– O' - O contains no external actions

Note that a behavior B does not describe the order in which the external actions in B are observed, but other (internal) constraints on how the external actions are generated and performed may impose such constraints.

17.5 `final` Field Semantics

Fields declared final are initialized once, but never changed under normal circumstances. The detailed semantics of `final` fields are somewhat different from those of normal fields. In particular, compilers have a great deal of freedom to move reads of `final` fields across synchronization barriers and calls to arbitrary or unknown methods. Correspondingly, compilers are allowed to keep the value of a `final` field cached in a register and not reload it from memory in situations where a non-`final` field would have to be reloaded.

`final` fields also allow programmers to implement thread-safe immutable objects without synchronization. A thread-safe immutable object is seen as immutable by all threads, even if a data race is used to pass references to the immutable object between threads. This can provide safety guarantees against misuse of an immutable class by incorrect or malicious code. `final` fields must be used correctly to provide a guarantee of immutability.

An object is considered to be *completely initialized* when its constructor finishes. A thread that can only see a reference to an object after that object has been completely initialized is guaranteed to see the correctly initialized values for that object's `final` fields.

The usage model for `final` fields is a simple one: Set the `final` fields for an object in that object's constructor; and do not write a reference to the object being constructed in a place where another thread can see it before the object's constructor is finished. If this is followed, then when the object is seen by another thread, that thread will always see the correctly constructed version of that object's `final` fields. It will also see versions of any object or array referenced by those `final` fields that are at least as up-to-date as the `final` fields are.

Example 17.5-1. `final` Fields In The Java Memory Model

The program below illustrates how `final` fields compare to normal fields.

```
class FinalFieldExample {
    final int x;
    int y;
```

```
        static FinalFieldExample f;

        public FinalFieldExample() {
            x = 3;
            y = 4;
        }

        static void writer() {
            f = new FinalFieldExample();
        }

        static void reader() {
            if (f != null) {
                int i = f.x;  // guaranteed to see 3
                int j = f.y;  // could see 0
            }
        }
    }
```

The class `FinalFieldExample` has a `final int` field `x` and a non-`final int` field `y`. One thread might execute the method `writer` and another might execute the method `reader`.

Because the `writer` method writes `f` *after* the object's constructor finishes, the `reader` method will be guaranteed to see the properly initialized value for `f.x`: it will read the value `3`. However, `f.y` is not `final`; the `reader` method is therefore not guaranteed to see the value `4` for it.

Example 17.5-2. `final` Fields For Security

`final` fields are designed to allow for necessary security guarantees. Consider the following program. One thread (which we shall refer to as thread 1) executes:

```
    Global.s = "/tmp/usr".substring(4);
```

while another thread (thread 2) executes

```
    String myS = Global.s;
    if (myS.equals("/tmp"))System.out.println(myS);
```

`String` objects are intended to be immutable and string operations do not perform synchronization. While the `String` implementation does not have any data races, other code could have data races involving the use of `String` objects, and the memory model makes weak guarantees for programs that have data races. In particular, if the fields of the `String` class were not `final`, then it would be possible (although unlikely) that thread 2 could initially see the default value of `0` for the offset of the string object, allowing it to compare as equal to `"/tmp"`. A later operation on the `String` object might see the correct offset of `4`, so that the `String` object is perceived as being `"/usr"`. Many security features of the Java programming language depend upon `String` objects being perceived as truly immutable, even if malicious code is using data races to pass `String` references between threads.

17.5.1 Semantics of `final` Fields

Let o be an object, and c be a constructor for o in which a `final` field f is written. A *freeze* action on `final` field f of o takes place when c exits, either normally or abruptly.

Note that if one constructor invokes another constructor, and the invoked constructor sets a `final` field, the freeze for the `final` field takes place at the end of the invoked constructor.

For each execution, the behavior of reads is influenced by two additional partial orders, the dereference chain *dereferences()* and the memory chain *mc()*, which are considered to be part of the execution (and thus, fixed for any particular execution). These partial orders must satisfy the following constraints (which need not have a unique solution):

- Dereference Chain: If an action a is a read or write of a field or element of an object o by a thread t that did not initialize o, then there must exist some read r by thread t that sees the address of o such that r *dereferences(r, a)*.
- Memory Chain: There are several constraints on the memory chain ordering:
  - If r is a read that sees a write w, then it must be the case that *mc(w, r)*.
  - If r and a are actions such that *dereferences(r, a)*, then it must be the case that *mc(r, a)*.
  - If w is a write of the address of an object o by a thread t that did not initialize o, then there must exist some read r by thread t that sees the address of o such that *mc(r, w)*.

Given a write w, a freeze f, an action a (that is not a read of a `final` field), a read r_1 of the `final` field frozen by f, and a read r_2 such that $hb(w, f)$, $hb(f, a)$, $mc(a, r_1)$, and $dereferences(r_1, r_2)$, then when determining which values can be seen by r_2, we consider $hb(w, r_2)$. (This *happens-before* ordering does not transitively close with other *happens-before* orderings.)

Note that the *dereferences* order is reflexive, and r_1 can be the same as r_2.

For reads of `final` fields, the only writes that are deemed to come before the read of the `final` field are the ones derived through the `final` field semantics.

17.5.2 Reading `final` Fields During Construction

A read of a `final` field of an object within the thread that constructs that object is ordered with respect to the initialization of that field within the constructor by the usual *happens-before* rules. If the read occurs after the field is set in the constructor, it sees the value the `final` field is assigned, otherwise it sees the default value.

17.5.3 Subsequent Modification of `final` Fields

In some cases, such as deserialization, the system will need to change the `final` fields of an object after construction. `final` fields can be changed via reflection and other implementation-dependent means. The only pattern in which this has reasonable semantics is one in which an object is constructed and then the `final` fields of the object are updated. The object should not be made visible to other threads, nor should the `final` fields be read, until all updates to the `final` fields of the object are complete. Freezes of a `final` field occur both at the end of the constructor in which the `final` field is set, and immediately after each modification of a `final` field via reflection or other special mechanism.

Even then, there are a number of complications. If a `final` field is initialized to a constant expression (§15.28) in the field declaration, changes to the `final` field may not be observed, since uses of that `final` field are replaced at compile time with the value of the constant expression.

Another problem is that the specification allows aggressive optimization of `final` fields. Within a thread, it is permissible to reorder reads of a `final` field with those modifications of a `final` field that do not take place in the constructor.

Example 17.5.3-1. Aggressive Optimization of `final` Fields

```
class A {
    final int x;
    A() {
        x = 1;
    }

    int f() {
        return d(this,this);
    }

    int d(A a1, A a2) {
        int i = a1.x;
        g(a1);
        int j = a2.x;
        return j - i;
    }

    static void g(A a) {
        // uses reflection to change a.x to 2
    }
}
```

In the `d` method, the compiler is allowed to reorder the reads of `x` and the call to `g` freely. Thus, `new A().f()` could return `-1`, `0`, or `1`.

An implementation may provide a way to execute a block of code in a `final`*-field-safe context*. If an object is constructed within a `final`-field-safe context, the reads of a `final` field of that object will not be reordered with modifications of that `final` field that occur within that `final`-field-safe context.

A `final`-field-safe context has additional protections. If a thread has seen an incorrectly published reference to an object that allows the thread to see the default value of a `final` field, and then, within a `final`-field-safe context, reads a properly published reference to the object, it will be guaranteed to see the correct value of the `final` field. In the formalism, code executed within a `final`-field-safe context is treated as a separate thread (for the purposes of `final` field semantics only).

In an implementation, a compiler should not move an access to a `final` field into or out of a `final`-field-safe context (although it can be moved around the execution of such a context, so long as the object is not constructed within that context).

One place where use of a `final`-field-safe context would be appropriate is in an executor or thread pool. By executing each `Runnable` in a separate `final`-field-safe context, the executor could guarantee that incorrect access by one `Runnable` to a object *o* will not remove `final` field guarantees for other `Runnable`s handled by the same executor.

17.5.4 Write-Protected Fields

Normally, a field that is `final` and `static` may not be modified. However, `System.in`, `System.out`, and `System.err` are `static final` fields that, for legacy reasons, must be allowed to be changed by the methods `System.setIn`, `System.setOut`, and `System.setErr`. We refer to these fields as being *write-protected* to distinguish them from ordinary `final` fields.

The compiler needs to treat these fields differently from other `final` fields. For example, a read of an ordinary `final` field is "immune" to synchronization: the barrier involved in a lock or volatile read does not have to affect what value is read from a `final` field. Since the value of write-protected fields may be seen to change, synchronization events should have an effect on them. Therefore, the semantics dictate that these fields be treated as normal fields that cannot be changed by user code, unless that user code is in the `System` class.

17.6 Word Tearing

One consideration for implementations of the Java Virtual Machine is that every field and array element is considered distinct; updates to one field or element must not interact with reads or updates of any other field or element. In particular, two threads that update adjacent elements of a byte array separately must not interfere or interact and do not need synchronization to ensure sequential consistency.

Some processors do not provide the ability to write to a single byte. It would be illegal to implement byte array updates on such a processor by simply reading an entire word, updating the appropriate byte, and then writing the entire word back to memory. This problem is sometimes known as *word tearing*, and on processors that cannot easily update a single byte in isolation some other approach will be required.

Example 17.6-1. Detection of Word Tearing

The following program is a test case to detect word tearing:

```
public class WordTearing extends Thread {
    static final int LENGTH = 8;
    static final int ITERS  = 1000000;
    static byte[] counts    = new byte[LENGTH];
    static Thread[] threads = new Thread[LENGTH];

    final int id;
    WordTearing(int i) {
        id = i;
    }

    public void run() {
        byte v = 0;
        for (int i = 0; i < ITERS; i++) {
            byte v2 = counts[id];
            if (v != v2) {
                System.err.println("Word-Tearing found: " +
                              "counts[" + id + "] = "+ v2 +
                              ", should be " + v);
                return;
            }
            v++;
            counts[id] = v;
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < LENGTH; ++i)
            (threads[i] = new WordTearing(i)).start();
    }
}
```

This makes the point that bytes must not be overwritten by writes to adjacent bytes.

17.7 Non-Atomic Treatment of `double` and `long`

For the purposes of the Java programming language memory model, a single write to a non-volatile `long` or `double` value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64-bit value from one write, and the second 32 bits from another write.

Writes and reads of volatile `long` and `double` values are always atomic.

Writes to and reads of references are always atomic, regardless of whether they are implemented as 32-bit or 64-bit values.

Some implementations may find it convenient to divide a single write action on a 64-bit `long` or `double` value into two write actions on adjacent 32-bit values. For efficiency's sake, this behavior is implementation-specific; an implementation of the Java Virtual Machine is free to perform writes to `long` and `double` values atomically or in two parts.

Implementations of the Java Virtual Machine are encouraged to avoid splitting 64-bit values where possible. Programmers are encouraged to declare shared 64-bit values as `volatile` or synchronize their programs correctly to avoid possible complications.