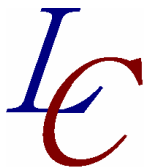




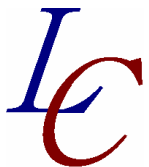
# **Safety-Critical Java Technology – JSR-302 Annotation Resolution**

**Doug Locke**  
**Locke Consulting LLC**  
**[www.douglocke.com](http://www.douglocke.com)**



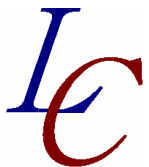
# Fundamental Safety-Critical Issues

- There are a number of critical issues that come to mind about Safety-Critical Java, e.g.,
  - Simplicity
  - Size
  - Sufficient functionality
  - Certification
  - Concurrency & Synchronization
  - Time Constraints
  - Finalization
  - Exceptions
  - etc.



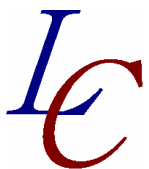
# Focus on Exceptions

- Some exceptions are “expected”
  - Such as divide by zero
  - Application designers generally know how to handle them
- Some exceptions are planned, e.g.,
  - That’s why *raise* is in the Java language
- Some exceptions cannot be successfully handled, e.g.,
  - Illegal Memory Reference



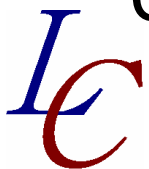
# Must Prevent Exceptions

- When an exception cannot be successfully handled by the application, it must be prevented.
- For memory reference errors, it must be possible to *prove* that such errors cannot ever happen
  - If proven, the run-time need not check for it
  - If proven, the application has a critical safety property



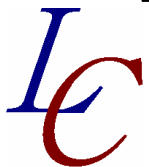
# How to Avoid These Exceptions

- In the Safety-Critical Java Expert Group
  - There were (and still are) several approaches to this problem
  - Presented by different vendors
  - Each approach has some significant benefits and drawbacks
  - The approaches were generally mutually exclusive
- The entire Expert Group badly wanted an acceptable solution that had a broad consensus



# The Solution

- Safety-Critical Java will define a set of annotations.
  - All annotations will be optional for application developers
    - If used, related safety properties will be guaranteed, and runtime checks will not be needed
    - If not used, related safety properties will not be guaranteed, and the implementation must check for the related errors at runtime
  - Support for all annotations will be mandatory for conformant implementations
- Acknowledgement: Jan Vitek was very instrumental in analyzing the proposals and pulling this together and showing that it could work. Jan created the examples.



# Current Annotations

## Preventing Exceptions

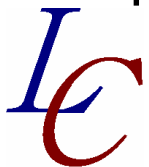
- `public @interface InScope { String name(); }`
- `public @interface Scope { String parent(); String name(); }`
- `public @interface Nested { String() target; }`
- `public @interface Outer { String() target; }`
- `public @interface Allocate { String() what }`
- `public @interface Immutable { String() type }`

## Defining permissible application API usage

- `public @interface SCJ_Allowed { int level }`

## Permitting synchronization optimization

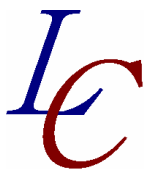
- `public @interface NoBlocking { }`



# @InScope, @Nested

- @InScope is an annotation on a class
  - names the scope in which instances of the class will be allocated.
- @Nested is an annotation on a Runnable
  - which performs an enter() on a child scope
- E.g.,

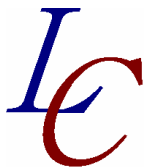
```
@InScope(name = "Top") @Nested(target = "A")
public class BRunner implements Runnable {
    Client c;
    A parent;
    int i;
    public void run() {
        i = new BalanceInvoker(c, parent).invoke();
    }
}
```



# @Scope

- Annotation on a subclass of ScopedMemory
  - Uniquely names the scope associated with instances of this subclass of ScopedMemory and its parent
  - E.g.,

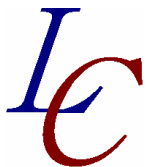
```
@Scope(parent = "Top", name = "A")
public class A extends LTMemory {
    public A(long sz0, long sz1) {super(sz0, sz1); }
}
```



# @Outer

- Annotation on a Runnable which performs an `executeInArea()` on an ancestor scope
- E.g.,

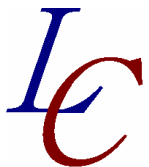
```
@InScope(name="B") @Outer(target="A")
public class BalanceInvoker implements Runnable {
    Client c;
    A scope;
    int result;
    BalanceInvoker(Client c , A scope ) {
        scope = scope ;
        c = c ;
    }
    int invoke() {
        scope.executeInArea(this);
        return result;
    }
    public void run() {
        result = c.balance();
    }
}
```



# @Allocate, @Immutable

- @Allocate is an annotation on a method that performs no *new*'s. Argument must be “none”
- @Immutable is an annotation on a method that limits modification of objects. Argument must be “reference” or “pure”
- E.g.,

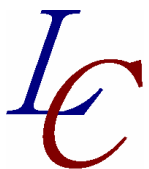
```
public class Client {  
    @Allocate (what=none) @Immutable(pure)  
    int balance() {  
        scope.executeInArea(this);  
        return result;  
    }  
    public void run() {  
        result = c.balance();  
    }  
}
```



# @SCJ\_Allowed

- Annotation on a class that expects the implementation to provide support for a particular compliance level (or higher)
- E.g.,

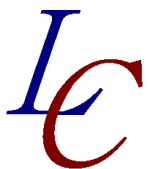
```
@SCJ_Allowed (level=1)
public class Client {
    @Allocate (what=none) @Immutable(pure)
    int balance() {
        scope.executeInArea(this);
        return result;
    }
    public void run() {
        result = c.balance();
    }
}
```



# @ NoBlocking

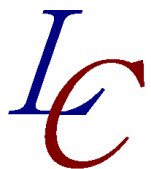
- Annotation on a synchronized method that will never voluntarily relinquish the CPU prior to exiting.
- E.g.,

```
@SCJ_Allowed (level=1)
public class Client {
    @Allocate (what=none) @Immutable(pure)
    int balance() {
        scope.executeInArea(this);
        return result;
    }
    @NoBlocking()
    public void synchronized run() {
        result = c.balance();
    }
}
```



# Conclusions

- These annotations are not complete
  - And they may not be correct (yet)
  - But they represent the Expert Group's current thinking
- Community comments welcome!





**Doug Locke, Ph.D.**  
**Locke Consulting LLC**  
[www.douglocke.com](http://www.douglocke.com)  
[doug@douglocke.com](mailto:doug@douglocke.com)

