

Analysis of Software Artifacts

Checking Typestate with Plural

Jonathan Aldrich

slides and research developed with Kevin Bierhoff and Nels Beckman

Overview (Java Platform SE 6) - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://java.sun.com/javase/6/docs/api/

Getting Started Latest Headlines

Java™ Platform Standard Ed. 6

All Classes

Packages

java.applet

java.awt

java.awt.color

java.awt.datatransfer

java.awt.dnd

java.awt.event

java.awt.font

All Classes

AbstractAction

AbstractAnnotatedValueVisitor6

AbstractBorder

AbstractButton

AbstractCellEditor

AbstractCollection

AbstractColorChooserPanel

AbstractDocument

AbstractDocument.AttributeContext

AbstractDocument.Content

AbstractDocument.ElementEdit

AbstractElementVisitor6

AbstractExecutorService

AbstractInterruptibleChannel

AbstractLayoutCache

AbstractLayoutCache.NodeDimensions

AbstractList

AbstractListModel

AbstractMap

AbstractMap.SimpleEntry

AbstractMap.SimpleImmutableEntry

AbstractMarshallerImpl

AbstractMethodError

AbstractOwnableSynchronizer

AbstractPreferences

AbstractProcessor

AbstractQueue

AbstractQueuedLongSynchronizer

AbstractQueuedSynchronizer

AbstractScriptEngine

AbstractSelectableChannel

AbstractSelectionKey

AbstractSelector

AbstractSequentialList

AbstractSet

Overview Package Class Use Tree Deprecated Index Help

PREV NEXT

FRAMES NO FRAMES

Java™ Platform Standard Ed. 6

This document is the API specification for version 6 of the Java™ Platform, Standard Edition.

See: [Description](#)

Packages

java.applet

Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.

java.awt

Contains all of the classes for creating user interfaces and for painting graphics and images.

java.awt.color

Provides classes for color spaces.

java.awt.datatransfer

Provides interfaces and classes for transferring data between and within applications.

java.awt.dnd

Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.

java.awt.event

Provides interfaces and classes for dealing with different types of events fired by AWT components.

java.awt.font

Provides classes and interface relating to fonts.

java.awt.geom

Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.

java.awt.im

Provides classes and interfaces for the input method framework.

java.awt.im.spi

Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.

java.awt.image

Provides classes for creating and modifying images.

java.awt.image.renderable

Provides classes and interfaces for producing rendering-independent images.

java.awt.print

Provides classes and interfaces for a general printing API.

java.beans

Contains classes related to developing *beans* -- components based on the JavaBeans™ architecture.

java.beans.beancontext

Provides classes and interfaces relating to bean context.

java.io

Provides for system input and output through data streams, serialization and the file system.

java.lang

Provides classes that are fundamental to the design of the Java programming language.

java.lang.annotation

Provides library support for the Java programming language annotation facility.

java.lang.instrument

Provides services that allow Java programming language agents to instrument programs running on the JVM.

java.lang.management

Provides the management interface for monitoring and management of the Java virtual machine as well as the operating

Done

Overview (Java Platform SE 6) - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://java.sun.com/javase/6/docs/api/

Getting Started Latest Headlines

Java™ Platform Standard Ed. 6

All Classes

Packages

java.applet

java.awt

java.awt.color

java.awt.datatransfer

java.awt.dnd

java.awt.event

java.awt.font

All Classes

AbstractAction

AbstractAnnotationV

AbstractBorder

AbstractButton

AbstractCellEditor

AbstractCollection

AbstractColorChoos

AbstractDocument

AbstractDocumentA

AbstractDocumentC

AbstractDocumentE

AbstractElementVisib

AbstractExecutorServ

AbstractInterruptibleChannel

AbstractLayoutCache

AbstractLayoutCache

AbstractList

AbstractListModel

AbstractMap

AbstractMap.Simple

AbstractMap.Simple

AbstractMarshaller

AbstractMethodError

AbstractOwnableSyn

AbstractPreferences

AbstractProcessor

AbstractQueue

AbstractQueuedLon

AbstractQueuedSyn

AbstractScriptEngine

AbstractSelectableC

AbstractSelectableKe

AbstractSelector

AbstractSequentialList

AbstractSet

Overview Package Class Use Tree Deprecated Index Help

PREV NEXT

FRAMES NO FRAMES

Java™ Platform, Standard Edition 6

API Specification

This document is the API specification for version 6 of the Java™ Platform, Standard Edition.

Provides classes and interface relating to fonts.

Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.

Provides a management interface for monitoring and management of the Java virtual machine as well as the operating

java.lang.management

Done

start

Inbox for k...

Java - Per...

Microsoft P...

Overview (...)

Adobe Rea...

94%

11:05 AM

Software development today heavily relies on re-usable APIs

APIs in this lecture include libraries and frameworks

Many APIs require clients to follow protocols that allow API implementations to work correctly

Protocols in Plural are defined based on timesteps

Protocols constrain temporal ordering of events

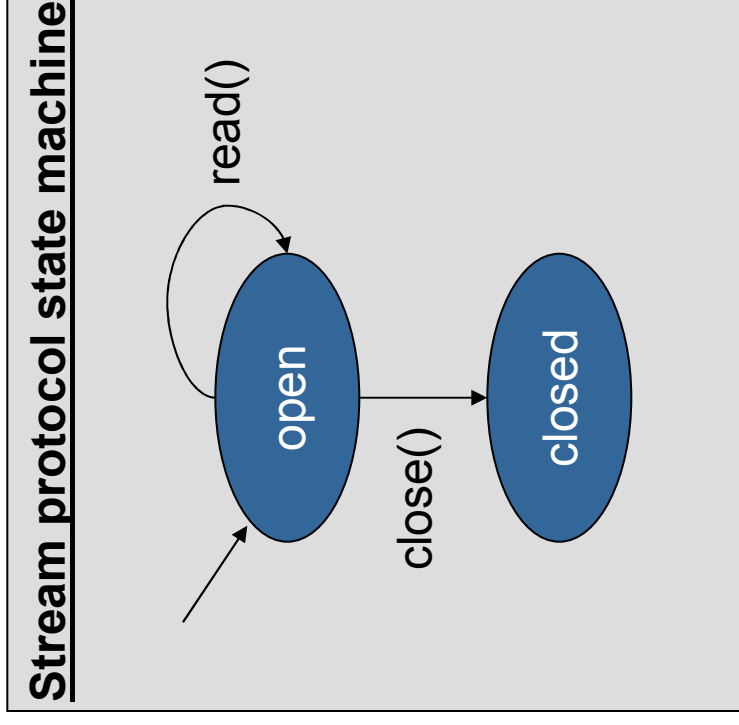
- Protocols define restrictions on which methods can be called when
- Clients have to follow protocols in order to avoid runtime errors
- Protocols can often be modeled as **state machines**

Streams can be read until they're closed

```
public interface InputStream {  
    public int read();  
    public void close();  
}
```

Stream sample client

```
InputStream f = new FileInputStream(...);  
int c = f.read(); // read first character  
while(c >= 0) {  
    // do something with c  
    c = f.read(); // read next character  
}  
f.close();
```



Sockets go through a well-defined sequence of states

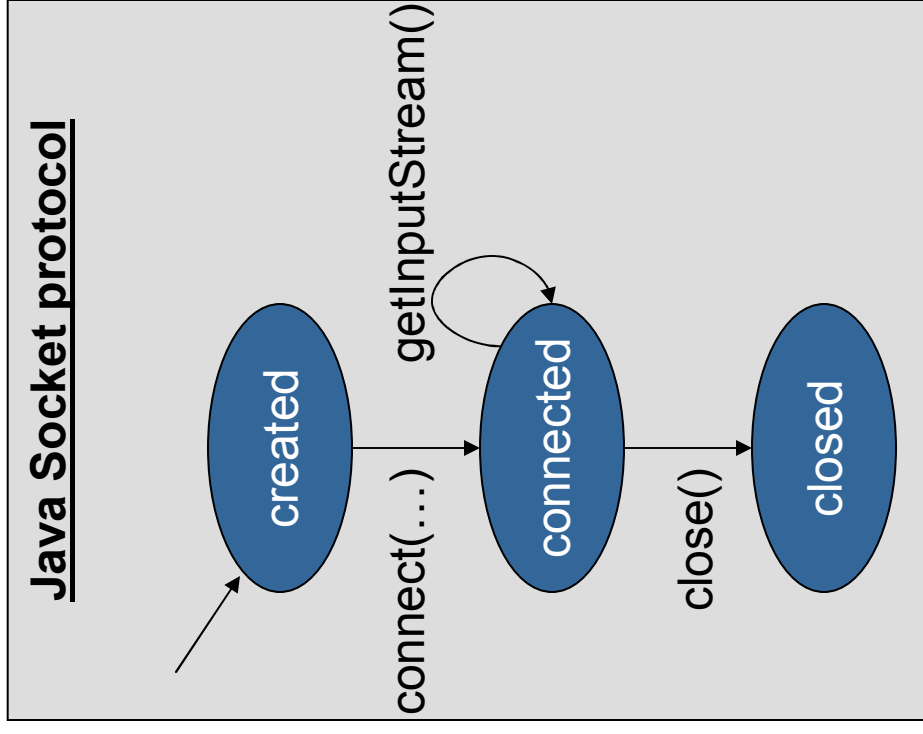
```
@States({"created", "connected", "closed"})
public class Socket {
    @Creates("created")
    public Socket()

    @ChangesState("created", "connected")
    public void connect(...)

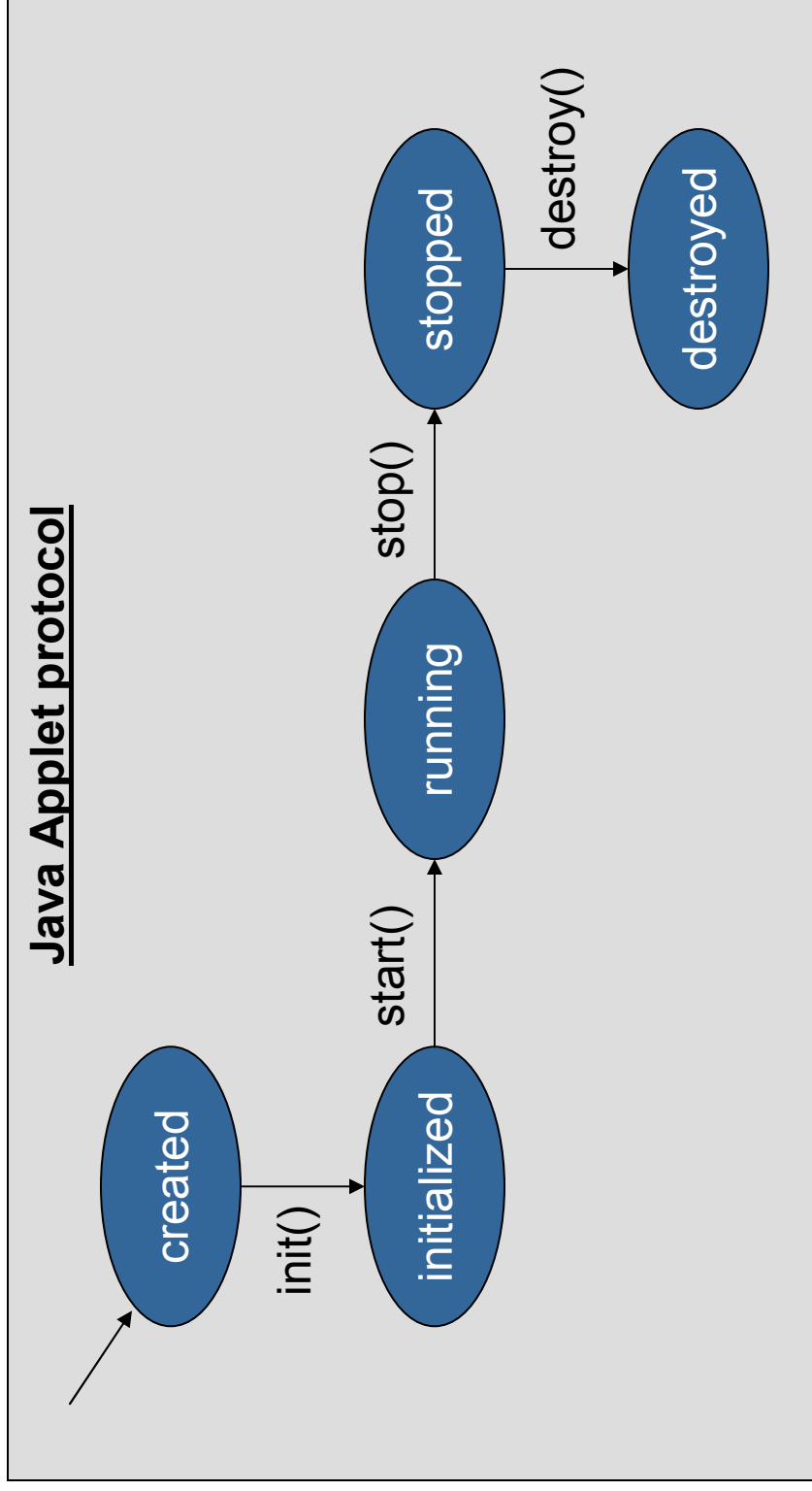
    @InState("connected")
    public InputStream getInputStream()

    @InState("connected")
    public OutputStream getOutputStream()

    @ChangesState("connected", "closed")
    public void close();
}
```

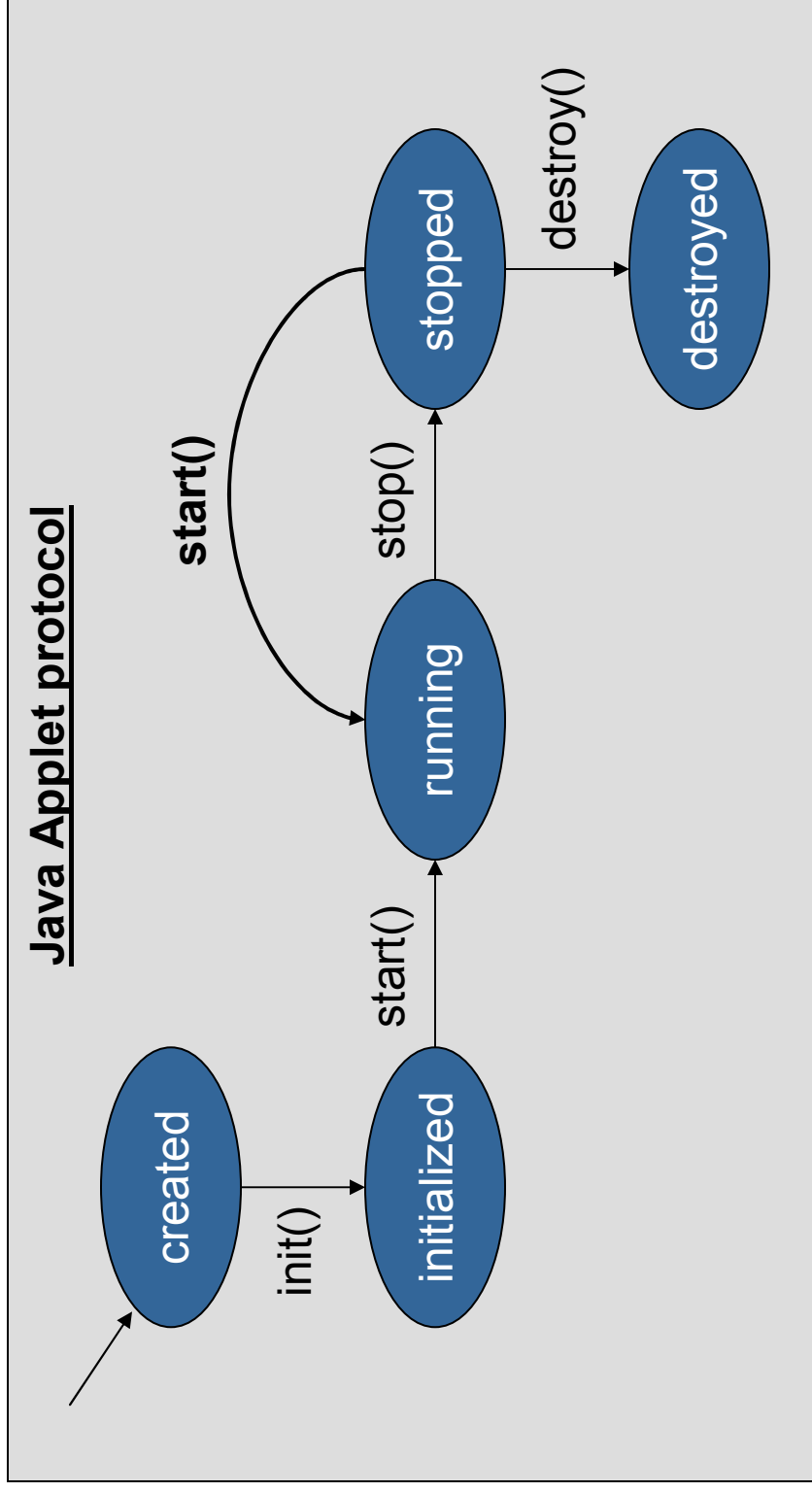


Java Applets have a funny back edge



Example based on: G. Fairbanks, D. Garlan & W. Scherlis. Design fragments make using frameworks easier. In *Proceedings of OOPSLA'06*, pp. 75-88. ACM Press, 2006.

Java Applets have a funny back edge



Example based on: G. Fairbanks, D. Garlan & W. Scherlis. Design fragments make using frameworks easier. In *Proceedings of OOPSLA'06*, pp. 75-88. ACM Press, 2006.

APIs are hard to implement and use

API users (clients)

- Can take a long time to get right the first time
- Extensive testing to make sure everything works
- Code modifications may introduce new errors
- Hard to guarantee that protocol is followed on all paths through the program

API implementers

- Documentation consistent with actual code
- Consistent runtime tests to protect against misuse
- Shared objects might be modified unexpectedly
- Reentrancy
- Unexpected overriding and open recursion

Implementation of one API will often use other APIs

Java - SimplePermissionTest.java - File Edit Source Refactor Navigate Search Project Crystal Run Window Help

Package Explorer: Java Resource

- Asst1
- BoardGame
- permission-test
 - edu.cmu.cs.plural.test
 - ArrayIterator.java
 - SimplePermissionTest.java
 - StreamInterface.java
 - JRE System Library [jdk1.5.
 - plural-annot.jar
 - protocol-test
 - Test

SimplePermissionTest.java

```
package edu.cmu.cs.plural.test;

import edu.cmu.cs.plural.annot.Full;
import edu.cmu.cs.plural.annot.Share;

public class SimplePermissionTest {

    public static void simpleTest() {
        StreamInterface s = new StreamInterface();
        while (s.available() > 0) {
            s.read();
            readBoth(s, s);
            s.close();
            s.read();
        }

        public static int readBoth(
            @Full("open") StreamInterface s1,
            @Share("open") StreamInterface s2) {
            return Math.max(s1.read(), s2.read());
        }
    }
}
```

StreamInterface.java

```
package edu.cmu.cs.plural.test;

import edu.cmu.cs.plural.annot.Full;

public class StreamInterface {

    @Unique(ensures = "open")
    public StreamInterface() {
        super();
    }

    @Full("open")
    public int read() {
        return -1;
    }

    @Pure("open")
    public int available() {
        return 0;
    }

    @Full(requires = "open", ensures = "closed")
    public void close() {
        return;
    }
}
```

Annotations: SimplePermissionTest.java, SimplePermissionTest.java, SimplePermissionTest.java

Errors: [PermissionAnalysis]: Need FULL(open) in open but have Permissions=[SHARE(open) in open], [PermissionAnalysis]: Need FULL(open) in open but have Permissions=[UNIQUE(alive) in closed], [PermissionAnalysis]: Need SHARE(open) in open but have Permissions=[PURE(open) in open]

Automatically check
code against protocols

API designers specify
API protocols

Interactive protocol violation warnings

with Plural

Agenda

- Motivation: APIs and protocols



Protocol checking example: Streams



- States
- Function side effects
- UML Statecharts and Permissions
- Checking challenges: Iterators
 - Dynamic tests
 - Aliasing and inter-object state dependencies
- Concurrent protocols: Queue
 - Proper synchronization
 - Implementation-side checking

TOOL DEMONSTRATION

Tracking Protocols

```
@States({"open", "closed"})
public class StreamProtocol {
    @Perm(ensures = "unique(this!fr) in open")
    public StreamProtocol() {
        super();
    }
}
```

```
public static void streamTest() {
```

```
    StreamProtocol s = new StreamProtocol();
```

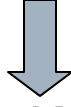
```
    while (s.available() > 0)
```

```
        dangerousProcess(s);
```

```
    s.close();
```

```
    s.read();
```

```
}
```



Object creation—initial
state is open

Tracking Protocols

```
@States({"open", "closed"})  
public class StreamProtocol {
```

```
    @Pure("open")  
    public int available() {  
        return 0;  
    }
```

```
    public static void
```

```
        test() {
```

```
        StreamProtocol s = new StreamProtocol();
```

```
        while (s.available() > 0) {  
            dangerousProcess(s);
```

```
            s.close();
```

```
            s.read();
```

```
        }
```

available requires the “open” state
but doesn’t change it
(that’s what @Pure means)

Tracking Protocols

```
private static void
dangerousProcess(@Full(requires = "open") StreamProtocol s) {
    System.out.println(s.read());
    s.close();
}
```

```
public static void main() {
    StreamProtocol s = new StreamProtocol();
    while (s.available() > 0)
        dangerousProcess(s);
    s.close();
    s.read();
}
```

dangerousProcess might close the object! State is now alive.

Tracking Protocols

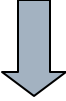
```
@States({"open", "closed"})  
public class StreamProtocol {
```

```
    @Pure("open")  
    public int available() {  
        return 0;  
    }
```

```
    public static void
```

```
        test() {
```

```
        StreamProtocol s = new StreamProtocol();
```

```
        while (s.available() > 0)  available requires the "open" state  
            dangerousProcess(s); but state is in alive - error
```

```
        s.close();
```

```
        s.read();
```

```
    }
```

Tracking Protocols

```
@States({"open", "closed"})
public class StreamProtocol {

    @Full(requires = "open", ensures = "closed", fieldAccess = true)
    public void close() {
        ...
    }
}
```

```
public static void test() {
    StreamProtocol s = new StreamProtocol();
    while (s.available() > 0)
        dangerousProcess(s);
    s.close();
    s.read();
}
```

close also requires the “open” state
(and changes it to close)
but state is in alive - error

Tracking Protocols

```
@States({"open", "closed"})
public class StreamProtocol {
    @Full("open")
    public int read() {
        ...
    }
}
```

```
public static void test() {
    StreamProtocol s = new StreamProtocol();
    while (s.available() > 0)
        dangerousProcess(s);
    s.close();
    s.read();
}
```

if we ignore the dangerousProcess error,
we still have a problem here because the
state is closed but read() requires “open”

Side Effects

- Sometimes desirable
 - `close()` transitions from open to close state
- Sometimes not
 - `dangerousProcess` might close the stream

- Declare effects with function annotations

```
@Full(requires = "open", ensures = "closed", ...)
public void close() {
}
```

No postcondition—
could do anything!

private static void

```
dangerousProcess(@Full(requires = "open")
StreamProtocol s) {
    System.out.println(s.read());
    s.close();
}
```

Not saying
requires/ensures means
guaranteed to stay in open

```
public static void
process(@Full("open") StreamProtocol s) {
    System.out.println(s.read());
}
```

Agenda

- Motivation: APIs and protocols
 - Protocol checking example: Streams
 - States
 - Function side effects
- ➔ UML Statecharts and Permissions ➔
- Checking challenges: Iterators
 - Dynamic tests
 - Aliasing and inter-object state dependencies
 - Concurrent protocols: Queue
 - Proper synchronization
 - Implementation-side checking

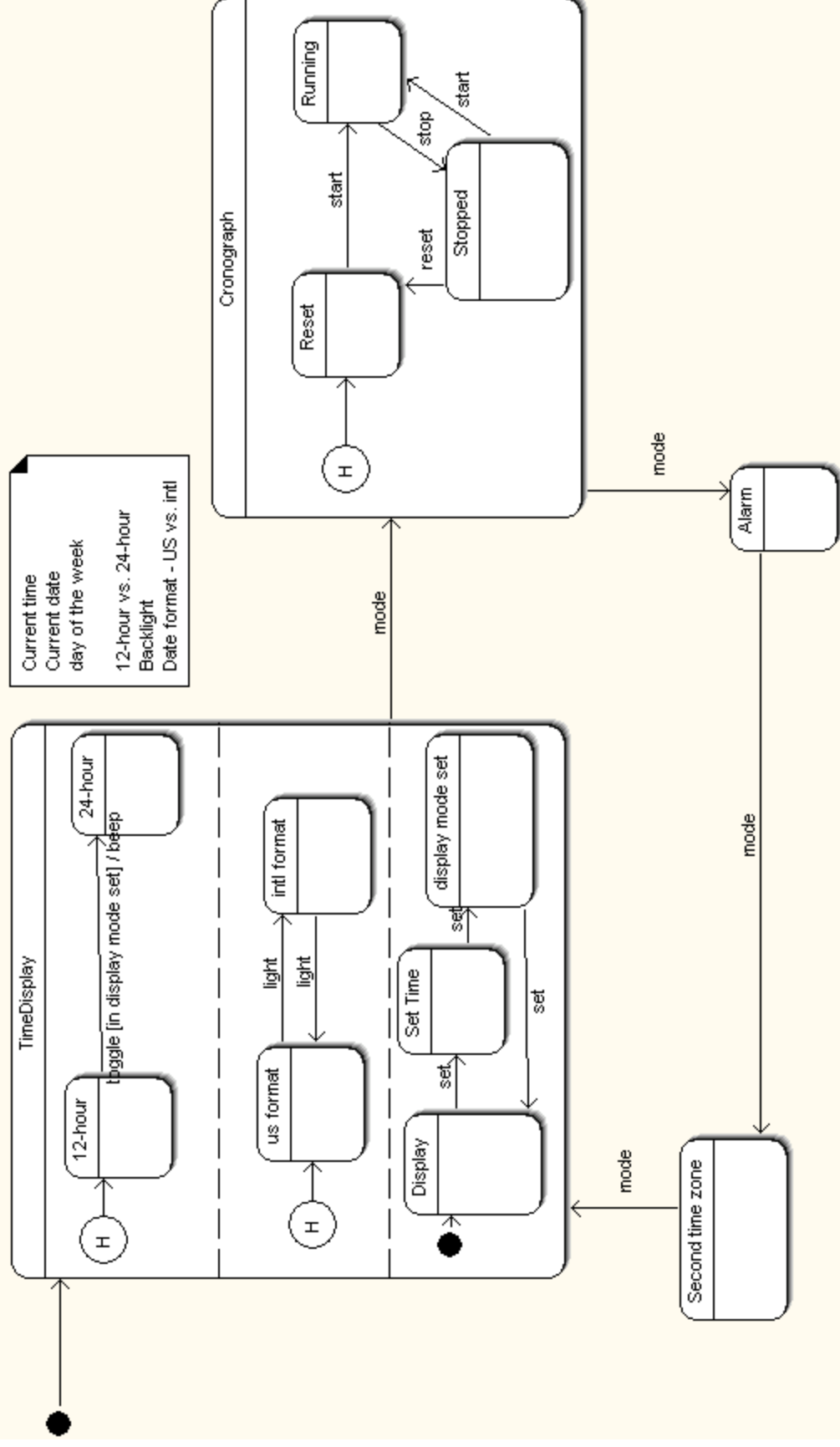
Protocol documentation...

- Protocols are informally documented
 - Example: `java.io.InputStream`
 - Detailed Javadoc for every method
 - Example: `java.net.Socket`
 - Exceptions describe when methods *cannot* be called
- Not always complete and precise

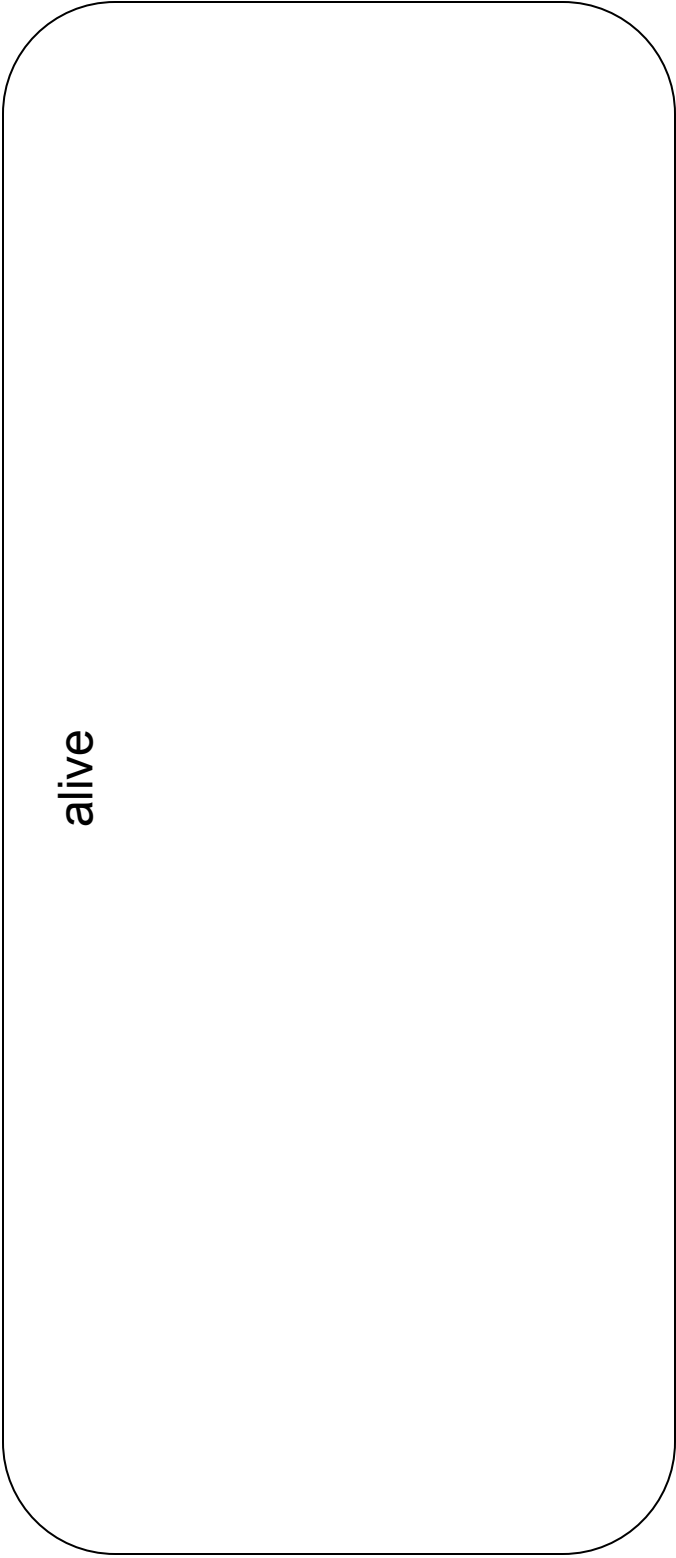
Statechart Diagrams

- Shows how system reacts to events
 - States
 - Transitions triggered by events
 - Initial and final states
 - Maybe no final states!
 - Nested state machines
- Especially useful for reactive control systems
- Vs. sequence diagram
 - Exhaustive coverage vs. one trace
 - Events vs. calls

Watch Statechart

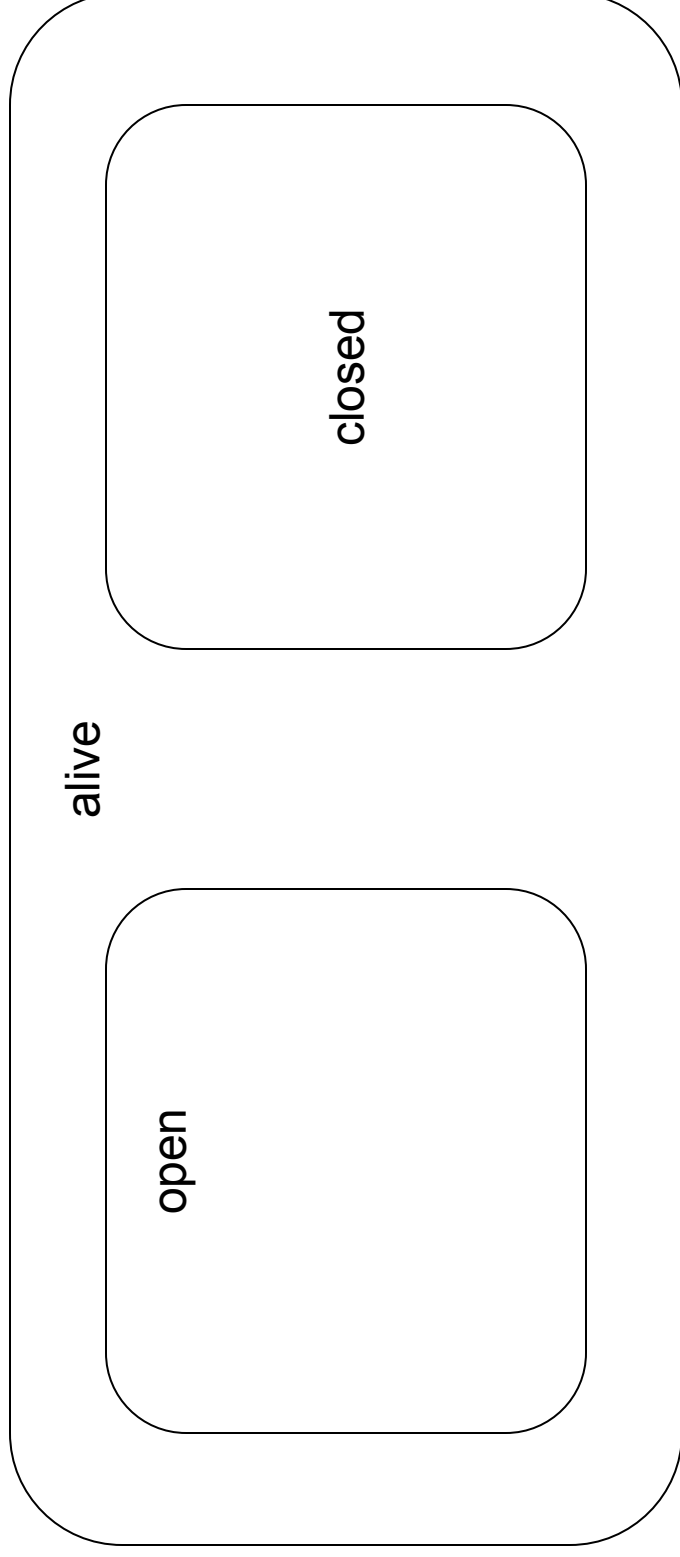


Stream Typestate Hierarchy

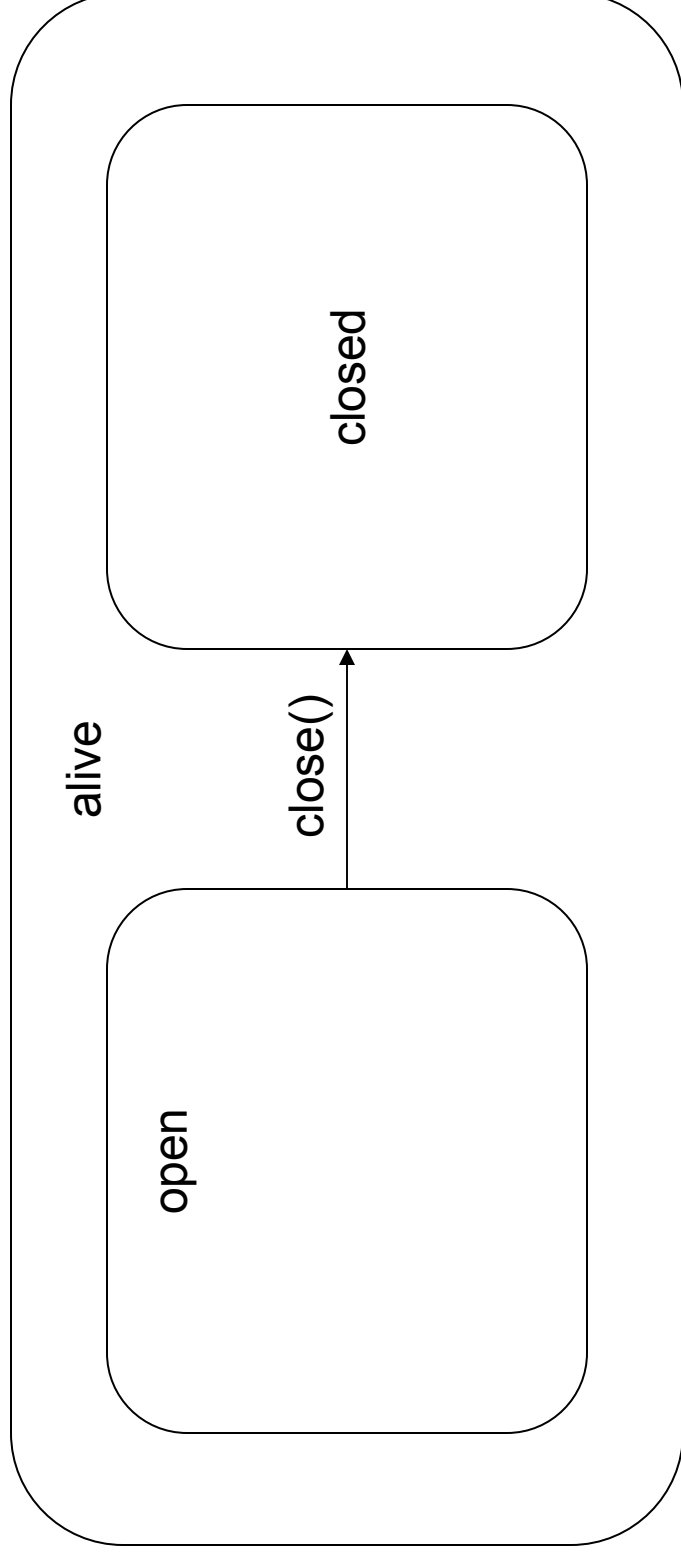


alive

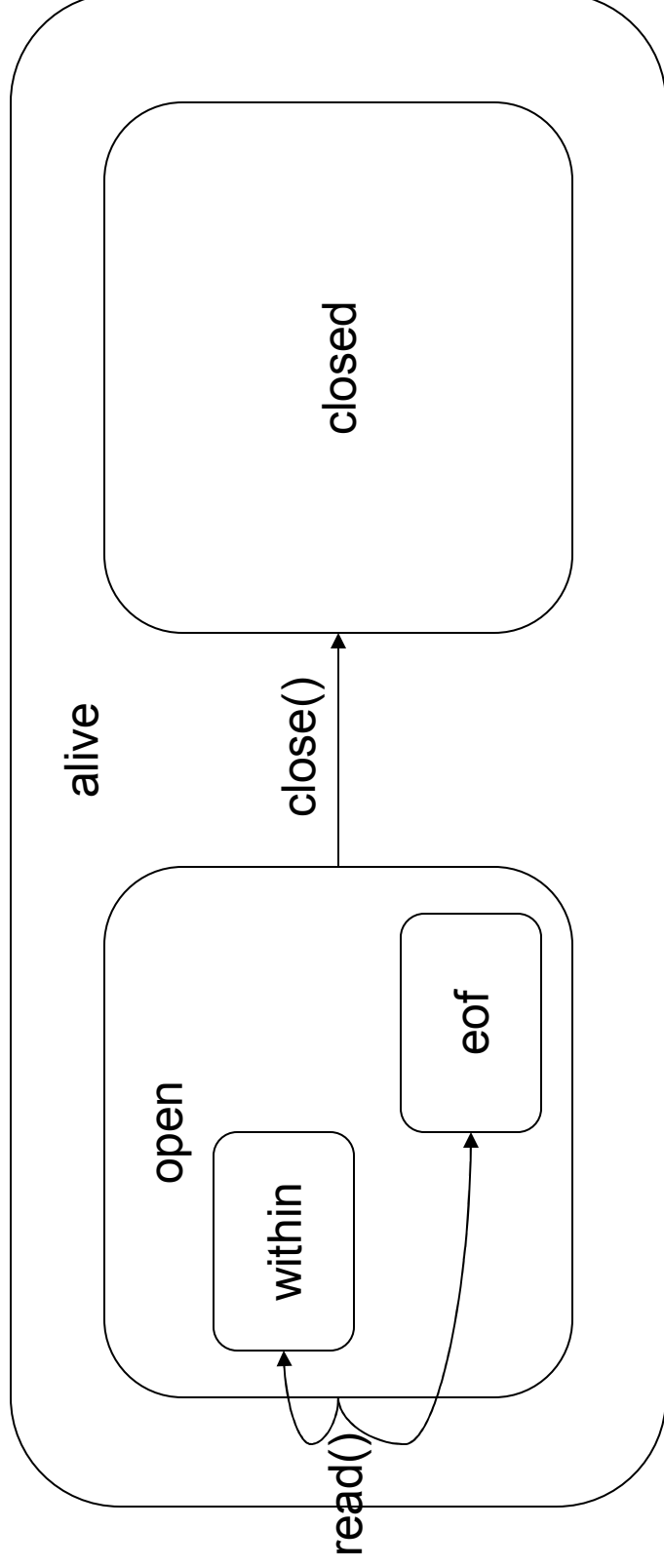
Stream Typestate Hierarchy



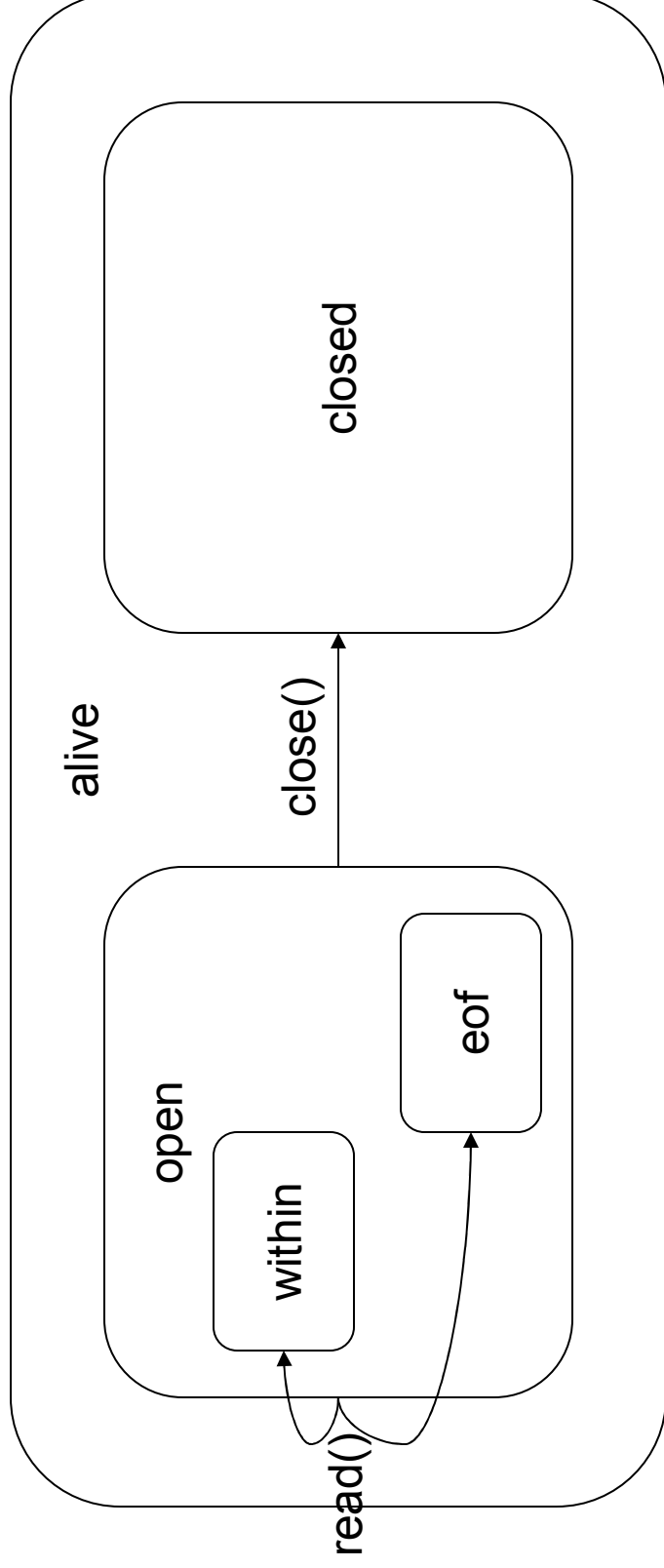
Stream Typestate Hierarchy



Stream Typestate Hierarchy



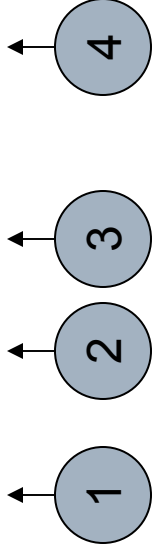
Stream Typestate Hierarchy



Subtypes can further refine existing states

Access Permissions = state + aliasing

Example: share(s, $\frac{1}{2}$, open) in within



- 1 What kinds of references exist?
- 2 How many references exist?
- 3 What state is guaranteed?
- 4 What do we know about the object's state at a given point?

Access Permission Taxonomy

Example: share(s), where s is a program variable

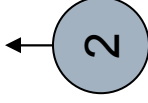
1

What kinds of references exist?

Current reference	
Other references	Read/write
None	unique
Read/write	share
Read-only	pure
	immutable

Fractional Permissions

Example: share(s, $\frac{1}{2}$)



How many references exist?

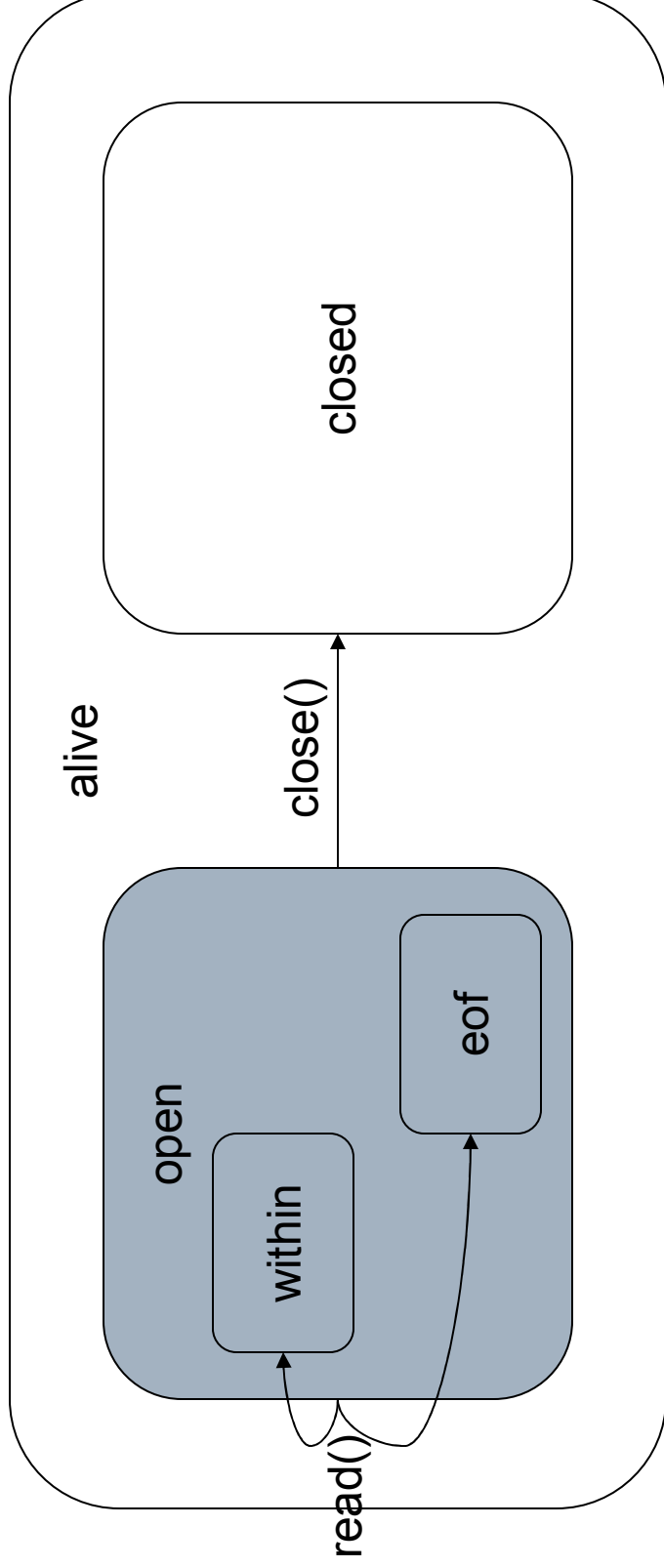
- Start with a fraction of 1
- Each time a reference is copied, split the fraction in half
 - To be precise, $\frac{1}{2}$ doesn't mean there are exactly 2 references
 - Rather, we need to assemble another $\frac{1}{2}$ permission to make a whole
- Recombine by adding fractions
 - When the fraction is 1 again, you have recovered a **unique** permission
 - Which can be converted to **full**, **pure**, **immutable**, or another **share**
 - Can also change state guarantee (next slide)

State Guarantees

Example: `share(s, 1/2, open)`

3

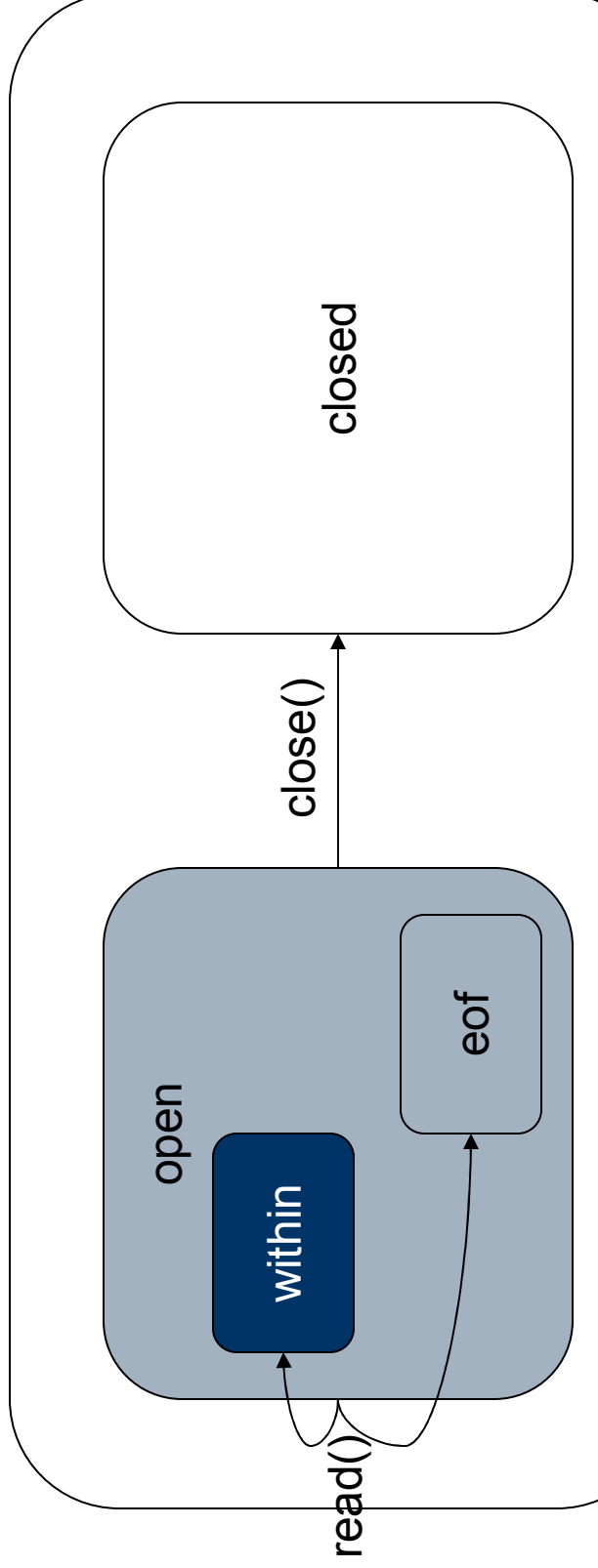
What state is guaranteed?



State Information

Example: `share(s, 1/2, open)` in within

4
What do we know about the object's state?



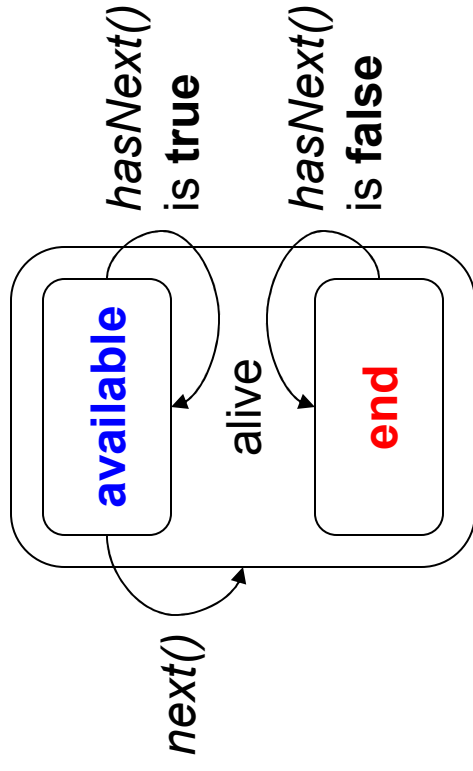
State information changes with every operation

Agenda

- Motivation: APIs and protocols
 - Protocol checking example: Streams
 - States
 - Function side effects
 - UML Statecharts and Permissions
-  Checking challenges: Iterators 
- Dynamic tests
 - Aliasing and inter-object state dependencies
- Concurrent protocols: Queue
 - Proper synchronization
 - Implementation-side checking

Read-only iterator protocol

- Iterators return all collection elements in sequence
- Object `next()`
 - Returns next element in sequence *if one is available*
- **boolean** `hasNext()`
 - Tests if there is another element **available**
- **Challenges**
 - Nondeterministic transitions
 - Dynamic state tests
 - Tracking which collection we iterate over



Moreover, collection must not be modified while iteration is in progress (“concurrent modification”)

Precisely capturing iterator protocol

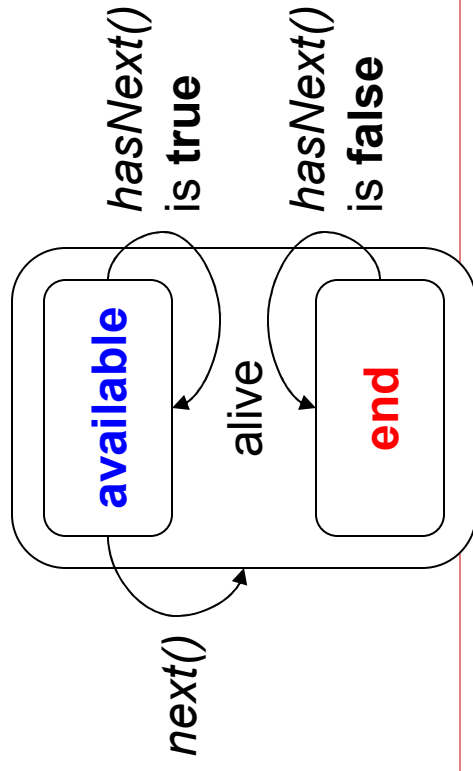
```
@Full(requires="available", fieldAccess = true)
public T next() {
    return array[pos++];
}
```

Requires the available state, but don't know the state after the call

```
@Pure(fieldAccess = true)
@TrueIndicates("available")
@FalseIndicates("end")
public boolean hasNext() {
    return pos < array.length;
}
```

Dynamic state test: result value paired with state information

Current reference	
Other references	Read/write
None	unique
Read/write	share
Read-only	full
	pure
	immutable



Fractions for splitting and joining permissions

$$\text{unique}(x, 1) \iff \text{full}(x, 1/2) \otimes \text{pure}(x, 1/2)$$

Other references	Current reference	
	Read/write	Read-only
	unique	---
	share	pure
	full	immutable

Fractions for splitting and joining permissions

$\text{unique}(x, 1)$	$\Leftarrow \Rightarrow$	$\text{full}(x, 1/2) \otimes \text{pure}(x, 1/2)$
$\text{full}(x, k)$	$\Leftarrow \Rightarrow$	$\text{full}(x, k/2) \otimes \text{pure}(x, k/2)$
$\text{full}(x, 1/2)$	$\Leftarrow \Rightarrow$	$\text{immutable}(x, 1/4) \otimes \text{immutable}(x, 1/4)$
$\text{immutable}(x, k)$	$\Leftarrow \Rightarrow$	$\text{immutable}(x, k/2) \otimes \text{immutable}(x, k/2)$
$\text{full}(x, 1/2)$	$\Leftarrow \Rightarrow$	$\text{share}(x, 1/4) \otimes \text{share}(x, 1/4)$
$\text{share}(x, k)$	$\Leftarrow \Rightarrow$	$\text{share}(x, k/2) \otimes \text{share}(x, k/2)$
$\text{share}(x, k)$	$\Leftarrow \Rightarrow$	$\text{share}(x, k/2) \otimes \text{pure}(x, k/2)$

Other references	Current reference	
	Read/write	Read-only
None	unique	—
Read/write	share	pure
Read-only	full	immutable

Collection is immutable while iterated

```
@Param(name = "list", releasedFrom = "alive")  
public class ListIterator<T> {
```

Iterator parameterized
by immutable permission
to iterated collection

```
    @Perm(requires = "immutable(#0)", ensures = "unique(this!fr)")  
public ListIterator(@Capture(param = "list") List<T> list) { ... }  
  
    @Unique(returned = false)  
    @Release("list")  
public void dispose() { }
```

Release “captured” collection
permission upon destruction

Iterator captures
immutable permission
to collection

```
public class List<T> {
```

Immutability prevents “concurrent modifications”

Checking, Verifying, and
with FURAL

Now we can verify clients

Code	Available permissions
Collection c = new HashSet(); c.add(x);	

Now we can verify clients

Code	Available permissions
<pre>Collection c = new HashSet(); c.add(x);</pre>	<pre>unique(c)</pre>

Now we can verify clients

Code	Available permissions
Collection c = new HashSet(); c. add (x);	unique(c)
add (x): $\forall z. \text{full}(\text{this}, z) \multimap \text{full}(\text{this}, z)$	

Now we can verify clients

Code	Available permissions
Collection c = new HashSet(); c. add (x);	unique(c) unique(c) $\Rightarrow$ full(c, $\frac{1}{2}$) $\otimes$ pure(c, $\frac{1}{2}$)
add (x): $\forall z. \text{full}(\text{this}, z) \multimap \text{full}(\text{this}, z)$	

Now we can verify clients

Code	Available permissions
Collection c = new HashSet(); c.add(x);	unique(c) $\text{unique}(c) \Rightarrow \underline{\text{full}(c, \frac{1}{2})} \otimes \text{pure}(c, \frac{1}{2})$
	 $\text{add}(x): \forall z. \text{full}(\text{this}, z) \multimap \text{full}(\text{this}, z)$

Now we can verify clients

Code	Available permissions
Collection c = new HashSet(); c.add(x);	$\text{unique}(c)$ $\text{unique}(c) \Rightarrow \underline{\text{full}(c, \frac{1}{2})} \otimes \underline{\text{pure}(c, \frac{1}{2})}$ $\text{add}(x): \forall z. \text{full}(\text{this}, z) \multimap \text{full}(\text{this}, z)$ $\text{full}(c, \frac{1}{2}) \otimes \text{pure}(c, \frac{1}{2})$

Now we can verify clients

Code	Available permissions
Collection c = new HashSet(); c.add(x);	$\text{unique}(c)$
	$\text{unique}(c) \Rightarrow \text{full}(c, \frac{1}{2}) \otimes \text{pure}(c, \frac{1}{2})$
	$\text{add}(x): \forall z. \text{full}(\text{this}, z) \multimap \text{full}(\text{this}, z)$
	$\text{full}(c, \frac{1}{2}) \otimes \text{pure}(c, \frac{1}{2}) \Rightarrow \text{unique}(c)$

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<C, 1/2> it = c.iterator();	

Now we can verify clients

Code	Available permissions
Collection c = new HashSet(); c.add(x); Iterator<C, 1/2> it = c.iterator();	unique(c) unique(c)
iterator() : imm(this, 1/2) $\rightarrow$ unique(result)	

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<C, 1/2> it = c.iterator();	unique(c) $\Rightarrow$ imm(c, 1/2) $\otimes$ imm(c, 1/2)
iterator(): imm(this, 1/2) $\multimap$ unique(result)	

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<C, 1/2> it = c.iterator();	unique(c) $\Rightarrow$ <u>imm(c, 1/2)</u> $\otimes$ imm(c, 1/2)
	
	iterator(): imm(this, 1/2) $\multimap$ unique(result)

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<C, 1/2> it = c.iterator();	$\text{unique}(c) \Rightarrow \text{imm}(c, 1/2) \otimes \text{imm}(c, 1/2)$
	$\swarrow$
iterator() : imm(this, 1/2) $\multimap$ unique(result)	$\searrow$
	$\text{unique}(it) \otimes \text{imm}(c, 1/2)$

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<c, 1/2> it = c.iterator();	unique(it) $\otimes$ imm(c, 1/2)

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<c, 1/2> it = c.iterator();	
while(it.hasNext() && ++i < 10) {	unique(it) $\otimes$ imm(c, 1/2)

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<c, 1/2> it = c.iterator();	unique(it) $\otimes$ imm(c, 1/2)
while(it.hasNext() && ++i < 10) {	unique(it) in available $\otimes$ imm(c, 1/2)

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<c, 1/2> it = c.iterator();	unique(it) $\otimes$ imm(c, 1/2)
while(it.hasNext() && ++i < 10) {	unique(it) in available $\otimes$ imm(c, 1/2)
Object o = it.next();	
... }	

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<c, 1/2> it = c.iterator();	unique(it) $\otimes$ imm(c, 1/2)
while(it.hasNext() && ++i < 10) {	unique(it) in available $\otimes$ imm(c, 1/2)
Object o = it.next();	unique(it) $\otimes$ imm(c, 1/2)
... }	

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<c, 1/2> it = c.iterator();	unique(it) $\otimes$ imm(c, 1/2)
while(it.hasNext() && ++i < 10) {	unique(it) in available $\otimes$ imm(c, 1/2)
Object o = it.next();	unique(it) $\otimes$ imm(c, 1/2)
... }	unique(it) $\otimes$ imm(c, 1/2)

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<C, 1/2> it = c.iterator();	unique(it) $\otimes$ imm(c, 1/2)
while(it.hasNext() && ++i < 10) {	unique(it) in available $\otimes$ imm(c, 1/2)
Object o = it.next();	unique(it) $\otimes$ imm(c, 1/2)
... }	unique(it) $\otimes$ imm(c, 1/2)
c.remove(x);	

Now we can verify clients

Code	Available permissions
<pre>Collection c = new HashSet(); c.add(x); Iterator<C, 1/2> it = c.iterator(); while(it.hasNext() && ++i < 10) { Object o = it.next(); ... } c.remove(x);</pre>	<pre>unique(c) unique(c) unique(it) $\otimes$ imm(c, 1/2) unique(it) in available $\otimes$ imm(c, 1/2) unique(it) $\otimes$ imm(c, 1/2) unique(it) $\otimes$ imm(c, 1/2)</pre>
remove() : $\forall z. \text{full}(\text{this}, z) \multimap \text{full}(\text{this}, z)$	

Now we can verify clients

Code	Available permissions
<pre>Collection c = new HashSet(); c.add(x); Iterator<c, 1/2> it = c.iterator(); while(it.hasNext() && ++i < 10) { Object o = it.next(); ... } c.remove(x);</pre>	<pre>unique(c) unique(c) unique(it) $\otimes$ imm(c, 1/2) unique(it) in available $\otimes$ imm(c, 1/2) unique(it) $\otimes$ imm(c, 1/2) unique(it) $\otimes$ imm(c, 1/2)</pre>
remove() : $\forall z. \text{full}(\text{this}, z) \multimap \text{full}(\text{this}, z)$	“kill”: $\text{unique}(\text{this}) \multimap \text{imm}(c, 1/2)$

Now we can verify clients

Code	Available permissions
<pre>Collection c = new HashSet(); c.add(x); Iterator<c, 1/2> it = c.iterator(); while(it.hasNext() && ++i < 10) { Object o = it.next(); ... } c.remove(x);</pre>	<pre>unique(c) unique(c) unique(it) $\otimes$ imm(c, 1/2) unique(it) in available $\otimes$ imm(c, 1/2) unique(it) $\otimes$ imm(c, 1/2) unique(it) $\otimes$ imm(c, 1/2)</pre> “kill”: $\text{unique(this)} \multimap \text{imm(c, 1/2)}$ remove(): $\forall z. \text{full(this, z)} \multimap \text{full(this, z)}$

Now we can verify clients

Code	Available permissions
<pre> Collection c = new HashSet(); c.add(x); Iterator<c, 1/2> it = c.iterator(); while(it.hasNext() && ++i < 10) { Object o = it.next(); ... } c.remove(x); </pre>	unique(c) unique(c) unique(it) $\otimes$ imm(c, 1/2) unique(it) in available $\otimes$ imm(c, 1/2) unique(it) $\otimes$ imm(c, 1/2) unique(it) $\otimes$ imm(c, 1/2) “kill”: unique(this) $\multimap$ imm(c, 1/2) imm(c, 1/2) $\otimes$ imm(c, 1/2)
<pre> remove(): $\forall z. \text{full}(\text{this}, z) \multimap \text{full}(\text{this}, z)$ </pre>	

Now we can verify clients

Code	Available permissions
<pre> Collection c = new HashSet(); c.add(x); Iterator<C, 1/2> it = c.iterator(); while(it.hasNext() && ++i < 10) { Object o = it.next(); ... } c.remove(x); </pre>	unique(c) unique(c) unique(it) $\otimes$ imm(c, 1/2) unique(it) in available $\otimes$ imm(c, 1/2) unique(it) $\otimes$ imm(c, 1/2) unique(it) $\otimes$ imm(c, 1/2) “kill”: unique(this) $\multimap$ imm(c, 1/2) imm(c, 1/2) $\otimes$ imm(c, 1/2) $\Rightarrow$ unique(c)
<pre> remove(): $\forall z. \text{full}(\text{this}, z) \multimap \text{full}(\text{this}, z)$ </pre>	imm(c, 1/2) $\otimes$ imm(c, 1/2) $\Rightarrow$ unique(c)

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<c, 1/2> it = c.iterator();	unique(it) $\otimes$ imm(c, 1/2)
while(it.hasNext() && ++i < 10) {	unique(it) in available $\otimes$ imm(c, 1/2)
Object o = it.next();	unique(it) $\otimes$ imm(c, 1/2)
... }	unique(it) $\otimes$ imm(c, 1/2)
c.remove(x);	unique(c)

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<C, 1/2> it = c.iterator();	unique(it) $\otimes$ imm(c, 1/2)
while(it.hasNext() && ++i < 10) {	unique(it) in available $\otimes$ imm(c, 1/2)
Object o = it.next();	unique(it) $\otimes$ imm(c, 1/2)
... }	unique(it) $\otimes$ imm(c, 1/2)
c.remove(x);	unique(c)
if(it.hasNext()) ...	

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<C, 1/2> it = c.iterator();	unique(it) $\otimes$ imm(c, 1/2)
while(it.hasNext() && ++i < 10) {	unique(it) in available $\otimes$ imm(c, 1/2)
Object o = it.next();	unique(it) $\otimes$ imm(c, 1/2)
... }	unique(it) $\otimes$ imm(c, 1/2)
c.remove(x);	unique(c)
if(it.hasNext()) ...	ERROR: no permission for it

Now we can verify clients

Code	Available permissions
Collection c = new HashSet();	unique(c)
c.add(x);	unique(c)
Iterator<C, 1/2> it = c.iterator();	unique(it) $\otimes$ imm(c, 1/2)
while(it.hasNext() && ++i < 10) {	unique(it) in available $\otimes$ imm(c, 1/2)
Object o = it.next();	unique(it) $\otimes$ imm(c, 1/2)
... }	unique(it) $\otimes$ imm(c, 1/2)
c.remove(x);	unique(c)
if(it.hasNext()) ...	ERROR: no permission for it

Illegal “concurrent modification” prevented

TOOL DEMONSTRATION

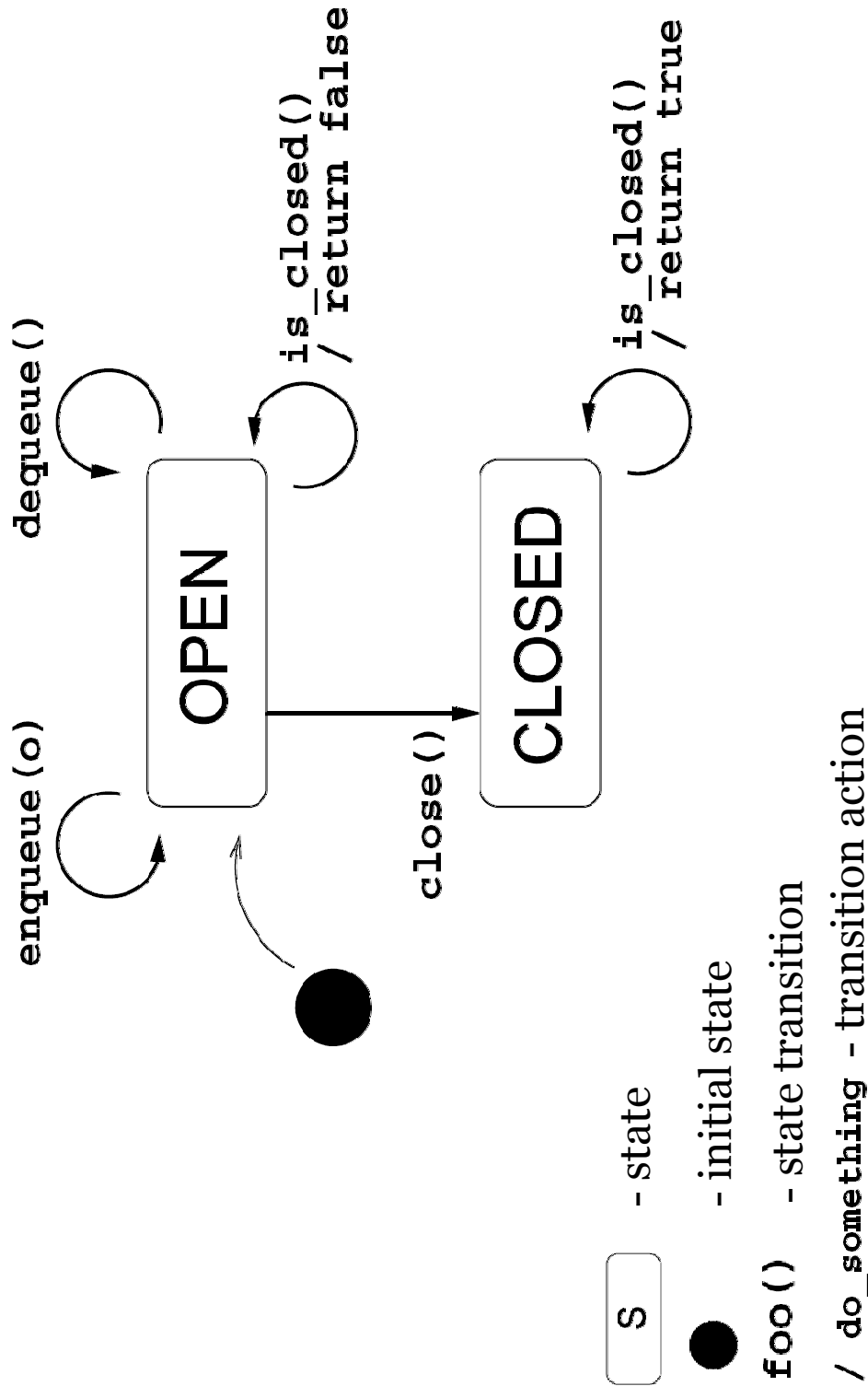
Agenda

- Motivation: APIs and protocols
 - Protocol checking example: Streams
 - States
 - Function side effects
 - UML Statecharts and Permissions
 - Checking challenges: Iterators
 - Dynamic tests
 - Aliasing and inter-object state dependencies
-  Concurrent protocols: Queue 
- Proper synchronization
 - Implementation-side checking

Example: Blocking Queue

- Thread-shared, blocking queue
 - One Producer
 - N Consumers
 - Defines Protocol
 - From Apache Axyl-Lucene Program
 - Thanks to Allen Holub!
- Meant to illustrate
 - Races on abstract state of an object
 - State transitions that are not atomic

Queue: Protocol



Race on Abstract State of Queue

```
final Blocking_queue queue = new Blocking_queue();

(new Thread() {
    @Override
    public void run() {
        while( !queue.is_closed() )
            System.out.println("Got object: "+queue.dequeue());
        // Important shut-down code...
    }).start();

for( int i=0;i<5;i++ )
    queue.enqueue("Object " + i);

queue.close();
```

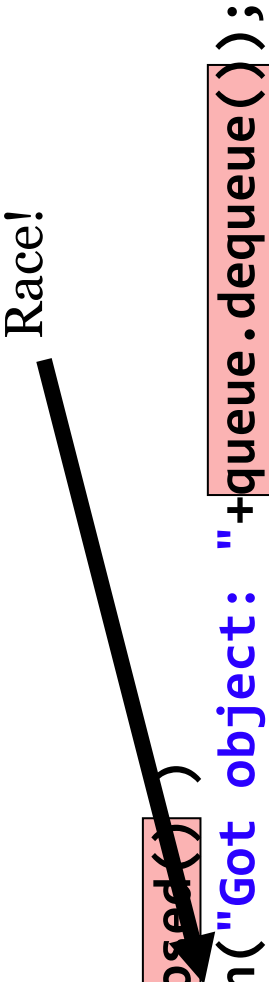
Race on Abstract State of Queue

```
final Blocking_queue queue = new Blocking_queue();

(new Thread() {
    @Override
    public void run() {
        while( !queue.is_closed() )
            System.out.println("Got object: "+queue.dequeue());
        // Important shut-down code...
    }}).start();

for( int i=0;i<5;i++ )
    queue.enqueue("Object " + i);

queue.close();
```



Race!

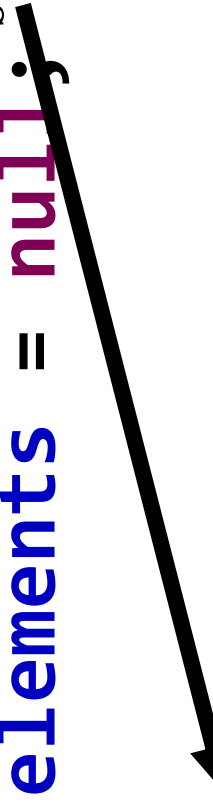
State Transition Not Atomic

```
class Blocking_queue {  
    // Class definition...  
    public void close() {  
        atomic: { elements = null; }  
        // ...  
        atomic: { closed = true; }  
    }  
}
```

State Transition Not Atomic

```
class Blocking_queue {  
    // Class definition...  
    public void close() {  
        atomic: { elements = null;  
                // ...  
                atomic: { closed = true; }  
        }  
    }  
}
```

Can be
observed in
inconsistent
state!



Verification w/o Alias Control

```
@Override
public void run() {
    while( !queue.is_closed() )
        System.out.println("Got object: " +
                           queue.dequeue());
    // Important shut-down code...
}
```

Verification w/o Alias Control

```
@Override
public void run() {
    while( !queue.is_closed() )
        System.out.println("Got object: " +
                           queue.dequeue());
    // Important shut-down code...
}
```

**queue is not
closed!**



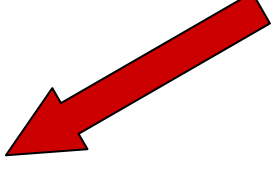
Verification w/o Alias Control

```
@Override
public void run() {
    while( !queue.is_closed() )
        System.out.println("Got object: " +
                             queue.dequeue());
    // Important shut-down code...
}
```

But could be
thread-
shared, so
forget it...

Verification w/o Alias Control

```
@Override
public void run() {
    while( !queue.is_closed() )
        System.out.println("Got object: " +
                           queue.dequeue());
    // Important shut-down code...
}
```



Call fails, pre-
condition not
met.

Verification w/o Alias Control

```
final Blocking_queue queue = new Blocking_queue();

(...).start(); // queue escapes to thread

for( int i=0;i<5;i++ )
    queue.enqueue("Object " + i);

queue.close();
```

Verification w/o Alias Control

```
final Blocking_queue queue = new Blocking_queue();  
  
(...).start(); // queue escapes to thread  
  
for( int i=0;i<5;i++ )  
    queue.enqueue("Object " + i);  
  
queue.close();
```

Queue must be
OPEN!

Verification w/o Alias Control

```
final Blocking_queue queue = new Blocking_queue();  
  
(...).start(); // queue escapes to thread  
  
for( int i=0;i<5;i++)  
    queue.enqueue("Obj");  
  
queue.close();
```

We must somehow encode:
The producer is ‘in control’
of the protocol!

Queue Method Signatures

```
@Full(requires="OPEN", ensures="OPEN")
```

```
void enqueue(@Share Object o)
```

```
@Full(requires="OPEN", ensures="CLOSED")
```

```
void close()
```

```
@Pure
```

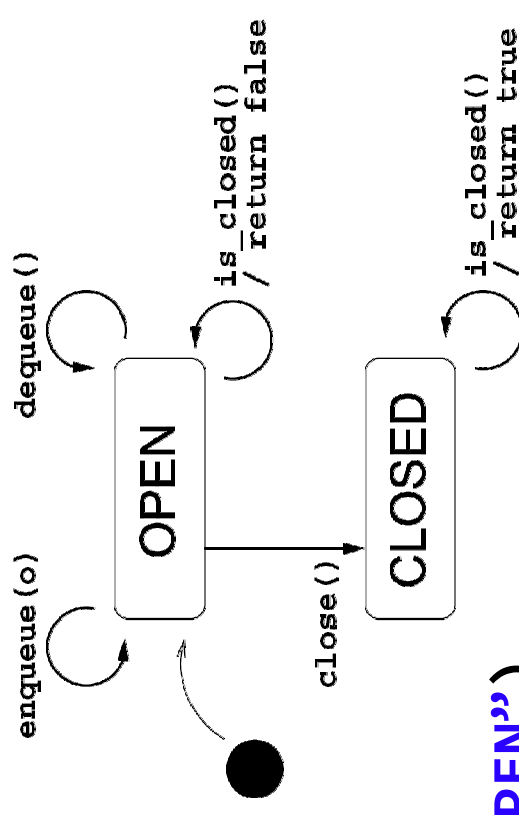
```
@TrueIndicates("CLOSED")
```

```
@FalseIndicates("OPEN")
```

```
boolean is_closed()
```

```
@Pure(requires="OPEN", ensures="OPEN")
```

```
Object dequeue()
```



Client-Side Verification: No Races on Abstract State

- Track permissions and state of references through method body
- At method call sites, use pre/post-conditions
- Discard object state if permission indicates concurrent modification
 - `@Pure` or `@Share`
- Unless inside atomic block!

```
final Blocking_queue queue = new Blocking_queue();

(...).start();

for( int i=0;i<5;i++ )
    queue.enqueue("Object " + i);

queue.close();
```

```
final Blocking_queue queue = new Blocking_queue();
```

```
(...).start();
```

```
for( int i=0;i<5;i++ )  
    queue.enqueue("Object " + i);
```

```
queue.close();
```

@Unique(queue)
in OPEN

```
final Blocking_queue queue = new Blocking_queue();
```

```
(...).start();
```

```
for( int i=0;i<5;i++ )  
    queue.enqueue("Object " + i);
```

```
queue.close();
```

@Full(queue) in
OPEN

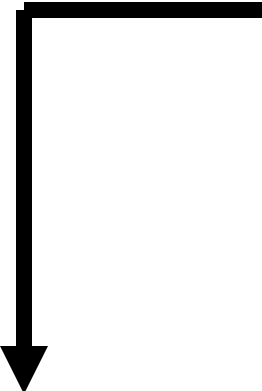
```
final Blocking_queue queue = new Blocking_queue();

(...).start();

for( int i=0;i<5;i++ )
    queue.enqueue("Object " + i);

queue.close();
```

Method
precondition
met



```
final Blocking_queue queue = new Blocking_queue();

(...).start();

for( int i=0;i<5;i++ )
    queue.enqueue("Object " + i);

queue.close();
```



Method
precondition
met

```
final Blocking_queue queue = new Blocking_queue();

(...).start();

for( int i=0;i<5;i++ )
    queue.enqueue("Object " + i);

queue.close();
```

→ @Full(queue) in
CLOSED

```
@Override
public void run() {
    while( !queue.is_closed() )
        System.out.println("Got object: " +
                           queue.dequeue());
    // Important shut-down code...
}
```

```
@Override
public void run() {
    while( !queue.is_closed() )
        System.out.println("Got object: " +
                           queue.dequeue());
    // Important shut-down code...
}

@Pure(queue)
```

from class
invariant...

```
@Override
public void run() {
    while( !queue.is_closed() )
        System.out.println("Got object: " +
                           queue.dequeue());
    // Important shut-down code...
}

@Pure(queue)
in
OPEN
```

```
@Override
public void run() {
    while( !queue.is_closed() )
        System.out.println("Got object: " +
                           queue.dequeue());
    // Important shut-down code...
}

@Pure(queue)
in
OPEN
```

```
@Override
public void run() {
    while( !queue.is_closed() )
        System.out.println("Got object: " +
                           queue.dequeue());
    // Important shut-down code...
}
ERROR!
```

But with ‘atomic’

```
@Override
public void run() {
    while( true ) {
        atomic: {
            if( !queue.is_closed() )
                System.out.println("Got object: "+queue.dequeue());
            else
                return;
        }
    }
    // Important shut-down code...
}
```

Implementation-Side Verification:

Transitions are Atomic

- States can be annotated with concrete invariants
 - Predicates over fields
- Use packing/unpacking for modular verification
 - Invariants must be reestablished before method returns
- Unpacking a `@Full`, `@Pure`, or `@Share` object must be within an atomic block

Verification Example

```
@ClassStates({
    @State(name="CLOSED",
        inv="closed == true * elements == null"), ...})
class Blocking_queue {
    private List elements;
    private boolean closed;
    // ...
    @Full(requires="OPEN", ensures="CLOSED")
    void close() {
        atomic: { elements = null; }
        // ...
        atomic: { closed = true; }
    }
    // ...
}
```

Verification Example

```
@ClassStates({
  @State(name="CLOSED",
    inv="closed == true * elements == null"), ...})
class Blocking_queue {
  private List elements;
  private boolean closed;
  // ...
  @Full(requires="OPEN", ensures="CLOSED")
  void close() {
    atomic: { elements = null; }
    // ...
    atomic: { closed = true; }
  }
  // ...
}
```

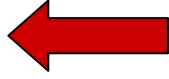
Verification Example

```
@ClassStates({
    @State(name="CLOSED",
        inv="closed == true * elements == null"), ...})
class Blocking_queue {
    private List elements;
    private boolean closed;
    // ...
    @Full(requires="OPEN", ensures="CLOSED")
    void close() {
        atomic: { elements = null; }
        // ...
        atomic: closed = true; }
    }
    // ...
}
```

**Unpacks from
OPEN state.**

Verification Example

```
@ClassStates({
    @State(name="CLOSED",
        inv="closed == true * elements == null"), ...})
class Blocking_queue {
    private List elements;
    private boolean closed;
    // ...
    @Full(requires="OPEN", ensures="CLOSED")
    void close() {
        atomic: { elements = null; }
        // ...
        atomic: { closed = true; }
    }
    // ...
}
```



**Packs to
CLOSED state.**

Verification Example

```
@ClassStates({
    @State(name="CLOSED",
        inv="closed == true * elements == null"), ...})
class Blocking_queue {
    private List elements;
    private boolean closed;
    // ...
    @Full(requires="OPEN", ensures="CLOSED")
    void close() {
        atomic: { elements = null; }
        // ...
        atomic { closed = true; }
    }
    // ...
}
```

**Error! Atomic
block ends while
receiver unpacked**

Queue: The Full Specification

```
@Refine({
    @States(dim="STRUCTURE", value={"STRUCTURESTATE"}),
    @States(dim="PROTOCOL", value= {"CLOSED", "OPEN"})
})
@ClassStates({
    @State(name="STRUCTURE",
        inv="share(elements) * reject_enqueue_requests ==
        true => full(this,PROTOCOL) in OPEN"),
    @State(name="OPEN", inv="closed == false"),
    @State(name="CLOSED", inv="closed == true")
})
public class Blocking_queue
{
```

Queue: The Full Specification

```
@Share(value="STRUCTURE")
@Full(requires="OPEN", ensures="OPEN",
      value="PROTOCOL")
void enqueue( Object new_element )

@Perm(requires="full(this,PROTOCOL) in OPEN *
      share(this,STRUCTURE)")
void enqueue_final_item(Object elm)

@Perm(requires="share(this!fr,STRUCTURE) *
      pure(this!fr,PROTOCOL) in OPEN",
      ensures="share(this!fr,STRUCTURE) *
              pure(this!fr,PROTOCOL)")
public Object dequeue( )
```

Queue: The Full Specification

```
@Pure(fieldAccess=true, value="PROTOCOL")
@TrueIndicates("CLOSED")
@FalseIndicates("OPEN")
boolean is_closed()

@Full(fieldAccess=true, value="PROTOCOL",
      requires="OPEN", ensures="CLOSED")
void close()
```



Take-Aways

- Protocols define temporal ordering of events
 - Can often be captured with state machines
- Protocol analysis needs to pay attention to
 - Interprocedural control flow
 - Aliasing of objects
- Disjoint sets and capabilities can handle aliasing correctly