Safe Publication in Java: Techniques

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand what "safe publication" means in the context of Java objects
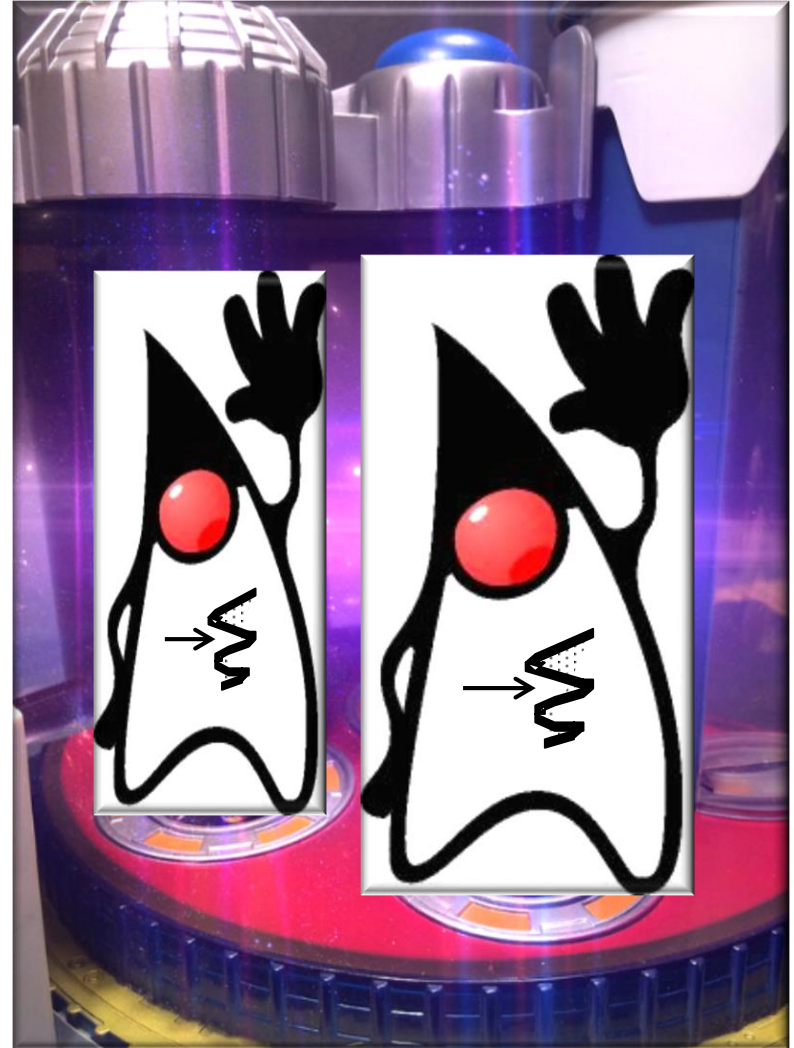- Recognize "safe publication" techniques in Java that enable multiple threads to share an object

Safe Publication Techniques in Java

Safe Publication Techniques in Java

- To publish a properly constructed Java object safely
  - The reference to the object &
  - The object's state

  must be made visible to other threads at the same time

See flylib.com/books/en/2.558.1/safe_publication.html

Safe Publication Techniques in Java

- An object can be published safely in several ways

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  > This critical section is protected by the Singleton Class instance's intrinsic lock

```java
class Singleton {
  private static Singleton sInst;

  public static Singleton instance(){
    synchronized(Singleton.class) {
      if (sInst == null)
        sInst = new Singleton();

      return sInst;
    }
  }
  ...
```

See docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

This lock ensures that both the sInst reference & the Singleton's state will be published to other threads

```
class Singleton {
  private static Singleton sInst;

  public static Singleton instance(){
    synchronized(Singleton.class) {
      if (sInst == null)
        sInst = new Singleton();

      return sInst;
    }
  }
  ...
```

See docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  The drawback with this technique is that every call to instance() is synchronized

  PROBLEM

```java
class Singleton {
  private static Singleton sInst;

  public static Singleton instance(){
    synchronized(Singleton.class) {
      if (sInst == null)
        sInst = new Singleton();

      return sInst;
    }
  }
...
```

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile

```java
class Singleton {
  private static volatile
    Singleton sInst;



  public static Singleton instance(){
    Singleton result = sInst;
    if (result == null) {
      synchronized(Singleton.class) {
        result = sInst;
        if (result == null)
          sInst = result =
            new Singleton();
      }
    }
    return result;
  }
  ...
```

See flylib.com/books/en/2.558.1.25/1

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile


```java
class Singleton {
  private static volatile
    Singleton sInst;
```

> volatile ensures that multiple threads share the singleton instance correctly

```java
  public static Singleton instance(){
    Singleton result = sInst;
    if (result == null) {
      synchronized(Singleton.class) {
        result = sInst;
        if (result == null)
          sInst = result =
            new Singleton();
      }
    }
    return result;
  }
  ...
```

See en.wikipedia.org/wiki/Double-checked_locking#Usage_in_Java

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile

Only acquire the lock the "first time in"

```java
class Singleton {
  private static volatile
    Singleton sInst;



  public static Singleton instance(){
    Singleton result = sInst;
    if (result == null) {
      synchronized(Singleton.class) {
        result = sInst;
        if (result == null)
          sInst = result =
            new Singleton();
      }
    }
    return result;
  }
  ...
```

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile

```java
class Singleton {
  private static volatile
    Singleton sInst;



  public static Singleton instance(){
    Singleton result = sInst;
    if (result == null) {
      synchronized(Singleton.class) {
        result = sInst;
        if (result == null)
          sInst = result =
            new Singleton();
      }
    }
    return result;
  }
  ...
```

Perform "lazy initialization" only the "first time in"

See en.wikipedia.org/wiki/Lazy_initialization

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile


```java
class Singleton {
  private static volatile
    Singleton sInst;


  public static Singleton instance(){
    Singleton result = sInst;
    if (result == null) {
      synchronized(Singleton.class) {
        result = sInst;
        if (result == null)
          sInst = result =
            new Singleton();
      }
    }
    return result;
  }
...
```

volatile avoids problems with partially constructed objects

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile

Return the singleton's value

```java
class Singleton {
  private static volatile
    Singleton sInst;


  public static Singleton instance(){
    Singleton result = sInst;
    if (result == null) {
      synchronized(Singleton.class) {
        result = sInst;
        if (result == null)
          sInst = result =
            new Singleton();
      }
    }
    return result;
  }
...
```

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile

  The drawback with this approach is that it only works with Java 1.5 or later

  PROBLEM

```java
class Singleton {
  private static volatile
    Singleton sInst;


  public static Singleton instance(){
    Singleton result = sInst;
    if (result == null) {
      synchronized(Singleton.class) {
        result = sInst;
        if (result == null)
          sInst = result =
            new Singleton();
      }
    }
    return result;
  }
...
```

Safe Publication Techniques in Java

- An object can be published safely in several ways
  - Storing a reference to it into a field protected by a lock
  - Storing a reference to it in a volatile or AtomicReference

```java
class Singleton {
  private static AtomicReference sInst
    = new AtomicReference(null);



  public static Singleton instance(){
    Singleton sing = sInst.get();




    if (sing == null) {
      sing = new Singleton();
      if (!sInst.compareAndSet
                    (null, sing))
        sing = sInst.get();
    }
    return sing;
  }
  ...
```

See day-to-day-stuff.blogspot.com/2011/06/lock-less-singleton-pattern.html

Safe Publication Techniques in Java

- An object can be published safely in several ways
  - Storing a reference to it into a field protected by a lock
  - Storing a reference to it in a volatile or AtomicReference

```java
class Singleton {
  private static AtomicReference sInst
    = new AtomicReference(null);
```

Create an AtomicReference

```java
  public static Singleton instance(){
    Singleton sing = sInst.get();



    if (sing == null) {
      sing = new Singleton();
      if (!sInst.compareAndSet
                    (null, sing))
        sing = sInst.get();
    }
    return sing;
  }
  ...
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicReference.html

Safe Publication Techniques in Java

- An object can be published safely in several ways
  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile or AtomicReference

```java
class Singleton {
  private static AtomicReference sInst
    = new AtomicReference(null);


  public static Singleton instance(){
    Singleton sing = sInst.get();
```

> Get Singleton value & check for null

```java
  if (sing == null) {
    sing = new Singleton();
    if (!sInst.compareAndSet
                  (null, sing))
      sing = sInst.get();
  }
  return sing;
}
...
```

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile or AtomicReference

Allocate Singleton & atomically CAS with sInst

```java
class Singleton {
  private static AtomicReference sInst
    = new AtomicReference(null);



  public static Singleton instance(){
    Singleton sing = sInst.get();




    if (sing == null) {
      sing = new Singleton();
      if (!sInst.compareAndSet
                    (null, sing))
        sing = sInst.get();
    }
    return sing;
  }
...
```

Safe Publication Techniques in Java

- An object can be published safely in several ways
  - Storing a reference to it into a field protected by a lock
  - Storing a reference to it in a volatile or AtomicReference

```java
class Singleton {
  private static AtomicReference sInst
    = new AtomicReference(null);



  public static Singleton instance(){
    Singleton sing = sInst.get();




    if (sing == null) {
      sing = new Singleton();
      if (!sInst.compareAndSet
                    (null, sing))
        sing = sInst.get();
    }
    return sing;
  }
...
```

> Update this local value if sInst was already set

Safe Publication Techniques in Java

- An object can be published safely in several ways
  - Storing a reference to it into a field protected by a lock
  - Storing a reference to it in a volatile or AtomicReference

```java
class Singleton {
  private static AtomicReference sInst
    = new AtomicReference(null);



  public static Singleton instance(){
    Singleton sing = sInst.get();




    if (sing == null) {
      sing = new Singleton();
      if (!sInst.compareAndSet
                    (null, sing))
        sing = sInst.get();
    }
    return sing;
  }
...
```

Return the singleton's value

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile or AtomicReference

The drawback is that singleton's constructor can be called multiple times..

PROBLEM

```java
class Singleton {
  private static AtomicReference sInst
    = new AtomicReference(null);

  public static Singleton instance(){
    Singleton sing = sInst.get();

    if (sing == null) {
     sing = new Singleton();
     if (!sInst.compareAndSet
                    (null, sing))
       sing = sInst.get();
    }
    return sing;
  }
...
```

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock
  - Storing a reference to it in a volatile or AtomicReference

- Initializing an object reference from a static initializer

```java
class Singleton {
  private Singleton() {}

  private static class LazyHolder {
    private static final
      Singleton sInst =
        new Singleton();
  }

  public static Singleton instance(){
    return LazyHolder.sInst;
  }
}
```

> This idiom relies on the initialization phase of execution within the Java execution environment (e.g., JVM)

See en.wikipedia.org/wiki/Initialization-on-demand_holder_idiom

Safe Publication Techniques in Java

- An object can be published safely in several ways
  - Storing a reference to it into a field protected by a lock
  - Storing a reference to it in a volatile or AtomicReference
- Initializing an object reference from a static initializer

```java
class Singleton {
  private Singleton() {}

  private static class LazyHolder {
    private static final
      Singleton sInst =
        new Singleton();
  }

  public static Singleton instance(){
    return LazyHolder.sInst;
  }
}
```

LazyHolder is only initialized when the static method instance is invoked on the class Singleton, which triggers the JVM to load & initialize the LazyHolder class

See en.wikipedia.org/wiki/Initialization-on-demand_holder_idiom

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile or AtomicReference

  - Initializing an object reference from a static initializer

- Storing a reference to it into a final field

```
class A {
  long mNotFinal = 1;
  final long mFinal = 2;
  ...
}
...
```

See www.ibm.com/developerworks/library/j-jtp1029

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile or AtomicReference

  - Initializing an object reference from a static initializer

  - Storing a reference to it into a final field

    - Final fields can be safely accessed without some form of synchronization

```
class A {
  long mNotFinal = 1;
  final long mFinal = 2;
  ...
}


// Thread T1
A a = new A();


// Thread T2
long l1 = a.mFinal;
long l2 = a.mNotFinal;
```

> mFinal is guaranteed to be initialized by the time thread T_2 gets a reference to object a

See www.ibm.com/developerworks/library/j-jtp1029/index.html#heading6

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile or AtomicReference

  - Initializing an object reference from a static initializer

  - Storing a reference to it into a final field

    - Final fields can be safely accessed without some form of synchronization

```
class A {
  long mNotFinal = 1;
  final long mFinal = 2;
  ...
}


// Thread T1
A a = new A();


// Thread T2
long l1 = a.mFinal;
long l2 = a.mNotFinal;
```

mNotFinal is not guaranteed to be initialized by the time thread T_2 gets a reference to object a

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile or AtomicReference

  - Initializing an object reference from a static initializer

- Storing a reference to it into a final field

  - Final fields can be safely accessed without some form of synchronization

    - Immutable objects in Java contain only final fields and/or only accessor methods

```java
final class String {
  private final char value[];
  ...

  public String(String s) {
    value = s;
    ...
  }

  public int length() {
    return value.length;
  }
  ...
}
```

See docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html

Safe Publication Techniques in Java

- An object can be published safely in several ways
  - Storing a reference to it into a field protected by a lock
  - Storing a reference to it in a volatile or AtomicReference
  - Initializing an object reference from a static initializer
- Storing a reference to it into a final field
  - Final fields can be safely accessed without some form of synchronization
    - Immutable objects in Java contain only final fields and/ or only accessor methods

```java
final class String {
  private final char value[];
  ...

  public String(String s) {
    value = s;
    ...
  }

  public int length() {
    return value.length;
  }
  ...
}
```

See www.programcreek.com/2013/04/why-string-is-immutable-in-java

Safe Publication Techniques in Java

- An object can be published safely in several ways
  - Storing a reference to it into a field protected by a lock
  - Storing a reference to it in a volatile or AtomicReference
  - Initializing an object reference from a static initializer
  - Storing a reference to it into a final field
    - Final fields can be safely accessed without some form of synchronization
  - If a final field refers to a mutable object, synchronization is needed to access the *state* of the referenced object

```java
class A {
  final String[] QBs = new String[]{
    "Brady", "Favre", "Newton", ...
  };
  ...
};

A a = new A();

// Thread T1
synchronized(m)
{ a.QBs[1] = "Manning"; }

// Thread T2
synchronized(m)
{ a.QBs[1] = "Montana"; }
```

See www.ibm.com/developerworks/library/j-jtp1029/index.html#limitations

Safe Publication Techniques in Java

- An object can be published safely in several ways
  - Storing a reference to it into a field protected by a lock
  - Storing a reference to it in a volatile or AtomicReference
  - Initializing an object reference from a static initializer
  - Storing a reference to it into a final field
    - Final fields can be safely accessed without some form of synchronization
  - If a final field refers to a mutable object, synchronization is needed to access the *state* of the referenced object

```
class A {
  final String[] QBs = new String[]{
    "Brady", "Favre", "Newton", ...
  };
  ...
};
```

QBs is final, but its contents are mutable

```
A a = new A();

// Thread T1
synchronized(m)
{ a.QBs[1] = "Manning"; }

// Thread T2
synchronized(m)
{ a.QBs[1] = "Montana"; }
```

Safe Publication Techniques in Java

- An object can be published safely in several ways

  - Storing a reference to it into a field protected by a lock

  - Storing a reference to it in a volatile or AtomicReference

  - Initializing an object reference from a static initializer

- Storing a reference to it into a final field

  - Final fields can be safely accessed without some form of synchronization

- If a final field refers to a mutable object, synchronization is needed to access the *state* of the referenced object

```
class A {
  final String[] QBs = new String[]{
    "Brady", "Favre", "Newton", ...
  };
  ...
};


A a = new A();

// Thread T1
synchronized(m)
{ a.QBs[1] = "Manning"; }

// Thread T2
synchronized(m)
{ a.QBs[1] = "Montana"; }
```

Access to QBs contents must be synchronized

End of Safe Publication in Java: Techniques