* directory, navigate to the `com.xyzcorp.patternmatching` explore the test and their corresponding pattern match

# PRIMITIVE `switch` HANDLING

JEP 455 now add support for pattern matching primitives!

```
switch (x.getStatus()) {
    case 0 -> "okay";
    case 1 -> "warning";
    case 2 -> "error";
    default -> "unknown status: " + x.getStatus();
}
```

# PRIMITIVE PATTERN MATCHING WITH `instanceof`

- Primitive type patterns in instanceof would subsume the lossy conversions built into the Java language
- Avoids the painstaking range checks that developers have been coding by hand for almost three decades.
- If the conversion lost information then the pattern does not match and the program should handle the invalid input in a different branch.

```
if (getPopulation() instanceof float pop) {
    ... pop ...
}

if (i instanceof byte b) {
    ... b ...
}
```

# UNNAMED VARIABLES

# UNNAMED VARIABLES

- Developers sometimes declare variables that they do not intend to use, whether as a matter of code style or because the language requires variable declarations in certain contexts.
- The intent of non-use is known at the time the code is written, but if it is not captured explicitly then later maintainers might accidentally use the variable, thereby violating the intent.
- If we could make it impossible to accidentally use such variables then code would be more informative, more readable, and less prone to error.

Source: JEP 456: Unnamed Variables & Patterns

# NOT REQUIRING VARIABLES IN ITERATION

```
static int count(Iterable<Order> orders) {
    int total = 0;
    for (Order order : orders)    // order is unused
        total++;
    return total;
}
```

Source: JEP 456: Unnamed Variables & Patterns

# NOT REQUIRING VARIABLES IN ITERATION

```
static int count(Iterable<Order> orders) {
    int total = 0;
    for (Order _ : orders)    // Unnamed variable
        total++;
    return total;
}
```

Source: JEP 456: Unnamed Variables & Patterns

# NOT REQUIRING VARIABLES IN "TRY-WITH-RESOURCES"

```
try (var acquiredContext = ScopedContext.acquire()) {
    ... acquiredContext not used ...
}
```

Source: JEP 456: Unnamed Variables & Patterns

# NOT REQUIRING VARIABLE IN "TRY-WITH-RESOURCES"

```
try (var _ = ScopedContext.acquire()) {    // Unnamed variable
    ... no use of acquired resource ...
}
```

Source: JEP 456: Unnamed Variables & Patterns

# NOT REQUIRING VARIABLES IN `Exception`S

```
String s = ...;
try {
    int i = Integer.parseInt(s);
    ... i ...
} catch (NumberFormatException ex) {
    System.out.println("Bad number: " + s);
}
```

Source: JEP 456: Unnamed Variables & Patterns

# NOT REQUIRING VARIABLES IN `Exception`S

```
String s = ...
try {
    int i = Integer.parseInt(s);
    ... i ...
} catch (NumberFormatException _) {        // Unnamed variable
    System.out.println("Bad number: " + s);
}
```

Source: JEP 456: Unnamed Variables & Patterns

# NOT REQUIRING VARIABLES IN LAMBDAS

```
stream.collect(Collectors.toMap(String::toUpperCase,
                                v -> "NODATA"));
```

Source: JEP 456: Unnamed Variables & Patterns

# NOT REQUIRING VARIABLES IN LAMBDAS

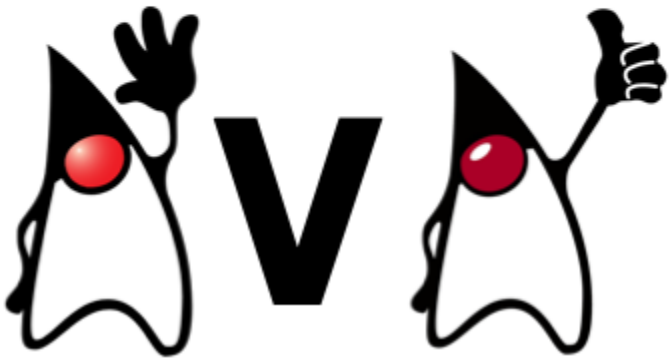
```
stream.collect(Collectors.toMap(String::toUpperCase,
                                _ -> "NODATA"))
```

Source: JEP 456: Unnamed Variables & Patterns

# DEMO: SWITCH EXPRESSIONS

In the *jdk22-features-excited/src/test/java* directory, navigate to the `com.xyzcorp.unnamedvariables` package, and open the `UnnamedVariablesTest` and review the different unnamed variable possibilities.

# WIN WITH JAVA!



# THANK YOU


- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Mastodon: https://mastodon.social/@dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext