Why We Should Not Add “Read-Only” to Java (yet)

John Boyland
(Updated Slides, 2005/8/4)
Why We Should Not Add “readonly” to Java (yet)

○ Why? Java method signatures don’t constrain pointers:
  – parameters may be mutable or retained;
  – return values may be mutable, or retained indefinitely.

○ Other people propose “readonly” type qualifier:
  – similar to “const” in C++.

○ Why not? Poor Solution:
  – doesn’t address “real” problem (representation exposure);
  – most solutions enshrine overly strict transitivity rule.

○ (Other solutions to address problem are maturing.)
Example (1 of 5)

- Integer set class [Birka & Ernst 2004]
  
  ```java
  public class IntSet {
      /** ints in the set with no duplicates. **/ 
      private int[] ints;

      /** Removes all elements from this that 
      * are not in set, without modifying set. 
      **/ 
      public void intersect(IntSet set) {
          ...
          ...
      }
  }

  ...
  ```

FTJP ‘05 (Glasgow, UK)  Why we should not add readonly to Java 3
Example (1 of 5)

- Integer set class [Birka & Ernst 2004]

```java
public class IntSet {
    /** ints in the set with no duplicates. /**/
    private int[] ints;

    /** Removes all elements from this that
     * are not in set, without modifying set. **/
    public void intersect(IntSet set) {
        ... set.ints[0] = 0;
        ...
    }
}
...
```

Code in this method could accidentally or intentionally mutate the data in parameter

Why we should not add readonly to Java 4
Example (1 of 5)

Integer set class [Birka & Ernst 2004]

```java
public class IntSet {
    /** ints in the set with no duplicates. **/
    private int[] ints;

    /** Removes all elements from this that
     * are not in set, without modifying set.
     **/
    public void intersect(IntSet set) {
        ... this.lastInter = set;
        ...
    }
}
```

Code in this method could accidentally or intentionally retain the parameter
Example (1 of 5)

Integer set class [Birka & Ernst 2004]

```java
public class IntSet {
    /** ints in the set with no duplicates. **/  
    private int[] ints;

    /** Removes all elements from this that
       * are not in set, without modifying set.
       **/  
    public void intersect(readonly IntSet set) {
        ...
    }
    ...
}
... 
```

The “readonly” qualifier means
that mutations to the object through
the reference are forbidden.
Example (2 of 5)

- Integer set class (cont’d)
  ```java
  /** ints in the set with no duplicates. **/
  private int[] ints;
  ...
  /** Makes an IntSet from an int[].
   * Throws BadArgumentException if
   * duplicates in the argument. **/
  public IntSet(int[] ints) {
    if (hasDuplicates(ints))
      throw new BadArgumentException();
    this.ints = ints;
  }
  ...
  ```

Client could retain argument and accidentally or intentionally break invariant by mutating array.
Example (2 of 5)

- Integer set class (cont’d):
  /** ints in the set with no duplicates.**/
  private int[] ints;
  ...
  /** Makes an IntSet from an int[].
   * Throws BadArgumentException if
   * duplicates in the argument. **/
  public IntSet(int[] ints) {
    if (hasDuplicates(ints))
      throw new BadArgumentException();
    this.ints = ints;

Three reasonable designs:
1. client is supposed to release arg (unique);
2. array is supposed to be immutable;
3. constructor is supposed to copy array,
   (and neither mutate nor retain it).
Example (2 of 5)

- Integer set class (cont'd)

```java
/** ints in the set with no duplicates. **/
private int[] ints;
...
/** Makes an IntSet from an int[].
 * Throws BadArgument Exception if
 * duplicates in the argument. **/
public IntSet(readonly int[] ints) {
    if (hasDuplicates(ints))
        throw new BadArgument Exception();
    this.ints = (int[]) ints.clone();
}
...
```

Birka and Ernst suggest “readonly” to enforce design (3), except non-retention. Constructor must clone argument.
Example (3 of 5)

- **Integer set class (cont’d)**
  
  ```java
  /** ints in the set with no duplicates. ***/
  private int[] ints;
  ...
  /** Number of distinct elements of this. ***/
  public int size() {
    return ints.length;
  }
  ```

- **The size() method doesn’t change the set.**
Example (3 of 5)

- Integer set class (cont’d)

  /** ints in the set with no duplicates. ***/
  private int[] ints;

  /** Number of distinct elements of this. ***/
  public int size() {
    return ints.length;
  }

  Specification of size() does not prevent body of method from mutating this set.

  Neither does it prevent this set from being retained.
Example (3 of 5)

Integer set class (cont’d)
/** ints in the set with no duplicates.***/
private int[] ints;
...
/** Number of distinct elements of this. ***/
public int size() {
    return ints.length;
}

Birka and Ernst propose “readonly” to qualify the receiver of the method. In such a method, mutation is forbidden.
Example (4 of 5)

- Integer set (cont’d)
  ```java
  /** ints in the set with no duplicates. **/
  private int[] ints;
  ...
  /** Return an array with ints. */
  public int[] toArray() {
    return ints;
  }
  ```

- The `toArray` method is a standard collection method that returns a new array. Here, we have an “optimized” version.
Example (4 of 5)

- Integer set (cont’d)

```java
/** ints in the set with no duplicates. **/
private int[] ints;
...
/** Return an array with ints. */
public int[] toArray() {
    return ints;
}
```

This code exposes the internal representation to accidental or intentional changes by client.

Client can also see inner workings, accidentally (or intentionally) if it retains the pointer a “long” time.
Example (4 of 5)

- Integer set (cont’d)
  /** ints in the set with no duplicates. ***/
  private int[] ints;
  ...
  /** Return an array with ints. */
  public readonly int[] toArray() {
    return ints;
  }

Birka and Ernst concede that an “observational” exposure still results, but at least client cannot mutate.
Example (5 of 5)

- Graphical viewer of IntSet: (not from [Birka & Ernst 2004])

```java
public class IntSetView extends JPanel {
    private final IntSet model;

    /** Construct a view of the given set.
     * When the set changes, the client
     * should call repaint().
     **/
    public IntSetView(IntSet set) {
        model = set;
        ...
    }

    ...
```

FTJP ‘05 (Glasgow, UK)  Why we should not add readonly to Java 16
Example (5 of 5)

Graphical viewer of IntSet: (not from [Birka & Ernst 2004])

```java
public class IntSetView extends JPanel {
    private final IntSet model;

    /** Construct a view of the given set.
     * When the set changes, the client
     * should call repaint().
     **/
    public IntSetView(IntSet set) {
        model = set;
        ...
    } ...
```

Client does not expect class to mutate the set, but typesystem does not prevent this.

Client *does* expect class to retain reference, and observe changes!
Example (5 of 5)

Graphical viewer of IntSet: (not from [Birka & Ernst 2004])

```java
public class IntSetView extends JPanel {
    private final IntSet model;

    /** Construct a view of the given set. 
     * When the set changes, the client 
     * should call repaint(). 
     **/
    public IntSetView(IntSet set) {
        model = set;
        ...
    }

    ... 
}

Here "readonly" gives precisely the correct semantics.
```

Why we should not add readonly to Java 18
Examples: Conclusions

○ Correct functioning of code may require
  – lack of mutation though reference;
  – lack of any mutation;
  – non-retention of pointers.
  And this is in motivating examples for “readonly”!

○ A “readonly” qualifier handles the first requirement only.

○ Representation exposure is a problem
  – “readonly” can help prevent outsiders from corrupting rep.
  – Is this sufficient?
Problems Using "readonly" to Prevent Rep. Exposure

- Anything that the client can see is part of the interface:
  - Outsiders may make unwarranted assumptions;
  - Changing internal representation resisted by clients;
  "Observational Exposure" increases (bad) COUPLING.

- Concurrency complicated by existence of many readers:
  - Can clients see internal structures in invalid states?
  - What mutual exclusion locks will prevent race conditions?
  - Performance on distributed machines is compromised.

Glass Walls Are Not Enough!

FTJP ‘05 (Glasgow, UK)  Why we should not add readonly to Java 20
Rep. Exposure is Similar to Observational Exposure

- Each permits “covert communication”: information passed around the abstraction barrier
  - manifest communication goes through a declared service
  - covert communication goes through shared mutable state:

- The only difference is on “which side” of the abstraction barrier the state notionally lives.
Read-Only in OO Languages:

- C++ "const"
- JAC (Java with Access Control) [Kniesel & Theisen 2001]
- Mode system for Java [Skoglund & Wrigstad 2001]
- Javari [Birka & Ernst 2004, Tschantz & Ernst 2005]

Similar rules
Read-Only Rules

- Read-only reference is supertype of read-write:
  ```java
  readonly List rol;
  List l;
  rol = l;  // OK!
  l = rol;  // NOT OK
  ```
  Similarly with parameter passing.

- Fields of readonly reference cannot be updated:
  ```java
  Point p;
  readonly Point rop;
  p.x = 3;  //OK! p is mutable
  System.out.println(rop.x);  //OK! only reading field
  rop.x = 10;  // NOT OK
  ```
Read-Only Rules: Loop Holes

- Some fields shouldn’t be protected from mutation:
  - a cache that is updated on demand;
  - a statistics counter.

- Most proposals have a “mutable” qualifier for such fields:
  ```java
  mutable int hashCode = -1; // -1 = invalid
  public int hashCode() readonly {
    if (hashCode != -1) return hashCode;
    hashCode = sum(); // update cache
    return hashCode;
  }
  
  This follows C++.

  Javari [2005] uses “assignable” as the keyword here.
Sub-objects should be read-only if owning object is:
- inherited fields (obviously)
- representation objects / internal objects
  - Points of a Rectangle
  - Nodes in a LinkedList

C++ const applies to objects nested in *this
- but not to “pointed to” objects (even if rep)!

Java does not distinguish internal/external objects:
- strict: assume it is a representation object;
- lenient: assume it isn’t.
Read-Only Rules: Transitivity (2 of 2)

- All the Java read-only proposals are strict:
  - a field read through a read-only pointer is made read-only.

- This rule is **wrong** when the field does not refer to a “rep”
  - e.g.: A read-only list of movable shapes:
    the shapes fetched from the list will be read-only too.
  - Javari [2005] “mutable” qualifier is useful here.

- It should distinguish internal from external objects:
  - internal ones subject to transitivity;
  - external objects, not.

Read-Only Rules: Casts (1 of 2)

- C++ permits `const` to be cast away.

- “Universes” permit “readonly” to be cast away if in the class which “owns” the pointer.
  - “readonly” means “writes only if authorized”

- JAC does not permit “readonly” to be cast away
  - a dynamic check would be too costly;
  - unchecked casts break guarantees.

- Skoglund & Wrigstad permit dynamic casts in polymorphic methods to check whether they actually can mutate.
Read-Only Rules: Casts (2 of 2)

○ In Javari, mutable casts always succeed, but mutations are checked dynamically.
  – Slows down modifications everywhere;
  – Not how dynamic casts are supposed to work.
Used to interface with legacy code.

○ Safe dynamic casts require expensive run-time support:
  – In our proposed formulation, the dynamic casts would almost always be unsuccessful anyway.
  – Perhaps casts should simply leave a verification error.
Conclusions and Further Work

○ “readonly” only handles some aspects of the problems.

○ “readonly” rules would enforce wrong principles:
  – strict transitivity;
  – “observational exposure”

○ Better is some technique of aliasing control:
  – true ownership
    • [Clarke et al]
    • [Aldrich et al]
    • [Boyapati et al]
  – permissions [Boyland et al]
Our Proposal: Permissions

- Every piece of state is associated with a permission:
  - if you have the whole thing, you can write;
  - if you have a fraction, you can read.

- You can split a piece, or merge two pieces (of same permission).

- Prevents a write and a read from happening at *same* time.
Our Proposal: Method Effects

- Method effects are passed to method:
  - so it can do its work,
  - and then are returned.

A read effect can be handled by any fraction:

FTJP ‘05 (Glasgow, UK)  Why we should not add readonly to Java 31
Our Proposal: Unique and Immutable Pointers

- We can package permissions with a pointer:
  - “unique” using the entire permissions;
  - “immutable” using some non-zero fraction:

- Packages cannot be copied; immutable ones can be split:
Our Proposal: Nesting State

- State for internal objects can be nested inside

(Internal permissions are hidden from external viewing.)

- Internal state can be accessed by “carving out” permission:

This ensures transitivity for internal objects.
Our Proposal: Global Permissions

- Permissions can be nested in the global space:
  - Full permissions has meaning of “shared”;
  - Partial permission has meaning of “readonly”;
  - Permissions can be carved out by any method that announces it needs to do global access.
Our Proposal: Annotations (1 of 5)

- Integer set class [Birka & Ernst 2004]
  ```java
  public class IntSet {
      /** ints in the set with no duplicates. **/
      private unique int[] ints;

      /** Removes all elements from this that
       * are not in set, without modifying set.
       **/
      public void intersect(IntSet set)
          reads set.*, writes this.*
      {
          ...
          ...
      }
  }
  ```

FTJP ‘05 (Glasgow, UK) Why we should not add readonly to Java 35
Our Proposal: Annotations (2 of 5)

- Integer set class (cont’d)
  /** ints in the set with no duplicates.**/
  private unique int[] ints;
  ...
  /** Makes an IntSet from an int[].
   * Throws BadArgumentException if
   * duplicates in the argument. **/
  public IntSet(unique int[] ints) {
    if (hasDuplicates(ints))
      throw new BadArgumentException();
    this.ints = ints;
  }
Our Proposal: Annotations (3 of 5)

- Integer set class (cont'd)
  /** ints in the set with no duplicates. **/  
  private unique int[] ints;
  ...  
  /** Number of distinct elements of this. **/  
  public int size() reads this.* {  
    return ints.length;
  }
Our Proposal: Annotations (4 of 5)

- Integer set (cont’d)
  /** ints in the set with no duplicates. ***/
  private unique int[] ints;
  ...
  /** Return an array with ints. */
  public from(this.*) int[] toArray()
      reads this.*
  {
      return ints;
  }
Our Proposal: Annotations (5 of 5)

Graphical viewer of IntSet: (not from [Birka & Ernst 2004])

```java
public class IntSetView extends JPanel {
    private final Readonly IntSet model;

    /** Construct a view of the given set.  
     * When the set changes, the client
     * should call repaint().
     **/
    public IntSetView(Readonly IntSet set) {
        model = set;
        ...
    }
```
Conclusions

- “readonly” only handles some aspects of the problems.

- “readonly” rules would enforce wrong principles:
  - strict transitivity;
  - “observational exposure”

- Permissions describes solution better.

- Flexible enough to give meaning to many annotations:
  - unique, immutable, shared
  - read, write effects
  - AND readonly, when this is the correct semantics.