



J-Spring

16 april 2008 Spant! - Bussum



Preventing bugs with pluggable type-checking

Michael Ernst

MIT

<http://pag.csail.mit.edu/jsr308/>



Problem:

Java's type checking is too weak

- Type checking prevents many bugs

```
int i = "42"; // type error
```
- Type checking doesn't prevent **enough** bugs

```
// possible NullPointerException!  
getValue().toString();
```
- Java types cannot express important properties
 - Non-null, interned, immutable, encrypted, tainted, ...
- Solution: **pluggable** type systems
 - Design a type system to solve a specific problem
 - Annotate your code with type qualifiers
 - Type checker warns about violations (bugs)



Code example: Neighbors of a graph node

```
class Graph<Node, Edge> {
    Set<Edge> edges;

    // ...

    List<Node>
    getNeighbors(          Node v) {
        List<Node> neighbors = new LinkedList<Node>();
        for (Edge e : edges)
            if (e.from() == v)
                neighbors.add(e.to());
        return neighbors;
    }
}
```



Prevent null pointer errors: @NonNull type qualifier

@NonNullDefault

```
class Graph<Node, Edge> {
    Set<Edge> edges;

    // ...

    List<Node>
    getNeighbors(          Node v) {
        List<Node> neighbors = new LinkedList<Node>();
        for (Edge e : edges)
            if (e.from() == v)
                neighbors.add(e.to());
        return neighbors;
    }
}
```



Prevent null pointer errors: @NonNull type qualifier

@NonNullDefault

```
class Graph<Node, Edge> {
    @NonNull Set<@NonNull Edge> edges;

    // ...

    @NonNull List<@NonNull Node>
    getNeighbors( @NonNull Node v) {
        List<Node> neighbors = new LinkedList<Node>();
        for (Edge e : edges)
            if (e.from() == v)           // OK: no null ptr
                neighbors.add(e.to());  // OK: no null ptr
        return neighbors;
    }
}
```



Prevent incorrect equality tests: @Interned type qualifier

```
class Graph<Node, Edge> {
    Set<Edge> edges;

    // ...

    List<Node>
    getNeighbors(@Interned Node v) {
        List<Node> neighbors = new LinkedList<Node>();
        for (Edge e : edges)
            if (e.from() == v)           // OK
                neighbors.add(e.to());
        return neighbors;
    }
}
```



Prevent incorrect side effects: @ReadOnly type qualifier

```
class Graph<Node, Edge> {
    Set<Edge> edges;

    // ...

    List<Node>
    getNeighbors(@ReadOnly Node v) @ReadOnly {
        List<Node> neighbors = new LinkedList<Node>();
        for (Edge e : edges)
            if (e.from() == v)
                neighbors.add(e.to());
        return neighbors;
    } // body is OK: no side effects
}
```



Using a checker

- Write annotations on the code
 - Or use an inference tool
- Compiler plugin for javac

```
javac -processor NonnullChecker MyFile.java
```

- Produces additional errors/warnings
- Can use it from an IDE (e.g., Eclipse)



Outline

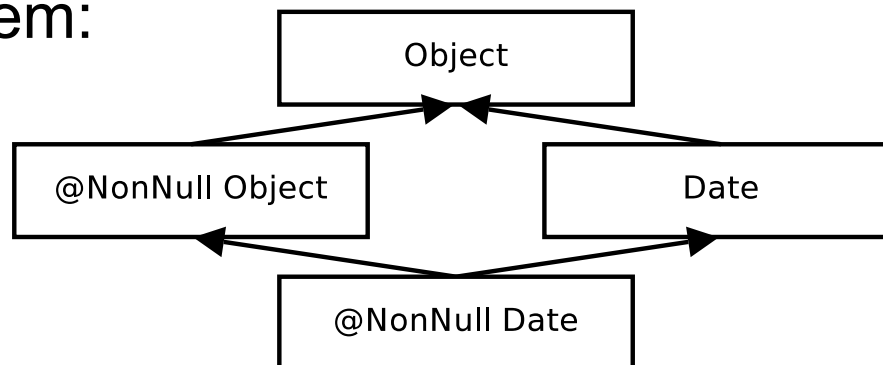
- NonNull checker
- Interned checker
- IGJ (immutability) checker
- Creating your own checker



Null dereference checker. Problem: NullPointerExceptions

```
Object obj;    // might be null
@NonNull Object nobj; // never null
nobj.toString(); // OK
obj.toString(); // possible NPE
obj = ...;     // OK
nobj = obj;    // nobj may become null
```

Type system:





Demo of null dereference checker



Comparison with other tools

	Null pointer errors			Annotations written
	Found	Missed	False	
JSR 308	7	0	7	45
FindBugs	0	7	1	0
Jlint	0	7	8	0
PMD	0	7	0	0

Checking a 4KLOC program

Type checking finds all the NPE bugs

The other tools also find non-NPE bugs



Good defaults reduce user effort

- Nullable default:
 - Dictated by backward compatibility
 - Many instances of `@NonNull` – too verbose
 - Flow-sensitive analysis inferred many NonNull types for local variables, reducing annotation burden
- NonNull default
 - Less verbose in signatures
 - Draws attention to exceptions rather than the rule
 - More annotations in method bodies
- Our system: Nullable only for local variables
 - Not for generics on local variables
 - An alternative to type inference; effect may be similar



Local type inference (flow-sensitivity)

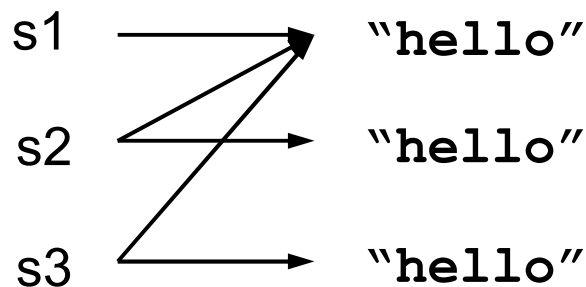
```
/*@Nullable*/ Date d1, d2, d3;  
d1 = new Date();  
d1.getMonth(); // OK: d1 is non-null  
assert d2 != null;  
d2.getMonth(); // OK  
if (d3 != null) {  
    d3.getMonth(); // OK  
}
```

Often, no need for annotations on local vars



Interning (= canonicalization, hash-consing)

- Reuse an existing object instead of creating a new one
- A **space** optimization: savings can be significant
- A **time** optimization: use `==` for comparisons
- Built into `java.lang.String`: `intern()` method



- Users can add interning for their own classes



Interning (= canonicalization, hash-consing)

- Reuse an existing object instead of creating a new one
- A **space** optimization: savings can be significant
- A **time** optimization: use `==` for comparisons
- Built into `java.lang.String`: `intern()` method

s1 → "hello"

s2 → "hello"

s3 → "hello"

- Users can add interning for their own classes



Interning (= canonicalization, hash-consing)

- Reuse an existing object instead of creating a new one
- A **space** optimization: savings can be significant
- A **time** optimization: use `==` for comparisons
- Built into `java.lang.String`: `intern()` method

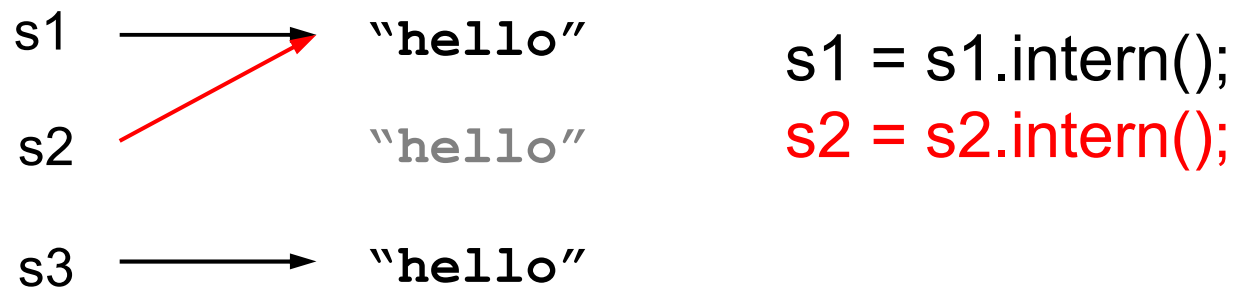
s1 \longrightarrow "hello" `s1 = s1.intern();`
s2 \longrightarrow "hello"
s3 \longrightarrow "hello"

- Users can add interning for their own classes



Interning (= canonicalization, hash-consing)

- Reuse an existing object instead of creating a new one
- A **space** optimization: savings can be significant
- A **time** optimization: use `==` for comparisons
- Built into `java.lang.String`: `intern()` method

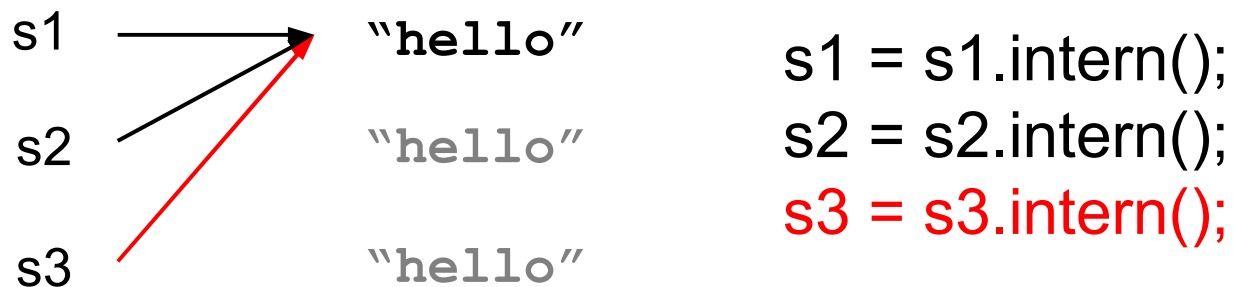


- Users can add interning for their own classes



Interning (= canonicalization, hash-consing)

- Reuse an existing object instead of creating a new one
- A **space** optimization: savings can be significant
- A **time** optimization: use `==` for comparisons
- Built into `java.lang.String`: `intern()` method

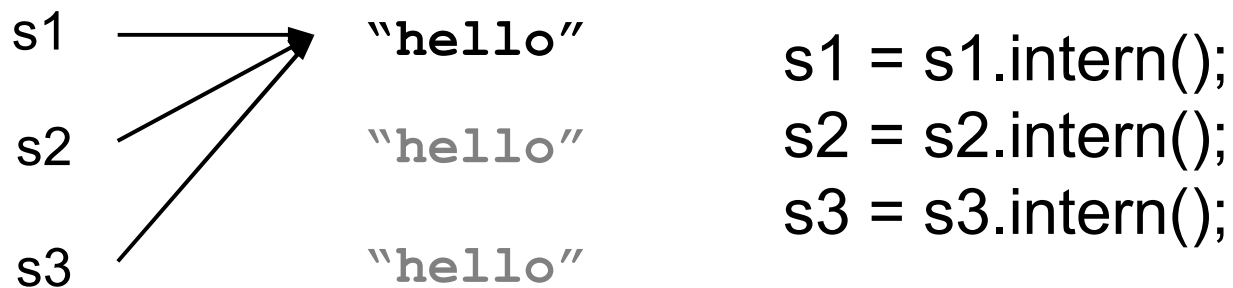


- Users can add interning for their own classes



Interning (= canonicalization, hash-consing)

- Reuse an existing object instead of creating a new one
- A **space** optimization: savings can be significant
- A **time** optimization: use `==` for comparisons
- Built into `java.lang.String`: `intern()` method



- Users can add interning for their own classes

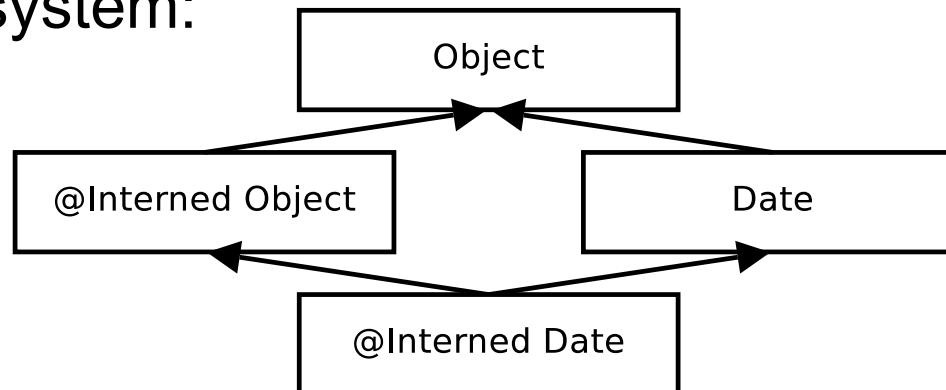


Interned checker.

Problem: incorrect equality checks

```
String s;  
@Interned String is, is2;  
... is = myString.intern() ... // OK  
if (is == is2) { ... } // OK  
if (s == is) { ... } // unsafe equality  
is = s; // unsafe assignment
```

- Type system:





Demo of Interned checker



Interned case study

- Program: Daikon (250 KLOC)
- Interning is important
 - Daikon's key scalability problem is memory
 - 1170 lines of code/comment refer to interning
 - 72% of files have none, 87% have 0, 1, or 2
 - 200 run-time assertions
 - Emacs plug-in for `string` equality checking
- 124 annotations in 11 files (12KLOC)



Interned results

- 9 errors
 - Missing calls to `intern`
- 2 performance bugs
 - Unnecessary interning in inner loop of file reading
- 1 design flaw
 - `VarInfoName` is not interned within its implementation, but all escaping objects are
 - Too hard to understand complex interning
- 14 false positives (used `@SuppressWarnings`)



Observer methods can expose state

```
class Class {  
    private Object[] signers;  
  
    Object[] getSigners() {  
        return signers; // JDK 1.1 bug  
    }  
}  
  
myClass.getSigners()[0] = "Sun";
```



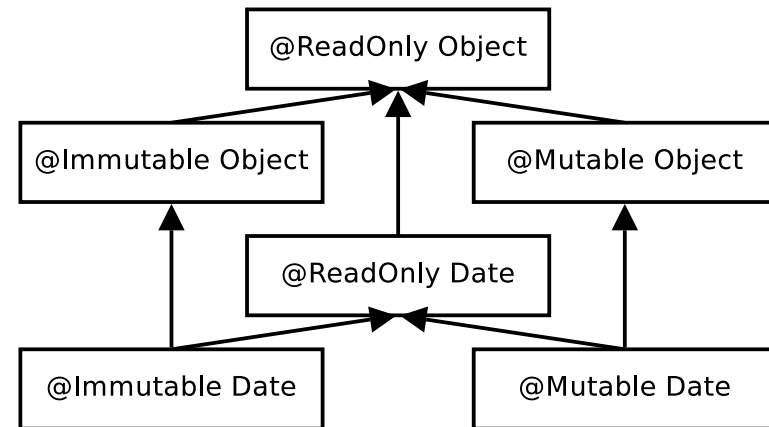
Observer methods can expose state

```
class Class {  
    private Object[] signers;  
  
    @ReadOnly Object[] getSigners() {  
        return signers; // Fixes JDK 1.1 bug  
    }  
}  
  
myClass.getSigners()[0] = "Sun"; // Error
```



IGJ (Immutability Generic Java). Problem: unintended side effects

- Type system:
 - `ReadOnly`: Reference immutability (aliases may modify)
 - `Immutable`: Object immutability (object cannot change)



- Inspired by generics
- Syntax uses annotations, not generics



Demo of IGJ checker



IGJ case study

- Programs (128KLOC)
 - JOlden benchmarks
 - htmlparser library
 - tinySQL library
 - SVNKit subversion client
 - IGJ checker
- 4760 annotations (62133 possible locations)



IGJ case study results

- Representation exposure errors
- Constructors that left object in inconsistent state
- In SVNKit:
 - some getters have side effects
 - some setters have no side effects
- Used both reference and object immutability
- More opportunities for immutability in new code than old



Java syntax for Annotations on types

- Permits annotations in new locations

```
List<@NonNull String> myList;  
myGraph = (@Immutable Graph) tmpGraph;  
class UnmodifiableList<T>  
    implements @ReadOnly List<@ReadOnly T> {...}
```

- Planned for inclusion in Java 7 (“JSR 308”)
- Publicly-available implementation
- Implementation is backwards compatible
 - Optionally write annotations in comments

```
List</*@NonNull*/ String> myList;
```
 - Compiles with any Java compiler



Case studies

- 4 checkers
- 360KLOC
- 75 bugs verified by a human and fixed
- See technical report from September 2007 (better today!)



Usability of pluggable checkers

- Scales to >200,000 LOC
- Found bugs in every codebase
 - More important: prevents bugs!
- Few false positives
- Programmers found the checkers easy to use
- Is it too verbose?
 - @NonNull: 1 per 75 lines
 - @Interned: 124 annotations in 220KLOC revealed 11 bugs
 - Possible to annotate part of program
 - Fewer annotations in new code



Creating your own checker

- Most users do not need to
- Basic functionality: mention annotation on javac command line
- Simple rules: declarative syntax (meta-annotations)
- Special rules: override a few methods
 - For NonNull: dereferences
 - For Interned: equality
- Examples of properties you can check:
 - General-purpose (tainting)
 - Project-specific (string formatting)



Your turn to use pluggable type checkers

- Webpage: Search for “JSR 308” or “Annotations on Java types”
 - <http://pag.csail.mit.edu/jsr308/>
- Downloads:
 - Checkers: @NonNull, @Interned, @ReadOnly, @Immutable
 - Inference tools: NonNull, ReadOnly
 - Checkers Framework for writing new checkers
 - JSR 308 compiler (patch to OpenJDK)
- Go forth and **prevent bugs!**