Lecture 8 — **Case Study:** Safety Critical Java

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

JSR302 — Safety-Critical Java Technology Specification

July 11th, 2006. *JSR302 vote passed*

(10 x yes, 1 x no and 5 x abstain)

July 25th, 2006. *Expert committee formed.*

June 7th, 2011. *Early Draft for Review.*

June 28th, 2013. *Early Draft 2 for Review.*

Feb 8th, 2017. *Early Draft 3 for Review. (875 pages)*

JSR302 — Introduction

“Most safety-critical software development projects are carefully designed to **reduce the application size and scope** *to its most* **minimal form** *to help manage the costs associated with the development of certification evidence. Examples of the resulting restrictions may include the elimination or severe limitations on* **recursion** *and the* **rigorous use of memory**, *especially heap space, to ensure that out-of-memory conditions are precluded.”*

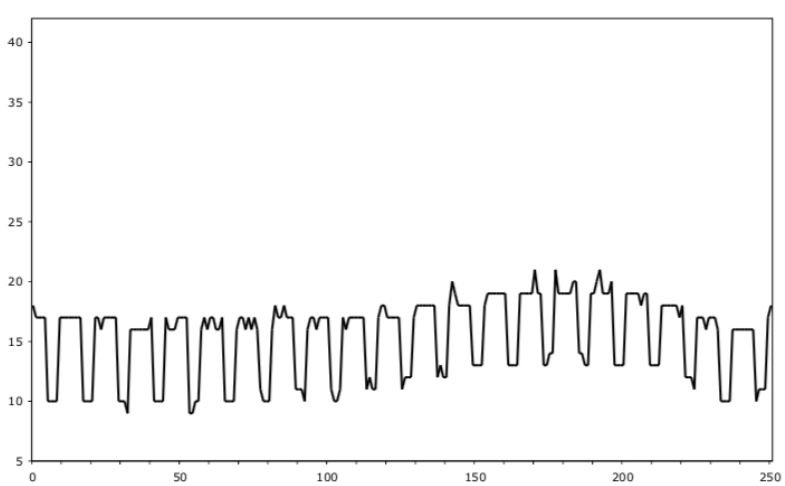
...

“This safety-critical specification is designed to enable the creation of safety-critical applications, built using safety-critical Java infrastructures, and using safety-critical libraries, **amenable to certification under DO-178C**, *Level A, as well as other safety- critical standards.”*

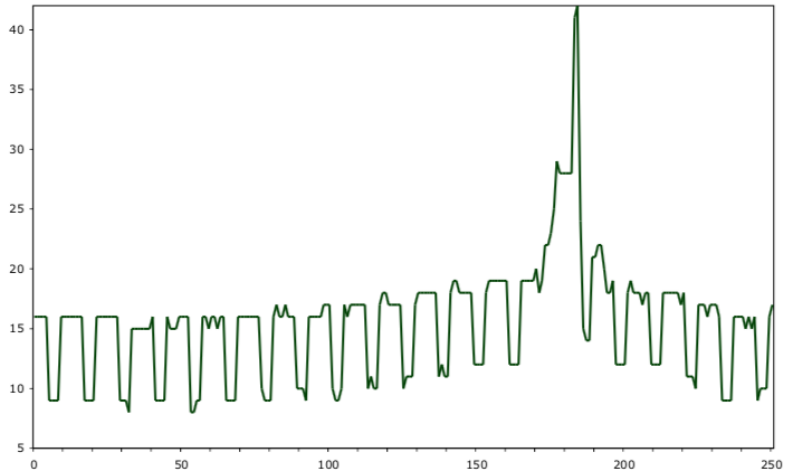
JSR302 — Java for Safety Critical Systems??

“There are **many issues** *associated with the use of Java technology in a safety-critical system but the two largest issues are related to ...”*

Garbage Collection



(Real-Time Specification for Java)



(Real-Time Garbage Collector)

- Comparative Performance of Real-Time **Collision Detection**
- Real-Time Specification for Java (RTSJ) was **strong influence** on Safety-Critical Java. E.g., RTSJ introduces **scoped memory**.

See: *"Scoped Types and Aspects for Real-Time Java"*, Chris Andreae, James Noble, *et al.*, ECOOP'06.

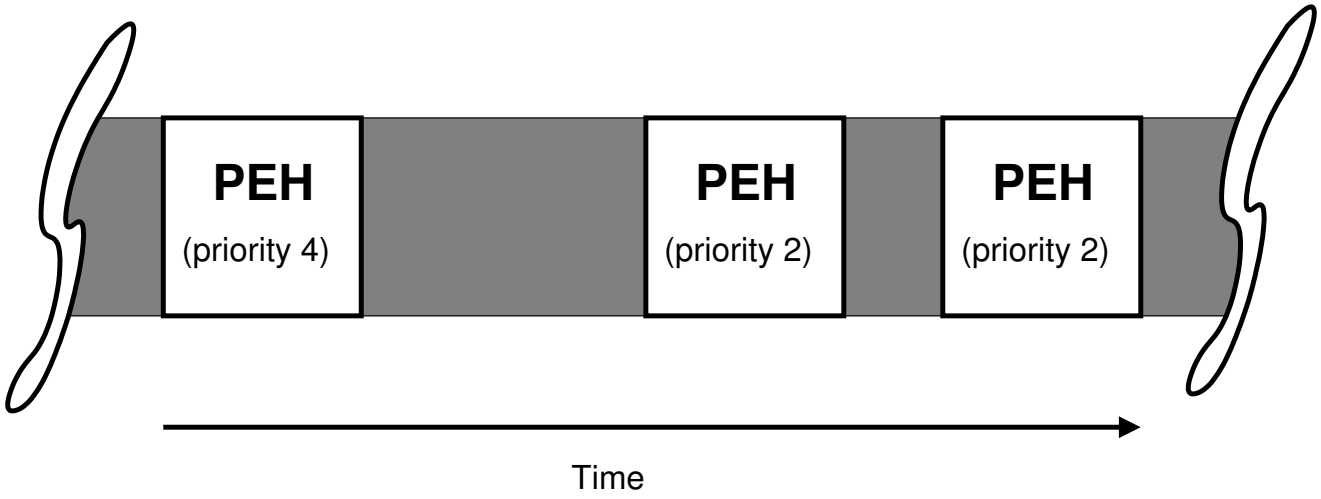
JSR302 — A Java Subset

- JSR302 defines a **subset** of Java
- Constraints on Java include:
  - Severe restrictions on *concurrency*
  - Constraints on the usage of *dynamic memory allocation*
  - *Dynamic class loading* not supported.
  - Many Java classes are omitted.
  - Different procedure for starting an application

JSR302 — Programming Model

“In this specification, a flexible programming model is defined that is intended to be sufficiently limited to enable certification under such standards as DO-178C Level A. This is accomplished by defining concepts such as a **mission**, **limited start up procedures**, *and specific levels of compliance. In addition, a set of* **special annotations** *is described that are intended for use by vendor-supplied and/or third-party tools to perform static off-line analysis that can ensure certain correctness properties for safety-critical applications.”*

JSR302 — Missions



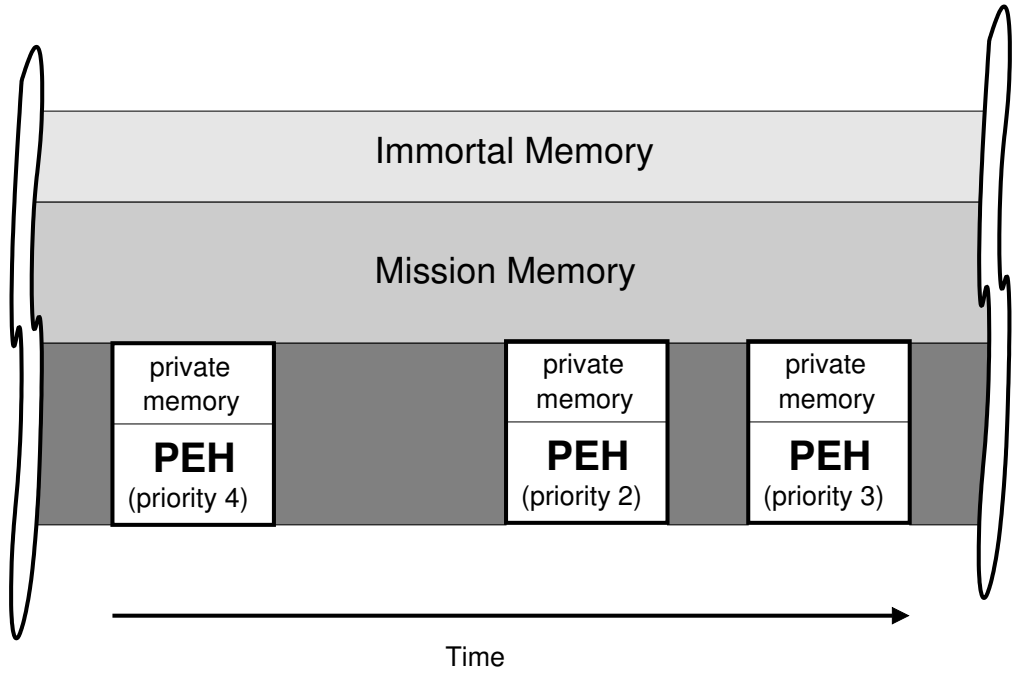
- A compliant JSR302 application consists of one or more **missions**.
- A mission consists of a set of **schedulable objects**, such as **Periodic Event Handler**.
- A schedulable object consists of a sequence of code that is scheduled by a **fixed-priority scheduler**.

JSR302 — Compliance Levels

- **Level 0** — *Mission is a set of computations executed periodically in a precise, clock-driven timeline.*
- **Level 1** — *Mission is a set of concurrent computations (with priority) running under fixed-priority preemptive scheduler.*
- **Level 2** — *Starts with single mission, but may create additional missions which run concurrently.*

Which offers greatest level of safety?

JSR302 — Mission Memory



- **Mission memory** persists for life of mission.
- Objects **allocated** into mission memory during **initialisation**.
- During **execution** applications *generally* don't **allocate new objects**!!

SCJ Annotations

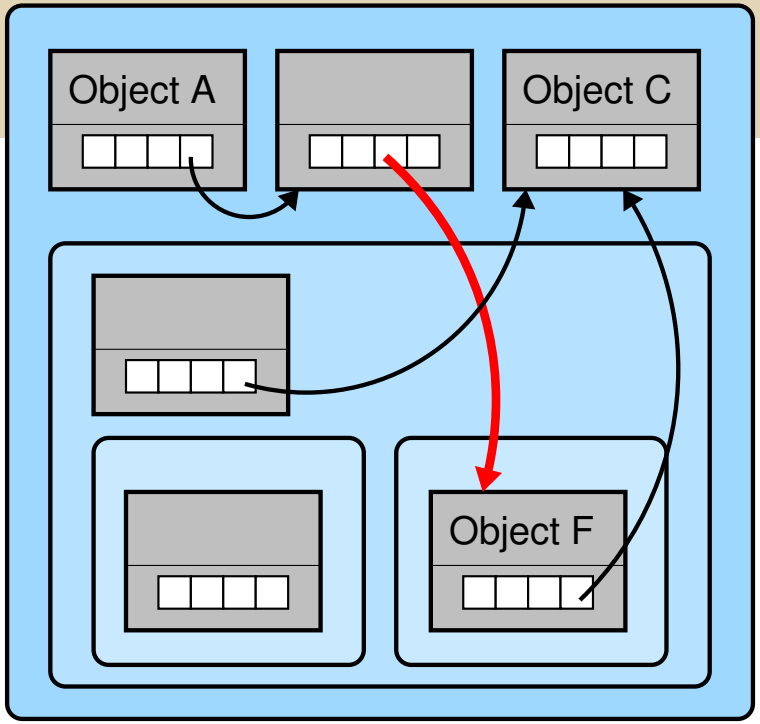
```
@SCJAllowed(0)
class Mission {

  @SCLMayAllocate(AllocationContext.THIS)
  public void initialise() {
    ...
  }
}
```

- `@SCJAllowed(level)` — minimum level at which class, interface, method or field may be used.

- `@SCJMayAllocate(context)` — only methods with this annotation may allocate memory, and only in the specified area.

Scoped Memory



- Memory organised into **scopes**.
- Scopes can **nest** inside each other.
- All memory in a scope is **deallocated** at once.
- Scopes don't need to be **garbage collected**.
- Objects in scopes with **longer lifetimes** cannot hold references to those in scopes with **shorter lifetimes**.

Class Initialisation Order

"Initialization of a class consists of executing its static initializers and the initializers for static fields (class variables) declared in the class."

–Java Language Specification

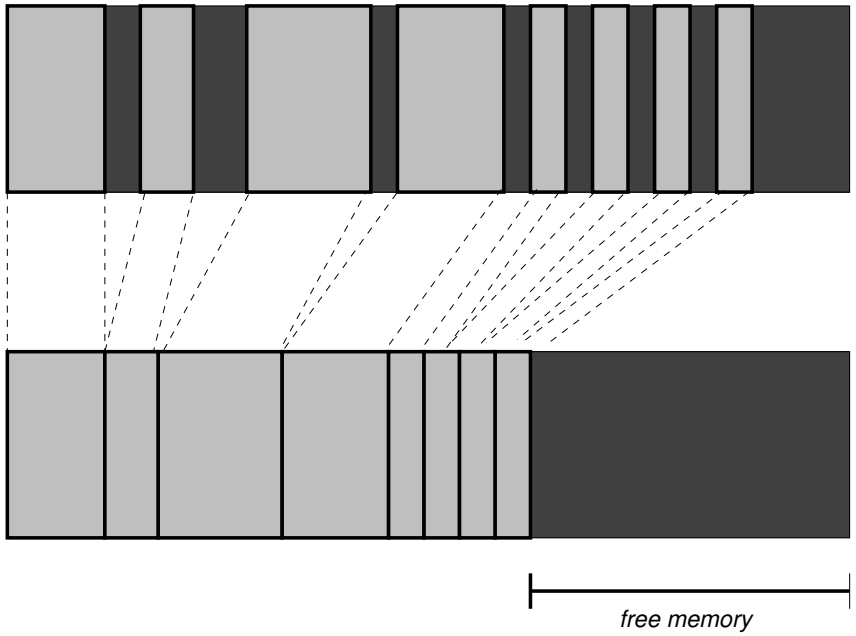
```
class X {
  static final int value = 1 + Y.value;
}
class Y {
  static final int value = 1 + X.value;
}
```

- When is a class **initialised** in Java?
- Why is above **problematic** for Safety-Critical Java?

Problematic Java Features

"One of the design goals of the SCJ specification has been to enable the development of SCJ applications that are not vulnerable to reliability failures due to **memory fragmentation***."*

–JSR302



- Memory fragmentation particularly problematic in Java. **Why?**

And, finally ...

"Today, most safety-critical software systems are developed in **either C or Ada***. Developers who choose C typically cite benefits such as* **improved availability of development tools** *and platform support, along with* **ease of recruiting** *engineers and new software engineering graduates. Today's developers who choose Ada for a new development effort typically explain that the language offers* **superior abstraction** *and* **type safety** *than C"*

- **SEE:** "DO-178C meets safety-critical Java", `http://vita.mil-embedded.com/articles/do-178c-meets-safety-critical-java/`