ALTERNATES TO JAVA REFLECTION AND UNSAFE USAGE

Chandra Guntur

@CGuntur

- **Java Champion**
- **JUG Leader**: NJ JavaSIG, NYJavaSIG and NYJavaSIG Hands-On Workshops
- **Speaker**: Several technology conferences and JUGs
- **Contributor** to the open sourced **eclipse-collections** and **java-katas**
- Active on **blogging** and **tweeting** about Java
- **JCP Executive Committee** secondary representative for BNY Mellon

CODE KATA LINK

https://github.com/c-guntur/java-katas/tree/master/java-handles

@CGuntur

AGENDA

- Java Reflection
  - Usage until now
  - New alternates - A Code Kata
- `sun.misc.Unsafe`
  - Usage until now
  - New alternates - A Code Kata

JRuby

2006-2007

JRuby

CAN THE JVM EASE THE JRUBY (OR ANY OTHER DYNAMIC LANGUAGE) IMPLEMENTATION?

2006-2007

JRuby

CAN THE JVM EASE THE JRUBY (OR ANY OTHER DYNAMIC LANGUAGE) IMPLEMENTATION?

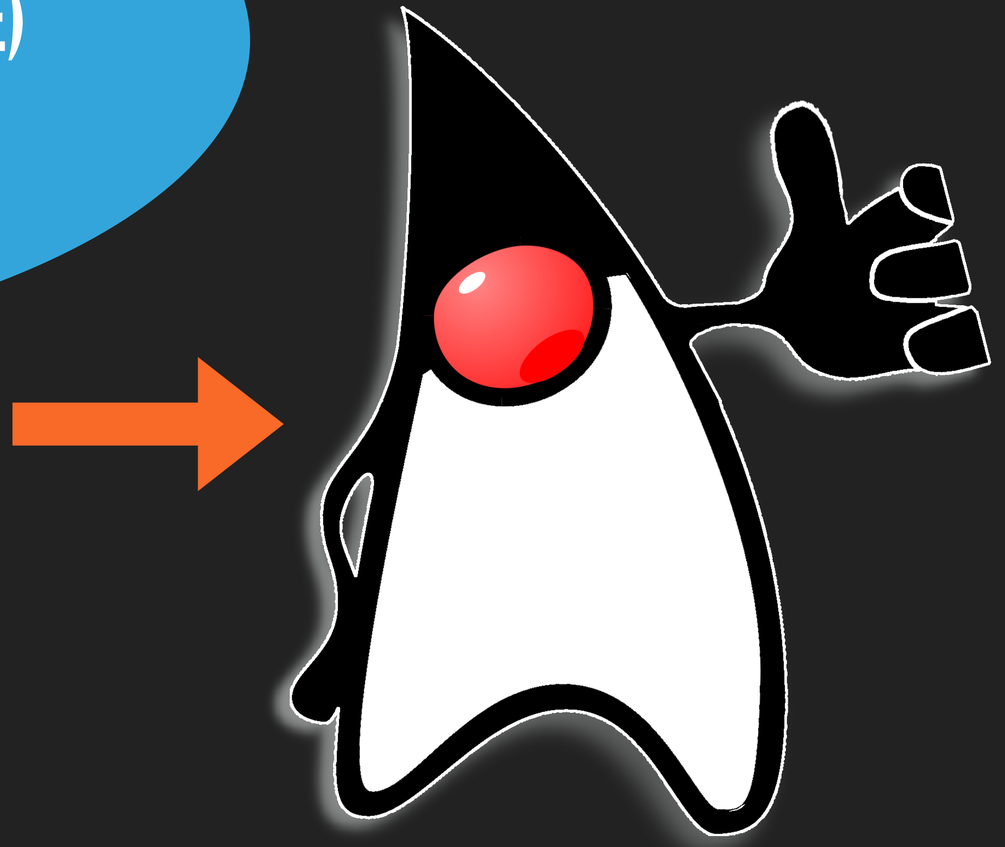
2006-2007

JRuby

CAN THE JVM EASE THE JRUBY (OR ANY OTHER DYNAMIC LANGUAGE) IMPLEMENTATION?

2006-2007

JRuby

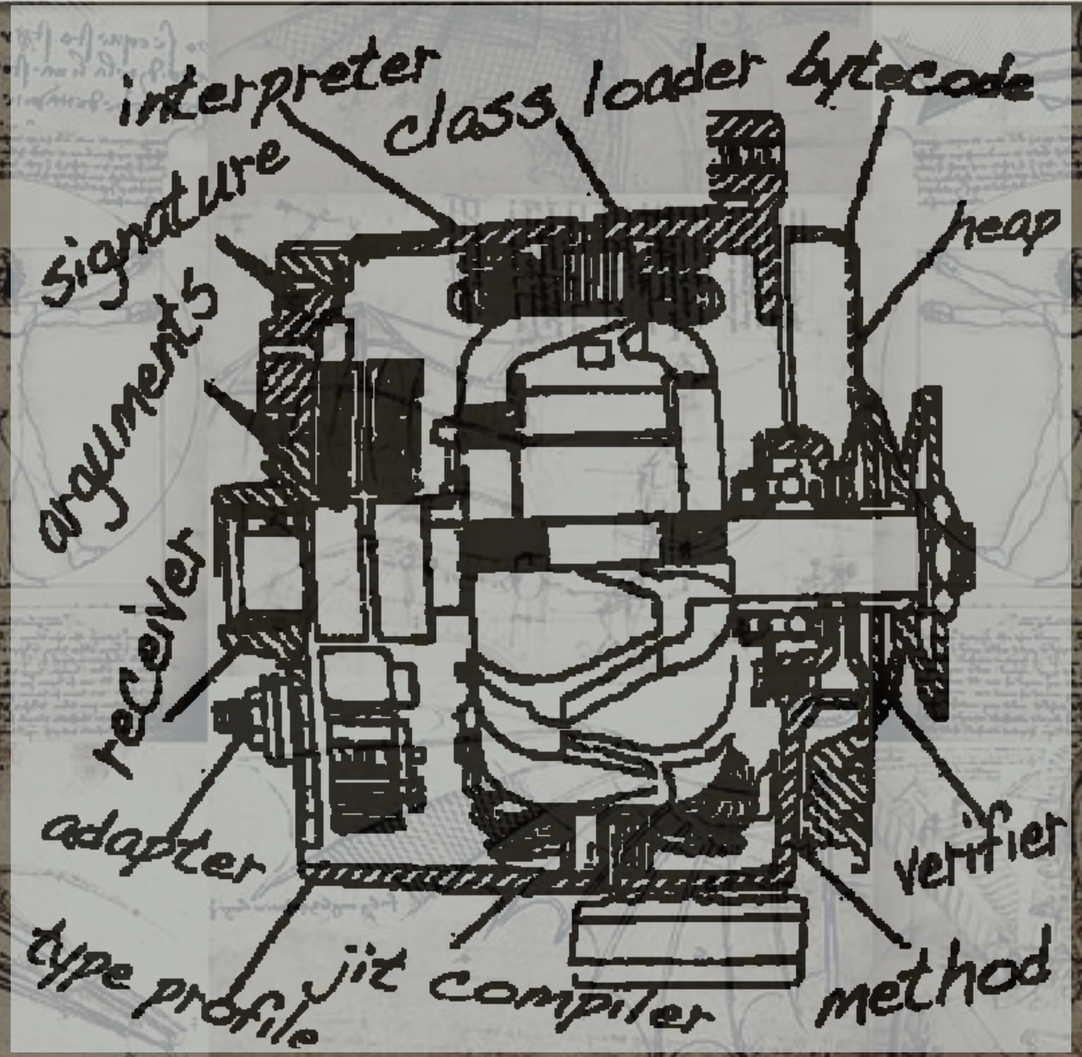


FUTURE ORIENTED CHANGES

IN JAVA

(A STEP BACK IN TIME)

Da Vinci Machine Project



Background Image credit: https://waldinadotcom.files.wordpress.com/2015/04/leonardo-da-vinci-5.jpg

Foreground Image credit: https://openjdk.java.net/projects/mlvm/images/leonardo_vm.png

@CGuntur

Future-Oriented Changes in Java - Circa 2006

- JSR-292 [*] - Multi-Language Virtual Machine
  Proposed in 2006

- JSR-292 [*] - Multi-Language Virtual Machine
  Proposed in 2006

- Code-named Da Vinci Machine Project

- JSR-292 [*] - Multi-Language Virtual Machine
  Proposed in 2006
- Code-named Da Vinci Machine Project
- Formulated to ease implementation of dynamic languages (started with JRuby [*])

- JSR-292 [*] - Multi-Language Virtual Machine
  Proposed in 2006
- Code-named Da Vinci Machine Project
- Formulated to ease implementation of dynamic languages (started with JRuby [*])

[*] JSR details: https://www.jcp.org/en/jsr/detail?id=292

[*] Read more at: https://groups.google.com/forum/#!topic/jvm-languages/28Oko4KGiQQ

Future-Oriented Changes in Java - Circa 2006

- Core features introduced in Java 7:
  1. addition of an **`invokedynamic`** instruction at the JVM level
     - gave us lambdas and method references in Java

- Core features introduced in Java 7:
  1. addition of an **`invokedynamic`** instruction at the JVM level
     - gave us lambdas and method references in Java
  2. ability to change classes & methods at runtime, dynamically
     - session focuses on this second core feature

- Core features introduced in Java 7:
  1. addition of an **`invokedynamic`** instruction at the JVM level
     - gave us lambdas and method references in Java
  2. ability to change classes & methods at runtime, dynamically
     - session focuses on this second core feature

REFLECTION

FAIREST IS ~~SNOW WHITE~~ THE DUKE (THE NEW STORY)

Every morning, the Evil Queen asked the Magic Mirror the question: *"Magic mirror in my hand, who is the fairest in the land?".*

The mirror always replied: *"My Queen, you are the fairest in the land."*

The Queen was always pleased with that, because the magic mirror never lied.

But, when **Duke** reached of age, he became as fair as the day and even more fairer than the Queen and when the Queen asked her mirror, it responded: *"My Queen, you are the fairest here, so true. But **Duke** is a thousand times fairer than you."* !


REFLECTION
@CGuntur

- Reflection introduced in Java 1.1 circa 1997

- Reflection introduced in Java 1.1 circa 1997
- Used to examine/alter behavior and structure

- Reflection introduced in Java 1.1 circa 1997
- Used to examine/alter behavior and structure
- Bypasses accessibility and instantiation rules

- Reflection drawbacks [*] :

- Reflection drawbacks [*] :
  - performance overhead
    dynamic resolution, no optimization

- Reflection drawbacks [*] :
  - performance overhead
    dynamic resolution, no optimization
  - security restrictions
    runtime permission, SecurityManager override

- Reflection drawbacks [*] :
  - performance overhead
    dynamic resolution, no optimization
  - security restrictions
    runtime permission, SecurityManager override
  - exposure of internals
    breaks abstractions and encapsulation

- Reflection drawbacks [*] :
  - performance overhead
    dynamic resolution, no optimization
  - security restrictions
    runtime permission, SecurityManager override
  - exposure of internals
    breaks abstractions and encapsulation

[*] Reflection drawbacks: https://docs.oracle.com/javase/tutorial/reflect/

WHAT CHANGED?

Reflection Replacement: MethodHandles (Java 7)

Reflection Replacement: MethodHandles (Java 7)

- JSR-292 led to lightweight references to methods

- JSR-292 led to lightweight references to methods
- The references are called Method Handles

- JSR-292 led to lightweight references to methods
- The references are called Method Handles
- A call via a handle is as fast as a statically linked Java call

- JSR-292 led to lightweight references to methods
- The references are called Method Handles
- A call via a handle is as fast as a statically linked Java call
- A bit more verbose than reflection code

Reflection Replacement: MethodHandles (Java 7)

Reflection Replacement: MethodHandles (Java 7)

- APIs to investigate :

- APIs to investigate :
  - `java.lang.invoke.MethodHandles.Lookup`

- APIs to investigate :
  - `java.lang.invoke.MethodHandles.Lookup`
  - `java.lang.invoke.MethodHandle`

- APIs to investigate :
  - `java.lang.invoke.MethodHandles.Lookup`
  - `java.lang.invoke.MethodHandle`
  - `java.lang.invoke.MethodType`

MethodHandles.Lookup - “Searcher”

MethodHandles.Lookup - “Searcher”

MethodHandles.Lookup - “Searcher”

- Enclosed in a MethodHandles factory class

MethodHandles.Lookup - “Searcher”

- Enclosed in a MethodHandles factory class
- Performs access checks and security manager interactions

MethodHandles.Lookup - “Searcher”

- Enclosed in a MethodHandles factory class
- Performs access checks and security manager interactions
  - If allowed, creates a method handle reference

MethodHandles.Lookup - “Searcher”

MethodHandles.Lookup - “Searcher”

Has several **lookup** methods such as :

MethodHandles.Lookup - “Searcher”

Has several **lookup** methods such as :

- `findGetter`
- `findStaticSetter`
- `findVirtual`
- `findConstructor`
- `...`

MethodHandle - “Executor”

MethodHandle - “Executor”

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/invoke/MethodHandle.html

MethodHandle - “Executor”

- Direct executable reference to an underlying method

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/invoke/MethodHandle.html

MethodHandle - “Executor”

- Direct executable reference to an underlying method
- Not distinguished by name of method or class

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/invoke/MethodHandle.html

MethodHandle - “Executor”

- Direct executable reference to an underlying method
- Not distinguished by name of method or class
- Immutable and stateless

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/invoke/MethodHandle.html

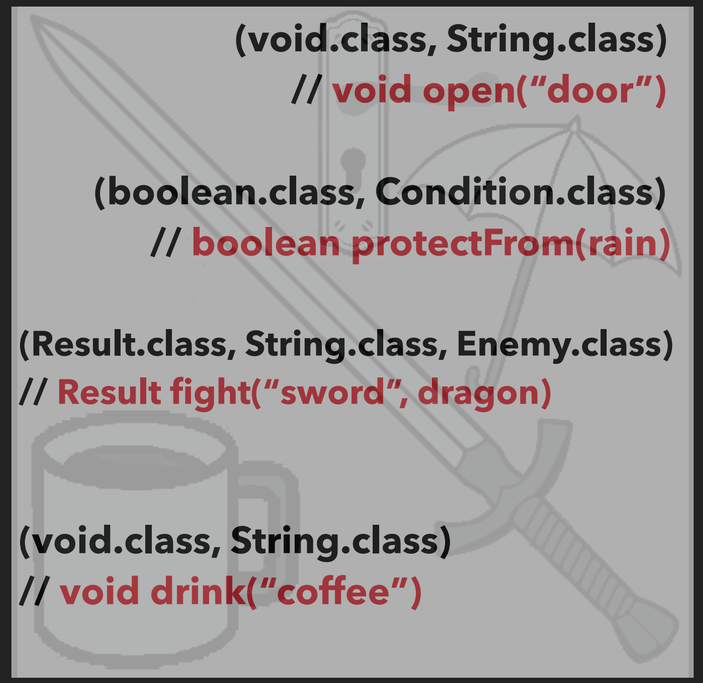
MethodHandle - “Executor”

Special invoker methods :

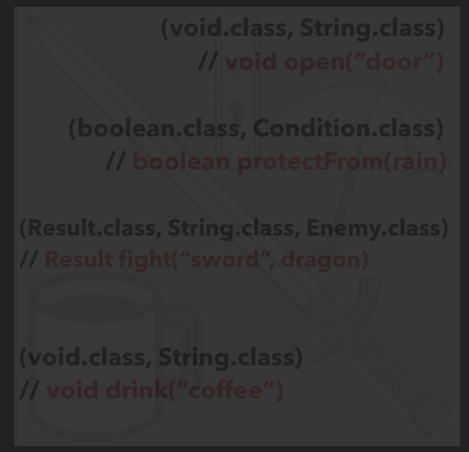
- `invoke`
- `invokeExact`
- `invokeWithArguments`
- `...`

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/invoke/MethodHandle.html

MethodType - “Discriminator”

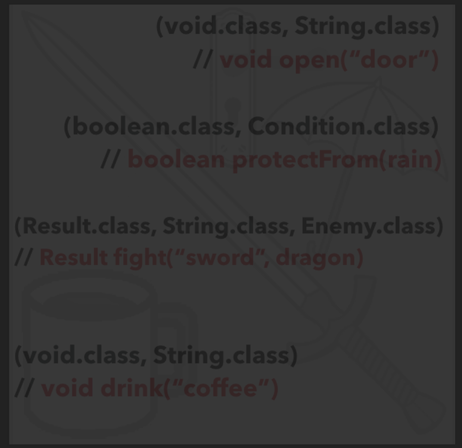


MethodType - “Discriminator”



- Represents return type & input parameters of a method :
  - First parameter of a method type is **return type**
  - Other parameters are input **parameter types**
- Immutable

MethodType - “Discriminator”



- Primitives, arrays and void (return) values are types :
  - `int.class`
  - `double.class`
  - `void.class`
  - `int[].class`
  - `...`

CODING TIME !

Search: Use `MethodHandles.Lookup` to search for methods

Execute: Use MethodHandle methods to execute the method.

- `invoke()`
- `invokeExact()`
- `invokeWithArguments()`
- …

Get Handle: Acquire a MethodHandle **using**:

- Method **type** approach =>
  - Describe the method to create a `MethodType`
  - `find***()` method to extract a `MethodHandle`

Approach 1

Get Handle: Acquire a MethodHandle using:

- Method **signature** approach =>
  - Get a `Method` instance by name and arity
  - `unreflect()` the method to extract a `MethodHandle`

Approach 2

UNSAFE

HANSEL AND GRETEL AND THE DUKE (THE NEW STORY)

Hansel, Gretel and **Duke** stood stunned for a moment, and then, without caution or hesitation, they ran to the house and began stuffing their faces full of candy.

"What little pretties have come to my house today?" screeched an odd voice.

The **three** stopped short and stared at one another – each was messy and covered in candy.

"Oh, **three** little ones to enjoy!" the voice screeched again. Hansel, Gretel and **Duke** turned and stared at a very small woman with scraggly hair wearing a long pink dress and a purple robe. She was very strange looking and smelled quite funny – a little too sweet.

The old woman then invited Hansel, Gretel and **Duke** into her house.


uNsafe
@CGuntur

- Low level memory information and direct memory access

- Low level memory information and direct memory access
- Object, Class, Array, field and static field manipulation

- Low level memory information and direct memory access
- Object, Class, Array, field and static field manipulation
- Avoiding initialization/constructor calls

- Low level memory information and direct memory access
- Object, Class, Array, field and static field manipulation
- Avoiding initialization/constructor calls
- Faster serialization/deserialization

- Low level memory information and direct memory access
- Object, Class, Array, field and static field manipulation
- Avoiding initialization/constructor calls
- Faster serialization/deserialization
- … and many other uses !

WHAT CHANGED?

Unsafe Replacement: VarHandle (Java 9)

- JSR-292 led to lightweight references to attributes

- JSR-292 led to lightweight references to attributes

- The references are called VarHandles

- JSR-292 led to lightweight references to attributes
- The references are called VarHandles
- Standard replacements for features in :
  - `java.util.concurrent.atomic.*`
  - `sun.misc.Unsafe`

Unsafe Replacement: VarHandle (Java 9)

The VarHandle API provides :

The VarHandle API provides :

- Field and array index element access

The VarHandle API provides :

- Field and array index element access
- Memory fences for fine-grained control of memory ordering

The VarHandle API provides :

- Field and array index element access
- Memory fences for fine-grained control of memory ordering
- A strong reachability-fence operation for an object

Unsafe Replacement: VarHandles - 4 Goals

Safety

GOAL 1

Safety

JVM cannot be placed in a corrupt memory state

GOAL 1

Safety

JVM cannot be placed in a corrupt memory state

- in both fields and arrays:
  - **Either** content data types must match
  - **Or** content data must be 'castable' to field type

GOAL 1

Safety

JVM cannot be placed in a corrupt memory state

- in both fields and arrays:
  - **Either** content data types must match
  - **Or** content data must be 'castable' to field type

- and in arrays:
  - Array indexes must exist in order to be written to/accessed

GOAL 1

Unsafe Replacement: VarHandles - 4 Goals

Integrity

GOAL 2

Integrity

GOAL 2

Field access rules cannot be violated

Integrity

GOAL 2

Field access rules cannot be violated

- Same rules as for `getfield` and `putfield` byte codes

Integrity

GOAL 2

Field access rules cannot be violated

- Same rules as for `getfield` and `putfield` byte codes
- A final field cannot be updated or modified

Unsafe Replacement: VarHandles - 4 Goals

Performance

GOAL 3

Performance

Performance must be similar to `Unsafe` operations

GOAL 3

Unsafe Replacement: VarHandles - 4 Goals

Usability

GOAL 4

Usability

GOAL 4

New API should be:

Usability

GOAL 4

New API should be:

- Friendlier and humane

Usability

GOAL 4

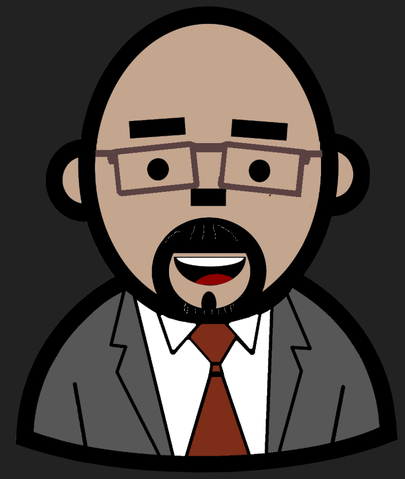
New API should be:

- Friendlier and humane
- More consistent than `Unsafe` APIs

CODING TIME !

- Get a `MethodHandles.Lookup` via :
  - `lookup().in(requestedLookupClass)`
  - `privateLookupIn(targetClass, lookup)`
- Find a `VarHandle` :
  - For fields :
    - `findVarHandle(receiverClass, attributeName, attributeType)`
  - For array elements :
    - `arrayElementVarHandle(arrayClass)`

Thank you !



https://twitter.com/CGuntur

https://cguntur.me

https://linkedin.com/in/cguntur

Appendix 1 - MethodHandle - Compilation and Execution

- MethodHandles are dynamically & strongly typed. They are immutable and have no visible state.
- Each MethodHandle reports its type via a type descriptor.
- The type descriptor is of type MethodType whose structure is a series of classes.
- The first of the types is the return type (including void.class, for methods with no returns).
- MethodHandles have special invoker methods invoke(), invokeExact() etc.
- The invoke() method can accept a range of calls that match the MethodType.
- Both invoke(), invokeExact() **compile** to an invokevirtual instruction.
- The MethodHandle instance itself is pushed onto the stack prior to the arguments being padded.
- On first execution, links via instruction names symbolically resolved & legality of the call verified.
- Post-linkage, method handle's type is checked to match type descriptor.
- The invoke() may method additionally invoke an asType() to adapt to the defined MethodType.
- The invoke() may thus create a secondary "exact" MethodHandle if a right candidate is found.

Appendix 2 - MethodHandle - Limits

- The JVM imposes on all methods and constructors of any kind an absolute limit of 255 stacked arguments.
- This limit can appear more restrictive in certain cases:
  - A long or double argument counts (for purposes of arity limits) as two argument slots.
  - A non-static method consumes an extra argument for the object on which the method is called.
  - A constructor consumes an extra argument for the object which is being constructed.
  - Since a method handle's invoke method (or other signature-polymorphic method) is non-virtual, it consumes an extra argument for the method handle itself, in addition to any non-virtual receiver object.
- Thus some method handles can't be created, because of the JVM limit on stacked arguments.
- Attempts to create such method handles lead to an IllegalArgumentException.
- In particular, a method handle's type must not have an arity of the exact maximum 255.

Appendix 3 - VarHandle - Compilation and Execution

- VarHandles are dynamically & strongly typed. They are immutable and have no visible state.
- VarHandle provides access to a variable via one of several access modes.
- The access modes include plain read/write access, volatile read/write access, and compare-and-set.
- A VarHandle has
  - a variable type T - the type of variable referenced by this VarHandle
  - a list of coordinate types $CT_1 \ldots CT_n$ - types of coordinate expressions that jointly locate the variable
- VarHandle calls compile to an invokevirtual instruction.
- VarHandles expose getters, setters and compareAndSet/Exchange operations with each access mode.
- In addition VarHandle has static memory fence operations.
- Breaking with Core Reflection, VarHandle access checks are performed only once on creation.
- VarHandles to non-public variables/members of non-public classes should be handled with care.

Appendix 4 - VarHandle - Access Modes

Memory Ordering	Access	Notes
plain	read/write	Atomic only for references and lower primitives (below 32 bit)
opaque	read/write	Only atomic for the same variable
acquire	read	Subsequent load & storage order not changed before access
release	write	Prior load and storage order not changed before access
volatile	read/write	Fully ordered operations on variables in memory

Plain reads and writes guarantee bitwise atomicity for references and primitives under 32 bits.

Opaque operations are bitwise atomic and coherently ordered with respect to access to the same variable.

Acquire and Release operations are Opaque. Acquire reads ordered only after matching Release writes.

Volatile operations are fully ordered with respect to each other.

Appendix 5 - VarHandle - Fences

- Fences are operators that provide fine-grained control over memory ordering.
- VarHandle API provides five static methods for fencing operations.
- Fences are temporary protective barriers around memory.

Fence	Notes
fullFence()	**Loads and stores** before the fencing will not be re-ordered with **loads and stores** after.
acquireFence()	**Only loads** before the fencing will not be re-ordered with **loads and stores** after.
releaseFence()	**Loads and stores** before the fencing will not be re-ordered with **only stores** after.
loadLoadFence()	**Only loads** before the fencing will not be re-ordered with **only loads** after.
storeStoreFence()	**Only stores** before the fencing will not be re-ordered with **only stores** after.