Final Keyword

How it behaves at Run-Time

final Keyword

final can be used with class, variable, method

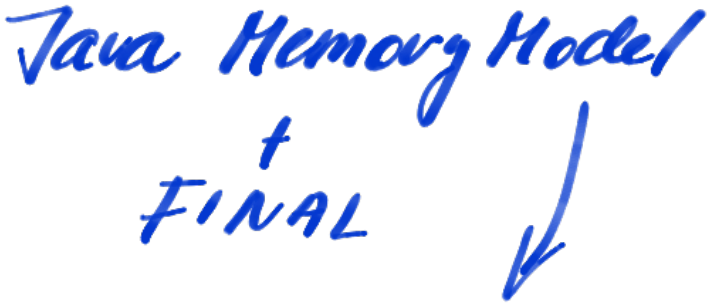
(1) Restrict changing value of variable

(2) Restrict method overriding

(3) Restrict inheritance

Final Keyword

- **COMPILE-TIME**
- Create unmodifiable references
- Restrict a method overriding
- Restrict a class inheritence



- **RUNTIME-TIME**
- Part of Java Memory Model
- Guarantees visibility in a multi-threaded application
- Safe initialization for objects, arrays and collections
- JIT Optimizations

https://docs.oracle.com/javase/specs/jls/se11/html/jls-17.html#jls-17.5

Is a FINAL field really constant ?

```java
@ConstantsAndAnnotation.ClassAnnotation(
        stringParam = ConstantsAndAnnotation.MY_STRING,
        stringArray = {"2", "1"}
)
class ConstantsAndAnnotation {

    static final String MY_STRING = "My Param";

    static final String[] STRING_ARRAY = {"2", "1"};

    @Retention(RetentionPolicy.CLASS)
    public @interface ClassAnnotation {
        String stringParam();
        String[] stringArray();
        String[] stringArray2() default {};
    }
}
```

compile-time

- compiler knows only about compile-time constants

```java
class IsConstant {

    private final int value;

    IsConstant(int value) {
        this.value = value;
    }
}
```

Constant 2
Really 2

```
$ javap -v -p ConstantsAndAnnotation.class
class pbouda.bytecode.examples.ConstantsAndAnnotation
Constant pool:
  #1 = Methodref          #7.#34         // java/lang/Object."<init>":()V
    ...
  #28 = Utf8
    Lpbouda/bytecode/examples/ConstantsAndAnnotation$ClassAnnotation;
  #29 = Utf8               stringParam
  #30 = Utf8               My Param
  #31 = Utf8               stringArray
  #32 = Utf8               2
  #33 = Utf8               1
  #34 = NameAndType        #17:#18        // "<init>":()V
  #35 = Utf8               java/lang/String
  #36 = NameAndType        #15:#16
    // STRING_ARRAY:[Ljava/lang/String;
  #37 = Utf8
    pbouda/bytecode/examples/ConstantsAndAnnotation
  #38 = Utf8               java/lang/Object
  #39 = Utf8
    pbouda/bytecode/examples/ConstantsAndAnnotation$ClassAnnotation
{
  static final java.lang.String MY_STRING;
    descriptor: Ljava/lang/String;
    flags: ACC_STATIC, ACC_FINAL
    ConstantValue: String My Param

  static final java.lang.String[] STRING_ARRAY;
    descriptor: [Ljava/lang/String;
    flags: ACC_STATIC, ACC_FINAL
...
SourceFile: "ConstantsAndAnnotation.java"
RuntimeInvisibleAnnotations:
  0: #28(#29=s#30,#31=[s#32,s#33])
```

```
public class IsConstant {

    private static final int VALUE = Integer.valueOf(1);

}
```

? IT'S CONSTANT !?

javac is not able to evaluate methods, does only very simple folding

- It doesn't belong to a particular object, it's not supposed to be changed
- Bytecode compiler doesn't know the value
  - it's initialized at runtime

JEP 334: JVM Constants API

- This is JUST-IN-TIME constant
  - Aggressively optimized and folded by JIT
- It can be even 5x faster then instance final field

benchmarks and examples

https://shipilev.net/jvm/anatomy-quarks/15-just-in-time-constants/

Can we trust Instance Final Field ?

- JLS allows ignoring the updates
  - Frameworks depend on visibility of all changes
  - Code cannot be aggressively optimized and folded
- However, there are some exceptions
  - *java/lang/invoke* and *sun/invoke packages*
  - Anonymous, Boxed classes and String
  - MethodHandle, VarHandle, Atomic*FieldUpdater

https://shipilev.net/jvm/anatomy-quarks/17-trust-nonstatic-final-fields/

FINAL fields and multi-threading applications?

- Can FINAL keyword help us in multi-threaded application?
- Is there any connection to Java Memory Model?
- How to safely create a shared object in multi-threaded environment?

https://docs.oracle.com/javase/specs/jls/se11/html/jls-17.html#jls-17.5

Object Initialization

```
public class MyObject {

    private int x;
    private int y;

    public MyObject() {
        this.x = 1;
        this.y = 2;
    }
}
```

What happened

```
MyObject temp = <new>
temp.x = 1;
temp.y = 2;
mo = temp;
```

} Constructor

What is reality.[2]

- everything can be reordered
- even after a reference publish !

Publish a reference

⇒ JIT and CPU OPTIMIZATIONS

```java
public class Initialization {

  private Object obj1 = new Object();
  private Object obj2;
  private Object obj3;

  {
    this.obj2 = new Object();
  }

  public Initialization() {
    this.obj3 = new Object();
  }
}
```

```
public pbouda.bytecode.examples.Initialization();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=3, locals=1, args_size=1
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: aload_0
       5: new           #2                  // class java/lang/Object
       8: dup
       9: invokespecial #1                  // Method java/lang/Object."<init>":()V
      12: putfield      #3                  // Field obj1:Ljava/lang/Object;
      15: aload_0
      16: new           #2                  // class java/lang/Object
      19: dup
      20: invokespecial #1                  // Method java/lang/Object."<init>":()V
      23: putfield      #4                  // Field obj2:Ljava/lang/Object;
      26: aload_0
      27: new           #2                  // class java/lang/Object
      30: dup
      31: invokespecial #1                  // Method java/lang/Object."<init>":()V
      34: putfield      #5                  // Field obj3:Ljava/lang/Object;
      37: return
```

Always the same result in Bytecode!

Safe Publication

- Initialize an object reference from a static initializer
  - Class is always initialized by a single initializer thread
  - Initialization *happens-before* everything that uses that class (no synchronization needed)
- Store reference into *volatile* field or *AtomicReference*
- Guard a field using a lock
- Store reference into *final* field and never leak *this* from a given object

Safe initialization via FINAL

- JMM adds a synthetic **freeze** action when a constructor completes
- **Freeze** ensures that the 2nd thread sees *null* or a reference of a object with a fully constructed field marked as FINAL
- **Freeze** is implemented as store barrier (*sfence* instruction on x86)
- Other operations, after a freeze action, on the final field are racy between 2 threads and must be properly synchronized
  - JVM spec ensures that the 2nd thread sees the state of the final field corresponding to the freeze action - no more, no less

```java
public class NonFinalVariable {

    public Integer value;

    public NonFinalVariable() {
        this.value = 1;
    }
}
```

difference.?

```java
public class FinalVariable {

    public final Integer value;

    public FinalVariable() {
        this.value = 1;
    }
}
```

What can be a result.?

NFV temp = <new>
temp.value = 1;
nfv = temp;

Correct even in multithreaded system

NFV temp = <new>
nfv = temp
temp.value = 1

2nd thread can observe a reference with (null) value Field.

FV temp = <new>
temp.value = 1
<freeze value>
fv = temp

Freeze ensures correct ordering and publication.

```java
public class FinalVariable {

    public final int[] value;

    public FinalVariable() {
        this.value = new int[5];
        this.value[0] = 1;
        this.value[1] = 2;
        this.value[2] = 3;
    }
}
```

<Freeze value>

```java
public class FinalVariable {

    public final List<Integer> value;

    public FinalVariable() {
        this.value = new ArrayList<>();
        this.value.add(1);
        this.value.add(2);
        this.value.add(3);
    }
}
```

<Freeze value>

Arrays and collections are properly populated and exposed, but only with items assigned in the constructor!

Thread 1

```
FinalVariable fv = new FinalVariable();
fv.value.add(4);
```

shared variable

Thread 2

```
fv.size() // can observe
// 3 or 4
```

```java
public class FinalVariable {

    public final InnerClass value;

    public FinalVariable() {
        this.value = new InnerClass( val: 1);
    }

    private static class InnerClass {
        public int innerValue;

        public InnerClass(int val) {
            this.innerValue = val;
        }
    }
}
```

<freeze value>

value is exposed correctly as a whole!

even if **innerValue** is not marked as final.

entire graph of objects beginning from a final field is correctly visible.

no final!

```java
public class PartiallyFinalVariables {

    public final int value1;
    public int value2;
    public int value3;

    public PartiallyFinalVariables() {
        this.value1 = 1;
        this.value2 = 2;
        this.value3 = 3;
    }
}
```

Works on HOTSPOT!
but it's not supported by specification!

```java
public class FinalVariables {

    public final int value1;
    public final int value2;
    public final int value3;

    public FinalVariables() {
        this.value1 = 1;
        this.value2 = 2;
        this.value3 = 3;
    }
}
```

correct implementation according to JLS (specification)

ALWAYS CODE AGAINST SPECIFICATION!

<< **jcstress demo** >>

http://openjdk.java.net/projects/code-tools/jcstress/

Never leak your THIS !!

- Leaking THIS in your constructor bypasses **freeze** mechanism at the end of the constructor and can lead to a non-deterministic behavior.

```java
public class LeakingThis {
  public LeakingThis() {
    new Thread(new Runnable() {
      @Override
      public void run() {
        out.println("LEAK");
      }
    }).start();
  }
}
```

Generated class!

```
class pbouda.bytecode.examples.LeakingThis$1 implements java.lang.Runnable
  final pbouda.bytecode.examples.LeakingThis this$0;
    descriptor: Lpbouda/bytecode/examples/LeakingThis;
    flags: ACC_FINAL, ACC_SYNTHETIC

  pbouda.bytecode.examples.LeakingThis$1(pbouda.bytecode.examples.LeakingThis);
    descriptor: (Lpbouda/bytecode/examples/LeakingThis;)V
    flags:
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
         1: aload_1
         2: putfield      #1 // Field this$0:Lpbouda/bytecode/examples/LeakingThis;
         5: aload_0
         6: invokespecial #2 // Method java/lang/Object."<init>":()V
         9: return
      LineNumberTable:
        line 7: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      10     0  this   Lpbouda/bytecode/examples/LeakingThis$1;
            0      10     1 this$0   Lpbouda/bytecode/examples/LeakingThis;
```

Reference to an outer object!

```java
public class LeakingThis {
    public LeakingThis() {
        var LEAK = Integer.valueOf(1);
        new Thread(() ->
                out.println(LEAK)).start();
    }
}
```

Generated method to support lambdas!

```
private static void lambda$new$0(java.lang.Integer);
  descriptor: (Ljava/lang/Integer;)V
  flags: ACC_PRIVATE, ACC_STATIC, ACC_SYNTHETIC
  Code:
    stack=2, locals=1, args_size=1
       0: getstatic     #7 // Field java/lang/System.out:Ljava/io/PrintStream;
       3: aload_0
       4: invokevirtual #8 // Method java/io/PrintStream.println:(Ljava/lang/Object;)V
       7: return
    LineNumberTable:
      line 12: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       8     0  LEAK   Ljava/lang/Integer;
```

to leaking outer object just passing dependent fields

```java
public class LeakingThis {
  final Integer LEAK = Integer.valueOf(1);

  public LeakingThis() {
    new Thread(() ->
        System.out.println(LEAK)).start();
  }
}
```

```
private void lambda$new$0();
  descriptor: ()V
  flags: ACC_PRIVATE, ACC_SYNTHETIC
  Code:
    stack=2, locals=1, args_size=1
       0: getstatic     #8   // Field java/lang/System.out:Ljava/io/PrintStream;
       3: aload_0
       4: getfield      #3   // Field LEAK:Ljava/lang/Integer;
       7: invokevirtual #9   // Method java/io/PrintStream.println:(Ljava/lang/Object;)V
      10: return
    LineNumberTable:
      line 10: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      11     0  this   Lpbouda/bytecode/examples/LeakingThis;
```

generated method

loading and leaking THIS

```java
public class LeakingThis {
    final String LEAK = "LEAK";

    public LeakingThis() {
        new Thread(() ->
            System.out.println(LEAK)).start();
    }
}
```

generated method

```
private void lambda$new$0();
  descriptor: ()V
  flags: ACC_PRIVATE, ACC_SYNTHETIC
  Code:
    stack=2, locals=1, args_size=1
       0: getstatic     #8    // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #2    // String LEAK
       5: invokevirtual #10   // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
    LineNumberTable:
      line 10: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       9     0  this   Lpbouda/bytecode/examples/LeakingThis;
```

THIS is not used, value and reference is used from Runtime Constant Pool

THIS leaked in Variable table!

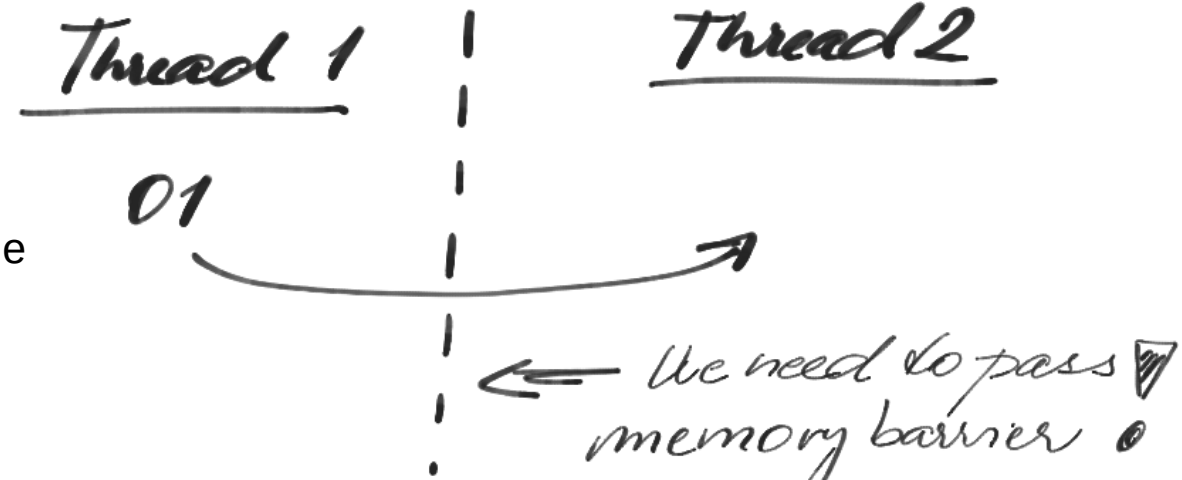
!! I can’t use Final !!

- Static Initializer (only for singletons)
  - Class is always initialized by a single initializer thread
  - Initialization *happens-before* everything that uses that class (no synchronization needed)
- Thread-confinement
  - Run a block of code (task, job, request, ...) only on one dedicated thread
  - Frameworks with a matured and well-documented thread model – e.g. EventLoop (Netty)
- Stack-confinement
  - Reachable only through local variables
- ThreadLocal
  - Maintains a per-thread value

!! I can’t use Final !!

- Synchronized block (implicit locks)
- Explicit locks, volatile keyword
- CAS operations
  - Atomic*, *Adder, Atomic*FieldUpdater
- java.util.concurrent package
  - SynchronizedMap, ConcurrenHashMap
  - CopyOnWriteArray(List|Set),
  - Synchronized(List|Set)
  - BlockingQueue, ConcurrentLinkedQueue
- VarHandles, ? Unsafe ?

How to creat an object and pass it to another thread with full visibility of all fields.?



Summary

Always start with a FINAL variable and change it only if the object's state is really supposed to be mutable.

.. then you need to handle visibility problems using memory barriers generated by LOCKS or VOLATILE.