Chapter 1
Memory Safety for Safety Critical Java

Daniel Tang, Ales Plsek, Jan Vitek

Abstract Memory is a key resource in computer systems. Safety-critical systems often must operate for long periods of time with limited available memory. Programmers must therefore take great care to use memory sparingly and avoid programming errors. This chapter introduces the memory management API of the Safety Critical Java specification and presents a static technique for ensuring memory safety.

1.1 Introduction

Memory is a key resource in any embedded system. Programmers must carefully apportion storage space to the different tasks that require it and, when necessary, repurpose memory locations that are currently not in use. This is traditionally done by a combination of static allocation at program startup and manual management of object pools. While static allocation is possible in Java, as shown by the Java Card standard [**?**], it is at odds with object-oriented programming principles and best practices. For this reason, the Real-time Specification for Java (RTSJ) [**?**] adopted a memory management API that allows dynamic allocation of objects during program execution but gives control to programmers over the time and cost of memory management. In Java, the memory management API is exceedingly simple. The `new` keyword allocates data in the heap and, before running out of memory, a *garbage collection* algorithm is invoked to reclaim objects that are unreferenced. Before reclaiming data, `finalize()` methods are called to clean up external state. This API together with other features in the Java programming language, such as array bounds checking, provides a property referred to as *memory safety*. This property ensures that a Java program will never access a memory location that it has not been given access to and will never access a memory location after that location has been freed. Thus, common software faults such as dangling pointers and buffer overflow are guaranteed to never occur in Java programs. The RTSJ eschews garbage collec-

tion in favor of a more complex API based on the notion of scoped memory areas, regions of memory which are not subject to garbage collection and which provide a pool of memory that can be used to allocate objects. A scoped memory area is active while one or more threads are evaluating calls to the memory area's `enter()` method. Objects allocated with `new` by those threads remain alive until all threads return from the `enter()` call. At this point in time, `finalize()` methods can be called and memory used by the data can be reclaimed. A hierarchy of scopes can be created at run-time allowing programmers to finely tune the lifetime of their data. This form of memory management is reminiscent of stack allocation in languages like C or C++. But, whereas stack allocation is inherently unsafe, the RTSJ mandates dynamic checks to ensure memory safety. This means that every assignment statement to a reference variable will be checked for safety, and if the assignment could possibly lead to the program following a dangling pointer, an exception will be thrown.

Safety-critical applications must undergo a rigorous validation and certification process. The proposed Safety Critical Java (SCJ) specification[1] is being designed to facilitate the certification of real-time Java applications under safety-critical standards such DO-178B in the US [**?**]. To ease the task of certification, SCJ reduces the complexity of the RTSJ memory management interface by removing some methods and classes and forbidding some usage patterns that may lead to errors. Furthermore, SCJ provides an optional set of Java metadata annotations which can be used to check memory safety at compile-time. SCJ program must perform the same dynamic checks as with RTSJ, but a fully annotated program is guaranteed to never throw a memory access exception, thus making it possible to optimize those checks away. It is important to observe that memory safety does not imply the absence of all memory errors such as, for instance, that the program will not run out of memory. This is a separate and orthogonal issue. We expect that other, independently developed, tools will provide static memory usage checks [**?**].

The RTSJ memory management API has proven rather controversial. Over the years, most vendors have offered real-time garbage collectors as a way to return to the simple Java memory API while bounding pause times. Unfortunately, real-time garbage collection slows down program execution and requires additional resources that are not always available in small devices. The main drawback of RTSJ-style memory management is that any reference assignment must be treated with suspicion as it may lead to an `IllegalAssignmentError` being thrown. Researchers have proposed disciplined uses of the API to reduce the likelihood of error [**?**, **?**, **?**, **?**]. However, users have continued to view scoped memory as hard to use correctly. SCJ annotations are intended as a way to help programmers structure their program to simplify reasoning about allocation contexts. Memory safety annotations are added to the Java source code to provide additional information to both developers and the Java Virtual Machine. The SCJ specification also defines metadata annotations that indicate behavioral characteristics of SCJ applications.

[1] JCP JSR-302, http://www.jcp.org/en/jsr/detail?id=302

For example, it is possible to specify that a method performs no allocation or self-suspension. We refer the interested reader to the specification for more details.

This chapter introduces the SCJ memory management API and describes a set of Java metadata annotations that has been proposed as a solution for ensuring static memory safety of real-time Java programs written against the SCJ memory API. We give examples of simple SCJ programming patterns and discuss how to validate them statically.

1.2 Scoped Memory and Safety Critical Java

The memory management API exposed by SCJ is a simplification of that of the RTSJ. A proper introduction to the RTSJ is outside of the scope of this chapter; we refer interested readers to [?, ?]. Explaining the memory management API requires some understanding of the main concepts of SCJ[2]; therefore we start with an overview.

1.2.1 SCJ Overview

A SCJ compliant application consists of one or more *missions*, executed concurrently or in sequence. Every application is represented by an implementation of the `Safelet` class which arranges for running the sequence of `Missions` that comprise the application. A mission consists of a bounded set of event handlers and possibly some RTSJ threads, known collectively as *schedulable objects*. For each mission, a dedicated block of memory is identified as the *mission memory*. Objects created in mission memory persist until the mission is terminated. All classes are loaded into a block of immortal memory when the system starts up. Conforming implementations are not required to support dynamic class loading, so all the code of a mission is expected to be loaded and initialized at startup. Each mission starts in an initialization mode during which objects may be allocated in mission memory. When a mission's initialization has completed, execution mode is entered. During execution mode, mutable objects residing in mission or immortal memory may be modified as needed. All application processing for a mission occurs in one or more schedulable objects. When a schedulable object is started, its initial memory area is a private memory area that is entered when the schedulable object is released and exited (and cleared) at the end of the release. By default, objects are allocated in this private memory which is not shared with other schedulable objects. A mission can be terminated. Once termination is requested, all schedulable objects in the mission are notified to cease operating. Once they have all stopped, the mission can cleanup before it terminates. This provides a clean restart or transition to a new mission.

[2] This chapter is based on SCJ v0.71 from August 2010.

The complexity of safety-critical software varies. At one end of the spectrum, some applications support a single function with only simple timing constraints. At the other end, there are complex, multi-modal systems. The cost of certification of both the application and the infrastructure is sensitive to their complexity, so enabling the construction of simpler systems is highly desirable. SCJ defines three compliance levels to which both implementations and applications may conform. SCJ refers to them as *Level 0, Level 1*, and *Level 2*. Level 0 refers to the simplest applications and Level 2 refers to the more complex applications. A Level 0 application's programming model is a familiar model often described as a timeline model, a frame-based model, or a cyclic executive model. In this model, the mission can be thought of as a set of computations, each of which is executed periodically in a precise, clock-driven timeline, and processed repetitively throughout the mission. A Level 0 application's schedulable objects consist only of a set of periodic event handlers (PEH). Each event handler has a period, priority, and start time relative to the beginning of a major cycle. A schedule of all PEHs is constructed by the application designer. A Level 1 application uses a programming model consisting of a single mission with a set of concurrent computations, each with a priority, running under control of a fixed-priority preemptive scheduler. The computation is performed in a mix of periodic and aperiodic event handlers. A Level 1 application shares objects in mission memory among its schedulable objects, using synchronized methods to maintain the integrity of its shared objects. A Level 2 application starts with a single mission, but may create and execute additional missions concurrently with the initial mission. Computation in a Level 2 mission is performed in a combination of event handlers and RTSJ threads. Each child mission has its own mission sequencer, its own mission memory, and may also create and execute other child missions.

The `Safelet` interface is shown in Fig. 1.1. The `@SCJAllowed` annotation indicates that this interface is available at all compliance levels. `Safelet` methods are invoked by the infrastructure (i.e. the virtual machine) and can not be called from user-defined code; such methods are annotated with `@SCJAllowed(SUPPORT)`. The infrastructure invokes in sequence `setUp()` to perform any required initialization actions, followed by `getSequencer()` to return an object which will create the missions that comprise the application. The missions are run in an independent thread while the safelet waits for that thread to terminate its execution. Upon termination, the infrastructure invokes `tearDown()`. The `Mission` class, shown in Fig. 1.2, is `abstract`; it is up to subclasses to implement the `initialize()` method, which performs all initialization operations for the mission. The `initialize()` method is called by the infrastructure after memory has been allocated for the mission. The main responsibility of `initialize()` is to create and register the schedulable objects that will implement the behavior of the mission. The `cleanUp()` method is called by the infrastructure after all schedulable objects associated with this mission have terminated, but before the mission memory is reclaimed. The infrastructure arranges to begin executing the registered schedulable objects associated with a particular `Mission` upon return from the `initialize()` method. The `CyclicExecutive` class, shown in Fig. 1.3, is used in Level 0 to combine the functionality of `Safelet` and `Mission`. Finally,

we have the abstract class `PeriodicEventHandler` in Fig. 1.4, which extends `ManagedEventHandler`, which implements the `Schedulable` interface. Instances of this class are created in the `initialize()` method of `Mission`; the method `register()` is called to add them to the current mission. The infrastructure will call the `handleAsyncEvent()` method periodically following the `PeriodicParameters` passed in as argument. Users must subclass `PeriodicEventHandler` to provide an implementation for `handleAsyncEvent()`.

```
@SCJAllowed public interface Safelet
  @SCJAllowed(SUPPORT) @SCJRestricted(phase=INITIALIZATION)
  MissionSequencer getSequencer()
  @SCJAllowed(SUPPORT) @SCJRestricted(phase=INITIALIZATION)
  void setUp()
  @SCJAllowed(SUPPORT) @SCJRestricted(phase=CLEANUP)
  void tearDown()
```

Fig. 1.1 Interface javax.safetycritical.Safelet.

```
@SCJAllowed public abstract class Mission
  @SCJAllowed(SUPPORT)
  protected abstract void initialize()
  @SCJAllowed
  public static Mission getCurrentMission()
  @SCJAllowed
  public final void requestTermination()
  @SCJAllowed(SUPPORT)
  protected void cleanUp()
```

Fig. 1.2 Class javax.safetycritical.Mission.

```
@SCJAllowed public class CyclicExecutive extends Mission implements Safelet
  @SCJAllowed
  public CyclicExecutive(PriorityParameters p, StorageParameters s)
```

Fig. 1.3 Class javax.safetycritical.CyclicExecutive.

```
@SCJAllowed
public abstract class PeriodicEventHandler extends ManagedEventHandler
  @SCJAllowed @SCJRestricted(phase=INITIALIZATION)
  public PeriodicEventHandler(PriorityParameters pp,
     PeriodicParameters r, StorageParameters sp)
  @SCJAllowed @SCJRestricted(phase=INITIALIZATION)
  public final void register()
  @SCJAllowed(SUPPORT)
  public abstract void handleAsyncEvent()
  @SCJAllowed(SUPPORT) @SCJRestricted(phase=CLEANUP)
  public void cleanUp()
```

Fig. 1.4 Abstract Class javax.safetycritical.PeriodicEventHandler.

1.2.2 Memory Management Interface

SCJ segments memory into a number of scoped memory areas, or *scopes*. Each scope is represented by an object, an instance of one of the subclasses of `MemoryArea` and a *backing store*, a contiguous block of memory that can be used for allocation. Scopes are related by a *parenting* relation that reflects their order of creation and the lifetime of the objects allocated within them. Child scopes always have a strictly shorter lifetime than their parent scope. A distinguished scope, called *immortal memory*, is the parent of all scopes, represented by a singleton instance of the `ImmortalMemory` class. Each mission has a scope called the *mission memory*, which stores objects that are need for the whole mission; this is represented by an instance of `MissionMemory` allocated in its own backing store. Each schedulable object has its own *private memory* represented by an instance of `PrivateMemory`. SCJ supports nested missions as well as nested private memories. Private memories are the default allocation context for the logic of the schedulable object to which they belong. As the name suggests, private memories are inaccessible to anything outside of the schedulable object that owns them. This is unlike RTSJ scopes, which may be accessed by multiple threads simultaneously.

Fig. 1.5 shows the subset of the RTSJ `MemoryArea` class which is allowed in SCJ. This class is the root of the memory area class hierarchy and it defines much of the memory management interface. The `executeInArea()` method accepts a `Runnable` and executes it in a memory area represented by the receiver object. The `getMemoryArea()` method returns the scope in which the argument was allocated. The `newInstance()` and `newArray()` methods allocate objects and arrays in the memory area represented by the receiver object. The `newArrayInArea()` allocate arrays in the memory area referenced by the object passed as the first argument. `memoryConsumed()` and `size()` return the number of bytes currently allocated in the backing store and the total size of the backing store of the receiver. Fig. 1.6 presents the `ImmortalMemory` class which has a single static method `instance()` to return the singleton instance of the class. The size of the immortal memory is given at startup as an argument to the JVM. Fig. 1.7 lists the methods of the `ManagedMemory` class which extends `LTMemory` (it is an RTSJ class that extends `MemoryArea` and guarantees linear time allocation; not shown here). The class introduces `getCurrentManagedMemory()` to return the instance of `ManagedMemory` used to allocate objects at the point of call, and `enterPrivateMemory()` to create a nested `PrivateMemory` area and execute the `run()` method of the `Runnable` argument in that memory area. Fig. 1.8 lists the methods of `MissionMemory` which extends `ManagedMemory`. The class has two methods, `getPortal()` and `setPortal()`, that provide means of passing information between Schedulable objects in a memory area, thus implementing a concept of a shared object. Fig. 1.9 shows `PrivateMemory`, the other subclass of `ManagedMemory`, which introduces no new methods. Neither of these classes can be created directly from user-defined code. Fig. 1.10 shows the `StorageParameters` interface which is used to reserve memory for the backing stores. It is passed as a parameter to the constructor of mission sequencers and

schedulable objects. The class can be used to tune vendor specific features such as the native and Java call stack sizes, and the number of bytes dedicated to message associated with exceptions and backtraces.

```
@SCJAllowed public abstract class MemoryArea implements AllocationContext
 @SCJAllowed
 public void executeInArea(Runnable logic) throws InaccessibleAreaException
 @SCJAllowed
 public static MemoryArea getMemoryArea(Object object)
 @SCJAllowed
 public Object newInstance(Class type)
   throws InstantiationException, InaccessibleAreaException
 @SCJAllowed
 public Object newArray(Class type, int size)
 @SCJAllowed
 public static Object newArrayInArea(Object object, Class type, int size)
 @SCJAllowed
 public abstract long memoryConsumed()
 @SCJAllowed
 public abstract long size()
```

Fig. 1.5 Abstract Class javax.realtime.MemoryArea.

```
@SCJAllowed public final class ImmortalMemory extends MemoryArea
 @SCJAllowed
 public static ImmortalMemory instance()
```

Fig. 1.6 Class javax.realtime.ImmortalMemory.

```
@SCJAllowed public abstract class ManagedMemory extends LTMemory
 @SCJAllowed
 public static ManagedMemory getCurrentManagedMemory()
 @SCJAllowed
 public void enterPrivateMemory(long size, Runnable logic)
 @SCJAllowed
 public static boolean allocatedInParent(Object c, Object p)
 @SCJAllowed
 public static boolean allocatedInSame(Object c, Object p)
```

Fig. 1.7 Abstract Class javax.safetycritical.ManagedMemory.

```
@SCJAllowed public class MissionMemory extends ManagedMemory
 @SCJAllowed
 public synchronized Object getPortal()
 @SCJAllowed
 public synchronized void setPortal(Object value)
```

Fig. 1.8 Class javax.safetycritical.MissionMemory.

```
@SCJAllowed public class PrivateMemory extends ManagedMemory
```

Fig. 1.9 Class javax.safetycritical.PrivateMemory.

```
@SCJAllowed public class StorageParameters
 @SCJAllowed
 public StorageParameters(long totalBackingStore, long nativeStackSize,
    long javaStackSize)
 @SCJAllowed
 public StorageParameters(long totalBackingStore, long nativeStackSize,
    long javaStackSize, int messageLength, int stackTraceLength)
```

Fig. 1.10 Class javax.safetycritical.StorageParameter.

1.2.3 Semantics of Memory Management API

Any statement of a SCJ application has an *allocation context* which can be either one of immortal, mission or private memory. Immortal memory is used for allocation of classes and static variables, and is the allocation context of the `Safelet.setup()` method. Mission memory is the allocation context of the `Mission.initialize()` method. Private memory is the allocation context of the `handleAsyncEvent()` method of the `PeriodicEventHandler` class. By extension, we also refer to the scope in which an object was allocated as the allocation context of this object.

The parenting relation is a transitive, non-reflexive, relation between scopes that is defined such that `ImmortalMemory` is the parent of any `MissionMemory` of a top-level `Mission` (nested missions are parented to their enclosing `MissionMemory`). `MissionMemory` is the parent of any `PrivateMemory` associated to a schedulable object registered to the corresponding `Mission`. A nested `PrivateMemory` is parented to its directly enclosing `PrivateMemory`. In SCJ, a reference assignment `x.f = y` is valid if `ax` is the scope of `x` (obtained by `MemoryArea.getMemoryArea(x)` and `ay` is the scope of y, and either `ax == ay` or `ay` is a parent of `ax`. If a SCJ program attempts to perform an assignment that is not valid, an `IllegalAssignmentError` will be thrown. Assignments to local variables and parameters, as well as assignments to primitive types, are always valid. There are no checks for reference reads.

The SCJ memory API is designed to ensure that a `PrivateMemory` is accessible to only one schedulable object. This is not the case for `MissionMemory` and `ImmortalMemory`, both of which can be accessed by multiple threads. Some of the complexity of the RTSJ memory management goes away because creation of scopes is not under programmer control and the RTSJ's `enter()` can not be used by user code. In particular, the RTSJ has `ScopeCycleException` to cover the case where multiple threads enter the same scope from different allocation contexts. This can not occur by construction of the SCJ API.

Objects allocated within a `MissionMemory` are reclaimed when the mission is terminated, all registered schedulable objects are inactive and the mission's

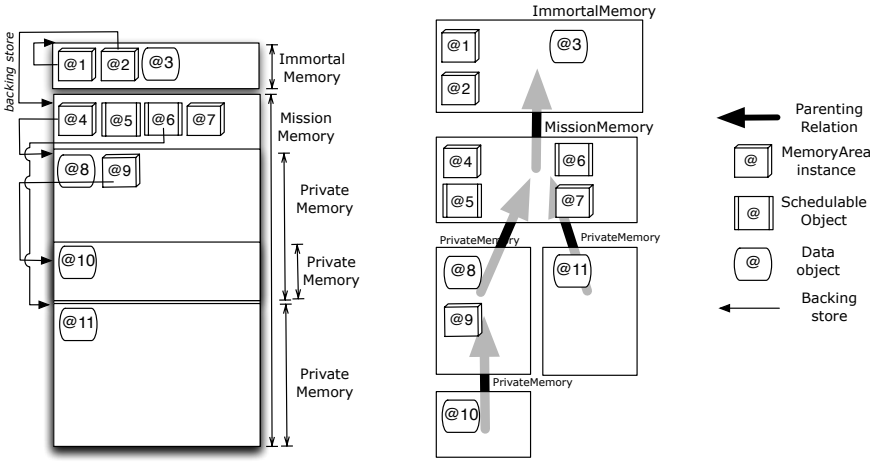


Fig. 1.11 Memory model. The figure on the left depicts the layout in memory of the objects and the figure on the right is the logical representation of the scopes with their parenting relation.

`cleanUp()` has returned. Objects allocated within a `PrivateMemory` are reclaimed with the `enterPrivateMemory()` returns, or, if this is a top-level `PrivateMemory`, when the `handleAsyncEvent()` method returns.

The method `executeInArea(runnable)` can be used to change temporarily the allocation context to another scope and execute the `run()` method of the `runnable` argument in that context. The target of the call can be an instance of `ImmortalMemory`, `MissionMemory` or `PrivateMemory` and must be a parent. However, after invoking `executeInArea()`, it is not possible to use `enterPrivateMemory()`.

The backing store for a scoped memory is taken from the backing store reservation of its schedulable object. The backing store is managed via reservations for use by schedulable objects, where the initial schedulable object partitions portions of its backing store reservation to pass on to schedulable objects created in its thread of control. Backing store size reservation is managed via `StorageParameters` objects.

The `Mission.initialize()` method call can `enterPrivateMemory()` on the mission memory and use the backing store that is reserved for its schedulable objects for temporary allocation. Once `initialize()` returns, the backing stores for all registered schedulable objects are created. Calling `enterPrivateMemory()` if the receiver is not the current allocation context will result in an `IllegalStateException`. Calling `enterPrivateMemory()` on a `MissionMemory` object after the corresponding mission's `initialize()` method has returned will result in an `IllegalStateException`. Calling `newInstance()`, `newArray()` or `executeInArea()` on a scope that is not on the current schedulable object's call stack is a programming error that results in an exception. Every schedulable object has `ImmortalMemory` and, once the mission is running, `MissionMemory` on its call stack by construction.

Fig. 1.11 provides an illustration of the memory representation of a SCJ application. The figure shows a mission consisting of two schedulable objects, one with a nested private memory area. The figure also shows the arrangement of the backing stores and highlights that schedulable objects use the backing store of their mission, and nested private memories use the backing store of their schedulable object. The parenting relation is depicted graphically as well.

1.2.4 Differences with the RTSJ

There are a number of simplifications in the SCJ memory API that lead to a reduction in possible errors. One of the most significant differences is that the RTSJ has a garbage collected heap. The RTSJ further makes the difference between `RealtimeThreads` and `NoHeapRealtimeThreads` (NHRTs). NHRTs are designed to not have to block for the garbage collector. In SCJ, all threads are NHRTs and there is no heap. We list some of the benefits of the design:

- In the RTSJ, any field access `x.f` or method invocation `x.m()` must be checked for validity. An attempt to follow a heap reference from a NHRT will result in an exception. This can not occur in SCJ as there is no heap.
- In the RTSJ, all scopes can be accessed by multiple threads. Finalizers must be run before a scope can be reused. If the thread running the finalizers is a NHRT, it may throw an exception if the finalizers manipulate heap data; if it is not, then finalization may have to wait for the garbage collection to complete, preventing other NHRTs from using the scope. This can not occur in SCJ: finalization is not supported, the last thread to leave a scope is always well defined, and there is no garbage collection.
- In the RTSJ, the parenting relation is established as a result of calling `enter()` on an unparented scope. Multiple threads may call `enter()` on the same scope from different allocation contexts, which is illegal. To prevent this, an exception is thrown if a scope that already has a parent is entered from a different allocation context. In SCJ, this can not occur for `MissionMemory` instances as they are created and entered by the infrastructure. For `PrivateMemory` instances, a runtime check is needed.

1.3 Programming with SCJ

We now illustrate how the SCJ memory API can be used for common tasks. We start with simple design patterns for pre-allocated memory and then show that per-release memory is easy to use; then we show how to fine tune the lifetime of data by choosing the right allocation context for every object. Finally, we illustrate on concrete examples errors caused by a wrong use of the SCJ memory API.

1.3.1 Static Allocation

Very conservative safety-critical systems may require pre-allocation of all of the data in advance. In SCJ, this can be achieved by allocating all data in static initializers and setting the backing stores of the mission to the minimal amount of memory (enough to hold the infrastructure-created objects like the `Mission` instance). Assume an application needs to measure release jitter in a periodic task for 1000 iterations. A first step towards this goal would be to have an event handler record the time of every periodic release. Fig. 1.12 shows such a class, `MyPEH`, which has a `handleAsyncEvent()` method that records timestamps in a statically allocated array data structure. All data manipulated by this class is allocated in static variables and no allocation will be performed in the `handleAsyncEvent()` method. To ensure prompt termination in case a developer was to accidentally allocate at run time, the `StorageParameters` request only 50 bytes as the size of the `PrivateMemory` used for `MyPEH`.

Fig. 1.12 and Fig. 1.13, which adds set up code necessary to create and start the event handler, constitute a complete Level 0 application. The class `MyApp` extends the Level 0 `CyclicExecutive` class, which merges the functionality of `Mission` and `Safelet`. Most supporting data is created in immortal memory, but the SCJ API mandates that the event handler be created in the `initialize()` method as the event handler's `register()` method (called during construction) must add the newly created handler to a mission. The `getSchedule()` method must also allocate because its argument holds a reference to the `MyPEH` instance, which is stored in mission memory. In the implementation used for this example, the size of mission memory can be limited to 500 bytes.

```
@SCJAllowed(members=true) class MyPEH extends PeriodicEventHandler {
  static int pos;
  static long[] times = new long[1000];
  static StorageParameters sp = new StorageParameters(50L,1000L,1000L);
  MyPEH() { super(null, null, sp); }
  void handleAsyncEvent() {
    times[pos++] = Clock.getRealtimeClock().getTime().getMilliseconds();
    if (pos == 1000) Mission.getCurrentMission().requestTermination();
  }
}
```

Fig. 1.12 Static allocation example.

A variant of static allocation is to have all the data needed by each mission reside in `MissionMemory`. This has the advantage that once a `Mission` is reclaimed, the space used for its objects can be safely reused for the next mission. Fig. 1.14 shows that little changes are needed to support static allocation in mission memory. Instead of using static fields, the data manipulated by the `MyPEH2` event handler resides in mission memory. To initialize the data, the allocation context of the constructor of `MyPEH2` is mission memory and therefore the array of long can be created in the constructor. For simplicity, we can reuse the set up code of Fig. 1.13 and

```
@SCJAllowed(members=true) class MyApp extends CyclicExecutive {
 static PriorityParameters p = new PriorityParameters(18);
 static StorageParameters s = new StorageParameters(500L, 1000L, 1000L);
 static RelativeTime t = new RelativeTime(5,0);
 MyApp() { super(p,s); }
 CyclicSchedule getSchedule(PeriodicEventHandler[] handlers) {
  return new CyclicSchedule(
     new CyclicSchedule.Frame[]{new CyclicSchedule.Frame(t, handlers)});
 }
 void initialize() { new MyPEH().register(); }
}
```

Fig. 1.13 Static allocation example, setup code.

```
@SCJAllowed(members=true) class MyPEH2 extends PeriodicEventHandler {
 MyPEH2() {
  super(null, null, new StorageParameters(1000L, 1000L, 1000L));
  times = new long[1000]);
 }
 int pos;
 long times[];
 void handleAsyncEvent() {
  times[pos++] = Clock.getRealtimeClock().getTime().getMilliseconds();
  if (pos == 1000) Mission.getCurrentMission().requestSequenceTermination();
 }
}
```

Fig. 1.14 Mission memory allocation example.

extend it with a new `initialize()` method that instantiates the `MyPEH2` event handler.

1.3.2 Per-release Allocation

Applications often have need of temporary storage to compute a result. In SCJ, any data allocated in the `handleAsyncEvent()` method of any event handler is temporary by default, lasting only until `handleAsyncEvent()` returns. Thus, no special programming idiom is required. Fig. 1.15 allocates an array for the duration of the release. The size of the scope is specified as an argument when the event handler is created. The application can allocate freely as long as allocated data does not overflow the backing store.

```
 void handleAsyncEvent() {
   long times[] = new long[1000];
 }
```

Fig. 1.15 Per-release allocation example.

```
long median;
void handleAsyncEvent() {
  final long times[] = new long[1000];
  Runnable r = new SCJRunnable(){
    void run() {
      long[] copy = new long[1000];
      for(int i=0;i<1000;i++) copy[i]=times[i];
      Arrays.sort(copy);
      median = copy[500];
    }
  };
  ManagedMemory m = ManagedMemory.getCurrentManagedMemory();
  m.enterPrivateMemory(8000,r);
}
```

Fig. 1.16 Per-release allocation with nested scopes.

```
@SCJAllowed(members=true) class MyPEH4 extends PeriodicEventHandler {
  Tick tock;
  void handleAsyncEvent() {
    ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(this);
    Tick time = (Tick) m.newInstance(Tick.class);
    m.executeInArea(new SCJRunnable() { void run() {
       MyPEH4.this.tock = new Tick();}});
  }
}
```

Fig. 1.17 Allocation context change in SCJ.

It is possible to have finer control over the lifetime of temporary data. Fig. 1.16 continues the previous example by creating a copy of the `times` array. That copy is allocated in a nested scope which is reclaimed when the `run()` method returns.

The SCJ memory API further provides support for flexible allocation patterns that allow users to implement different allocation strategies. The `newInstance()` creates an object in any scope that has been entered by the current thread, while `executeInArea()` allows the user to change the allocation context to a previously entered scope. Fig. 1.17 illustrates the two ways for programmers to allocate longer-lived objects. The allocation context of the `handleAsyncEvent()` method is an instance of `PrivateMemory`, but the `PeriodicEventHandler` (referenced by the `this` variable) is allocated in `MissionMemory`. The code snippet shows two ways to allocate an object in `MissionMemory`. The first way is to obtain a reference to `MissionMemory` from `this` and to call `newInstance()`. The second way is call `executeInArea()` with a `SCJRunnable` argument. In both cases an object will be allocated in `MissionMemory`. The assignment to `this.tock` is valid only because we have changed allocation context.

1.3.3 Memory Management Errors

This section illustrates the pitfalls of the SCJ memory management API with concrete examples. SCJ has two exceptions specific to memory related errors, `IllegalAssignmentError` and `IllegalStateException`. An assignment error can be thrown by any reference assignment. The challenge for developers is that inspection of the code of a single method is often insufficient to rule out errors. Consider the following code fragment.

```
void setTail(List from, List to) { from.tail = to; }
```

As there is no information in the program text as to where the arguments to the method were allocated, it is impossible to rule out the possibility that the assignment will throw an `IllegalAssignmentError`.

Consider the class declaration in Fig. 1.18. The instance of `List` referenced by field `a` is allocated in `MissionMemory`, whereas the instance referenced by variable `b` is allocated in the allocation context of `handleAsyncEvent()`, namely `PrivateMemory`. The first call to `setTail()` is valid as it will set a reference from an object allocated in a child scope (`@3`) to an object allocated in a parent scope (`@2`). The second call will throw an exception as the direction of the reference is reversed. The assignment `this.a=b` is also invalid. The `this` variable refers to the event handler (`@1`) allocated in `MissionMemory`.

```
class Err extends PeriodicEventHandler {
  List a = new List();
  ...
  void handleAsyncEvent() {
    List b = new List();
    setTail(b, a);
    setTail(a, b);
    this.a = b;
  }
}
```

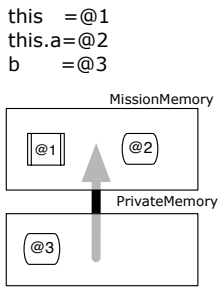


Fig. 1.18 IllegalAssignment error.

The `IllegalStateException` can be thrown if a schedulable object attempts to use a memory area that it has not entered for one of `newInstance()`, `executeInArea()` or `enterPrivateMemory()`. The SCJ specification makes this kind of error rather unlikely, but still possible. The program of Fig. 1.19 shows the steps needed for the error to occur. Handler `P1` stores a reference to its own `PrivateMemory` (`@1`) into a field of the mission `M1.pri`. Another handler, `P2`, tries to either allocate from `P1`'s memory or enter it. In both cases an exception will be thrown.

```
class M1 extends Mission {
  void initialize() { new P1().register(); new P2().register(); }
  PrivateMemory priv;
}
class P1 extends PeriodicEventHandler {
  void handleAsyncEvent() {
    M1 m1 = (M1) Mission.getCurrentMission();
    m1.priv = (PrivateMemory)
       MemoryArea.getMemoryArea(new int[0]);
  }
}
class P2 extends PeriodicEventHandler {
  void handleAsyncEvent() {
    M1 m1 = (M1) Mission.getCurrentMission();
    m1.priv.newInstance(List.class);
    m1.priv.enterPrivateMemory(500,
       new Runnable(){void run(){}});
  }
}
```

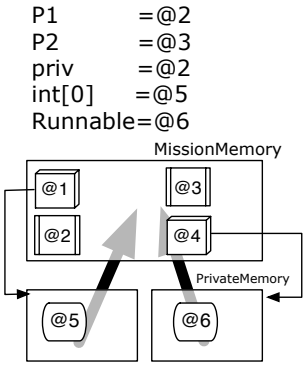
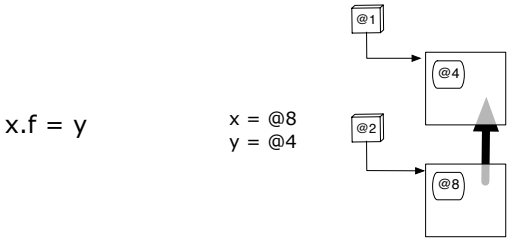


Fig. 1.19 IllegalState exceptions.

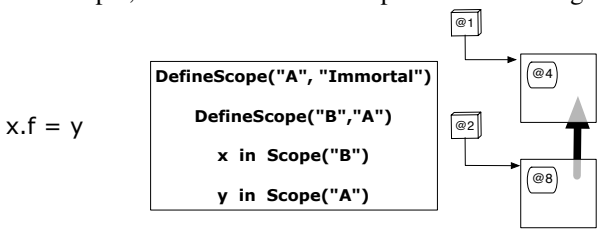
1.4 Static Memory Safety with Metadata Annotations

The SCJ specification ensures memory safety through dynamic checks. While all assignments that may lead to a dangling pointer will be caught, they are only caught at runtime and can be hard to detect through testing alone. The problem for verification of SCJ code is that the information needed to check an assignment statement is not explicit in the program text. Consider an assignment statement such as:



The steps that the infrastructure must take to ascertain that the assignment is valid are: (*i*) obtain the scope in which the object referenced by `x` is allocated (the scope `@2` here), (*ii*) obtain the scope in which the object referenced by `y` (scope `@1`) is allocated, and (*iii*) check that `@1` is a parent of `@2`. To do this at compile time, one either needs to perform a whole-program points to analysis to discover the sets of potential objects referenced by `x` and `y` and their scopes, or ask for more information to be provided by the developer. A checker needs these three pieces of information: the respective scopes of the source and target of each assignment statement and the parenting relation between the scopes. Of course, it would be rather impractical if the developer had to provide memory addresses of scopes, as this would hard-wire

the program to exactly that pair of scopes. A better solution is to provide symbolic, compile-time, names for scopes and guarantee through a set of rules that the variables will refer to objects that are allocated in those named scopes. We present an annotation system that lets developer express the relationship between scopes and specify where the objects referenced by program variables are allocated. Thus, in the above example, the metadata would express the following information:



It would specify that the program has at least two scopes, with symbolic names `A` and `B`, that `B` is a child of `A`, that the object referenced by variable `x` is allocated in scope `B` and that the object referenced by `y` is allocated in `A`. Equipped with this information, a checker can validate the assignment at compile time. The challenge for the design of memory safety annotations is to balance three requirements. The annotations should be *expressive*, letting developers write the code they want naturally. The annotations should be *non-intrusive*, requiring as few annotations as possible. The annotations must be *sound*; an annotated program must not be allowed to experience a memory error. We purposefully chose to emphasize soundness and non-intrusiveness with a very lightweight set of annotations. The core of this system is a set of annotations on classes and methods. These are very lightweight but somewhat restrictive. To recover expressiveness, the system also supports dynamic guards and variable annotations.

The system differentiates between *user* code and *infrastructure* code. User code follows the restrictions outlined in this chapter and is verified by a dedicated checker tool implementing this annotation system. Infrastructure code is verified by the vendor. Infrastructure code includes the `java` and `javax` packages as well as vendor specific libraries. The infrastructure code is assumed to be correct and will not be verified by the checker.

1.4.1 Annotations Overview

The SCJ specification introduces three Java metadata annotations, listed in Table 1.1, that guarantee the absence of memory errors. This guarantee requires that all classes are successfully checked together.

The annotation checker requires the following information. Given a statement, such as `this.x=y.n()`, occurring in the body of some method `m(T y)`, the checker must know in which scope the method will be called (the *current allocation context*), the scope of the receiver (the *allocation context* of the object referenced by the `this` variable), the scope of variable `y`, the scope in which the method `n()`

Annotation	Where	Arguments	Description
@DefineScope	Any	*Name*	Define a new scope.
@Scope	Class	*Name*	Instances are in named scope.
		CALLER	Can be instantiated anywhere.
	Field	*Name*	Object allocated in named scope.
		UNKNOWN	Allocated in unknown scope.
		THIS	Allocated enclosing class' scope.
	Method	*Name*	Returns object in named scoped.
		UNKNOWN	Returns object in unknown scope.
		CALLER	Returns object in caller's scope.
		THIS	Returns object in receiver's scope.
	Variable	*Name*	Object allocated in named scope.
		UNKNOWN	Object in an unknown scope.
		CALLER	Object in caller's scope.
		THIS	Object in receiver's scope.
@RunsIn	Method	*Name*	Method runs in named scope.
		CALLER	Runs in caller's scope.
		THIS	Runs in receiver's scope.

Table 1.1 Annotation summary. Default values in bold.

expects to execute, and the scope in which its result will be allocated. The situation is illustrated in Fig. 1.20.

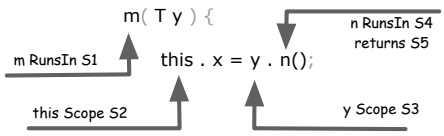


Fig. 1.20 A statement and associated scope information.

The @DefineScope Annotation. The first step is to define a *static scope tree*. The static scope tree is the compile-time representation of the run-time parenting relation. For this, we introduce the `@DefineScope` annotation with two arguments, the symbolic name of the new scope and of its parent scope. The checker will ensure that the annotations define a well formed tree rooted at `IMMORTAL`, the distinguished parent of all scopes. Scopes are introduced by mission (`MissionMemory`) and schedulable objects (`PrivateMemory`). Thus we require that each declaration of a class that has an associated scope (for instance, subclasses of the `MissionSequencer` class which define missions, and subclasses of the `PeriodicEventHandler` class which hold task logic) must be annotated with a scope definition annotation. Furthermore, nested `PrivateMemory` scopes are created by invocation of the `enterPrivateMemory()` method. As Java does not allow annotation on expression, we require that the argument of the method, an instance of a subclass of `SCJRunnable` be annotated with a scope definition.

The @Scope Annotation. The key requirement for being able to verify a program is to have a compile-time mapping of every object reference to some node in the static scope tree. With that information, verification is simply a matter of checking

that the right hand side of any assignment is mapped to a parent (or same) scope of the target object. This mapping is established by the `@Scope` annotation which takes a scope name as argument. Scope annotations can be attached to class declarations to constrain the scope in which all instances of that class are allocated. Annotating a field, local or argument declaration constrains the object referenced by that field to be in a particular scope. Lastly, annotating a method declaration constrains the value returned by that method.

Scope CALLER, THIS, and UNKNOWN. While a general form of parametric polymorphism for scopes such as full-fledged Java generics [**?**] was felt to be too complex by the SCJ expert group, we introduced a limited form of polymorphism that seems to capture many common use cases. Polymorphism is obtained by adding the following scope variables: `CALLER`, `THIS` and `UNKNOWN`. These can be used in `@Scope` annotations to increase code reuse. A reference that is annotated `CALLER` is allocated in the same scope as the "current" or calling allocation context. References annotated `THIS` point to objects allocated in the same scope as the receiver (i.e. the value of `this`) of the current method. Lastly, `UNKNOWN` is used to denote unconstrained references for which no static information is available. Classes may be annotated `CALLER` to denote that instances of the class may be allocated in any scope.

The @RunsIn Annotation. To determine the scope in which an allocation expression `new C()` is executed we need to associate methods with nodes in our static scope tree. The `@RunsIn` annotation does this. It takes as an argument the symbolic name of the scope in which the method will be executed. An argument of `CALLER` indicates that the method is scope polymorphic and that it can be invoked from any scope. In this case, the arguments, local variables, and return value are by default assumed to be `CALLER`. If the method arguments or returned value are of a type that has a scope annotation, then this information is used by the Checker to verify the method. If a variable is labeled `@Scope(UNKNOWN)`, the only methods that may be invoked on it are methods that are labeled `@RunsIn(CALLER)`.

An overriding method must preserve and restate any `@RunsIn` annotation inherited from the parent. `@RunsIn(THIS)` denotes a method which runs in the same scope as the receiver.

Default Annotation Values. To reduce the annotation burden for programmers, annotations that have default values can be omitted from the program source. For class declarations, the default value is `CALLER`. This is also the annotation on `Object`. This means that when annotations are omitted classes can be allocated in any context (and thus are not tied to a particular scope). Local variables and arguments default to `CALLER` as well. For fields, we assume by default that they infer to the same scope as the object that holds them, i.e. their default is `THIS`. Instance methods have a default `@RunsIn(THIS)` annotation.

Static Fields and Methods. This paragraph describes the rules for static fields and methods, we talk about methods/fields on other places but we assume that they are not static by default. The allocation context of static constructors and static fields is `IMMORTAL`. Thus, static variables follow the same rules as if they were explicitly annotated with `IMMORTAL`. Static methods are treated as being annotated `CALLER`.

Strings constants are statically allocated objects and thus should be implicitly `IMMORTAL`. However, this prevents users from assigning a string literal to a local variable even though the string literal is immutable. Therefore, we chose to treat these strings as `CALLER` so they may be safely assigned to local variables.

Dynamic Guards. Dynamic guards are our equivalent of dynamic type checks. They are used to recover the static scope information lost when a variable is cast to `UNKNOWN`, but they are also a way to side step the static annotation checks when these prove too constraining. We have found that having an escape hatch is often crucial in practice. A dynamic guard is a conditional statement that tests the value of one of two pre-defined methods, `allocatedInSame()` or `allocatedInParent()` or, to test the scopes of a pair of references. If the test succeeds, the check assumes that the relationship between the variables holds. The parameters to a dynamic guard are local variables which must be `final` to prevent an assignment violating the assumption. The following example illustrates the use of dynamic guards.

```
void method(@Scope(UNKNOWN) final List unk, final List cur) {
  if (ManagedMemory.allocatedInSame(unk, cur)) {
    cur.tail = unk;
  }
 }
```

The method takes two arguments, one `List` allocated in an unknown scope, and the other allocated in the current scope. Without the guard the assignment statement would not be valid, since the relation between the objects' allocation contexts can not be validated statically. The guard allows the checker to assume that the objects are allocated in the same scope and thus the method is deemed valid. Note that the parameters to `allocatedInSame()` and `allocatedInParent()` must be `final`, so that the variables cannot be modified to violate the assumption.

Arrays. Arrays are another feature that requires special treatment. By default, the allocation context of an array `T[]` is the same as that of its element class, `T`. Primitive arrays are considered to be labeled `THIS`. The default can be overriden by adding a `@Scope` annotation to an array variable declaration.

1.4.1.1 Scope Inference

The value of polymorphic annotations such as `THIS` and `CALLER` can be inferred from the context in certain cases. A concretization function (or scope inference) translates `THIS` or `CALLER` to a named scope. For instance a variable annotated `THIS`

takes the scope of the enclosing class (which can be `CALLER` or a named scope). An object returned from a method annotated `CALLER` is concretized to the value of the calling method's `@RunsIn` which, if it is `THIS`, can be concretized to the enclosing class' scope. We say that two scopes are the same if they are identical after concretization.

The concretization is used to determine the allocation context for an expression. In an ideal situation, where every class is annotated with `@Scope`, it is easy to determine the scope of an expression simply by examining the `@Scope` annotation for the type of the expression. However, unannotated types and `@Scope`-annotated variables complicate the situation. For example, suppose a method declares a new `Integer` object:

```
Integer myInt = new Integer();
```

It must be possible to know in which scope `myInt` resides. In the general case, it can be assumed to be `@Scope(THIS)`, since use of the `new` operator is, by definition, in the current allocation context; however, if the method has a `@RunsIn` annotation which names a specific scope `"a"`, then it is more precise to state that `myInt` is `@Scope("a")`.

It is important to infer a scope for every expression to ensure that an assignment is valid. Since a field may only refer to an object in the same scope or a parent scope, statically knowing the scope of every expression that is assigned to the field makes it possible to determine whether or not the assignment is actually legal.

Local variables, unlike fields and parameters, may have no particular scope associated with them when they are declared and are of a type that is unannotated. Scope inference is also used to bind the variable to the scope of the right-hand side expression of the first assignment. In the above example, if the containing method is `@RunsIn(CALLER)`, `myInt` is bound to `@Scope(CALLER)` while the variable itself is still in lexical scope. In other words, it is as if `myInt` had an explicit `@Scope(CALLER)` annotation on its declaration. It would be illegal to have the following assignment later in the method body:

```
myInt = Integer.MAX_INT;
```

It is intuitive to derive the scope of other, more complex expressions as well. Method calls that have a `@Scope` annotation or are of an annotated type take on the specific scope name of the annotation. If there is no `@Scope` annotation, then the method is assumed to return a newly allocated object in the current scope.

The scope of a field access expression may depend on the scope of the object itself. For example, if we have a field access `exp.f`, if the type of `f` and the declaration of `f` have no `@Scope` annotation, then the scope of `exp.f` is the same as the scope of `exp`. Note that `exp` can represent any expression, not just a variable.

1.4.1.2 Memory Safety Rules

We will now review the constraints imposed by the checker.

Overriding Annotations. Subclasses must preserve annotations. A subclass of a class annotated with a named scope must retain the exact same scope name. A subclass of a class annotated `CALLER` may override this with a named scope. Method annotations must be retained in subclasses to avoid upcasting an object to the supertype and executing the method in a different scope.

A method invocation `z=x.m(...,y,...)` is valid (1) if its `@RunsIn` is the same as the current scope or it is annotated `@RunsIn(CALLER)`, (2) if the scope of every argument `y` is the same as the corresponding argument declaration or if argument is `UNKNOWN`, (3) if the scope of the return value is the same as `z`.

An assignment expression `x.f=y` is valid if one of the following holds: (1) `x`.f and `y` have the same scope and are not `UNKNOWN` or `THIS`, (2) `x`.f has scope `THIS` and `x` and `y` has the same, non-`UNKNOWN` scope, or (3) `x`.f is `THIS`, `f` is `UNKNOWN` and the expression is protected by a dynamic guard.

A cast expression `(C) exp` may refine the scope of an expression from an object annotated with `CALLER`, `THIS`, or `UNKNOWN` to a named scope. For example, casting a variable declared `@Scope(UNKNOWN) Object` to `C` entails that the scope of expression will be that of `C`. Casts are restricted so that no scope information is lost.

An allocation expression `new C()` is valid if the current allocation context is the same as that of the class `C`. A variable or field declaration, `C x`, is valid if the current allocation context is the same or a child of the allocation context of `C`. Consequently, classes with no explicit `@Scope` annotation cannot reference classes which are bound to named scopes, since `THIS` may represent a parent scope.

1.4.1.3 Additional Rules and Restrictions of the Annotation System

The SCJ memory safety annotation system further dictates a following set of rules specific to SCJ API methods.

MissionSequencer and Missions. The `MissionSequencer` must be annotated with `@DefineScope`, its `getNextMission()` method has a `@RunsIn` annotation corresponding to this newly defined scope. Every `Mission` associated with a particular `MissionSequencer` is instantiated in this scope and they must have a `@Scope` annotation corresponding to that scope.

Schedulables. Each `Schedulable` must be annotated with a `@DefineScope` and `@Scope` annotation. There can be only one instance of a `Schedulable` class per `Mission`.

MemoryArea Object Annotation. The annotation system requires every object representing a memory area to be annotated with `@DefineScope` and `@Scope` an-

notations. The annotations allow the checker to statically determine the scope name of the memory area represented by the object. This information is needed whenever the object is used to invoke `MemoryArea` and `ManagedMemory` API methods, such as `newInstance()` or `executeInArea()` and `enterPrivateMemory()`. The example in Fig. 1.21 demonstrates a correct annotation of a `ManagedMemory` object `m`.

```
@Scope("M") @DefineScope(name="H", parent="M")
class Handler extends PeriodicEventHandler {
   @RunsIn("H") @SCJAllowed(SUPPORT)
   void handleAsyncEvent() {
      @Scope(IMMORTAL)
      @DefineScope(name="M", parent=IMMORTAL)
      ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(this);
      ...
   }
}
```

Fig. 1.21 Annotating `ManagedMemory` object example.

The example shows a periodic event handler instantiated in memory `M` that runs in memory `H`. Inside the `handleAsyncEvent()` method we retrieve a `ManagedMemory` object representing the scope `M`. As we can see, the variable declaration is annotated with `@Scope` annotation, expressing in which scope the memory area object is allocated – in this case it is the `IMMORTAL` memory. Further, the `@DefineScope` annotation is used to declare which scope is represented by this instance.

executeInArea(). Calls to a scope's `executeInArea()` method can only be made if the scoped memory is a parent of the current allocation context. In addition, the `SCJRunnable` object passed to the method must have a `@RunsIn` annotation that matches the name of the scoped memory. This is a purposeful limitation of what SCJ allows, since the system does not know what the scope stack is at any given point in the program.

enterPrivateMemory(). Calls to a scope memory's enterPrivateMemory(size, runnable) method are only valid if the runnable variable definition is annotated with `@DefineScope(name="x",parent="y")` where `x` is the memory area being entered and `y` is a the current allocation context. The `@RunsIn` annotation of the runnable's `run()` method must be the name of the scope being defined by `@DefineScope`.

newInstance(). Certain methods in the `MemoryArea` class encapsulate common allocation idioms. The `newArray()`, `newArrayInArea()`, and `newInstance()` methods may be used to allocate arrays and objects in a different allocation context than the current one. In these cases, invocations of these methods must be treated specially. Calls to a scope's `newInstance()` or `newArray()` methods are only valid if the class or element type of the array are annotated to be allocated in target scope or not annotated at all. Similarly, calls to `newArrayInArea()` are only

legal if the element type is annotated to be in the same scope as the first parameter or not annotated at all. The expression

```
ImmortalMemory.instance().newArray(byte.class, 10)
```

should therefore have the scope `IMMORTAL`. An invocation MemoryArea.newArrayInArea(o, byte.class, 10) is equivalent to calling MemoryArea.getMemoryArea(o).newArray(byte.class, 10). In this case, we derive the scope of the expression from the scope of `o`.

getCurrent*() methods. The `getCurrent*` methods are static methods provided by SCJ API that allow applications to access objects specific to the SCJ infrastructure. The `getCurrent*()` methods are `ManagedMemory.getCurrentManagedMemory()`, `RealtimeThread.getCurrentMemoryArea()`, `MemoryArea.getMemoryArea()`, `Mission.getCurrentMission()`, `MissionManager.getCurrentMissionManager()`, and `Scheduler.getCurrentScheduler()`. Typically, an object returned by such a call is allocated in some upper scope; however, there is no annotation present on the type of the object. To explicitly express that the allocation scope of returned object is unknown, the `getCurrent*()` methods are annotated with `@RunsIn(CALLER)` and the returned type of such a method call is `@Scope(UNKNOWN)`.

1.4.2 Impact on Standard Library Classes

The presented system was designed in part to minimize changes to the standard libraries. However, there are some patterns that the system cannot capture with full precision when using library classes across different scopes. This occurs for example when a method returns objects that live in different scopes, as illustrated in Fig. 1.22.

```
class BigInteger {
  static BigInteger ZERO = new BigInteger(0);
  BigInteger add(BigInteger o) {
    if (this == ZERO) return o;
    if (o == ZERO) return this;
    return slowAdd(this, o);
  }
}

ZERO.add(ZERO);
ZERO.add(new BigInteger(3));
```

Fig. 1.22 A library class `BigInteger` with a method returning objects in two different scopes.

The `BigInteger` class attempts to prevent unnecessary allocation when adding two objects together. If either operand is zero, the result will be the same as the other operand; since `BigInteger` objects are immutable, it is acceptable to simply return the other operand. However, with respect to SCJ, this means that `add()` can return objects in several different scopes. On the first use of `add()`, an object living

in the immortal memory is returned. However, the second addition returns the newly created object representing the value 3, which is allocated in the current allocation context that may or may not be immortal memory.

There are a few solutions to this problem. First, `add()` could be annotated to return an object in the `UNKNOWN` scope. This means that, in order to minimize the number of dynamic checks to call `BigInteger` methods, most of the methods must be labeled `@RunsIn(CALLER)`. This is safe to do because `BigInteger` objects are treated as value types and are therefore never mutated after construction, but litters the standard library class with annotations.

Another solution is to make `add()` always return a fresh object, so that the method can be implied as `@Scope(CALLER)`. This has the advantage of not requiring explicit annotations. However, the lack of `@RunsIn(CALLER)` annotations limits `BigInteger` operands and operators to living in the same scope. This both simplifies and limits how the standard library may be used.

Even though the system requires annotation of standard Java libraries, we believe that this one-time cost paid by JVM vendors is negligible in comparison to the costs of sanitizing those libraries to qualify them for safety certification and then actually gathering all of the required safety certification evidence.

1.5 Collision Detector Example

In this section we present the Collision Detector (CD*x*)[3] [**?**] example and illustrate the use of the memory safety annotations. The classes are written with a minimum number of annotations, though the figures hides much of the logic which has no annotations at all.

The CD*x* benchmark consists of a periodic task that takes air traffic radar frames as input and predicts potential collisions. The main computation is executed in a private memory area, as the CD*x* algorithm is executed periodically; data is recorded in a mission memory area. However, since the CD*x* algorithm relies on positions in the current and previous frame for each iteration, a dedicated data structure, implemented in the `Table` class, must be used to keep track of the previous positions of each airplane so that the periodic task may reference it. Each aircraft is uniquely represented by its `Sign` and the `Table` maintains a mapping between a `Sign` and a `V3d` object that represents current position of the aircraft. Since the state table is needed during the lifetime of the mission, placing it inside the persistent memory is the ideal solution.

First, a code snippet implementing the Collision Detector mission is presented in Fig. 1.23. The `CDMission` class is allocated in a scope named similarly and implicitly runs in the same scope. A substantial portion of the class' implementation is dedicated to the `initialize()` method, which creates the mission's handler and then shows how the `enterPrivateMemory()` method is used to perform some

[3] The CD*x* open-source distributions is at www.ovmj.net/cdx (Version miniCDj).

```
@DefineScope(name="M", parent=IMMORTAL)
@Scope("M") class CDMission extends Mission {
 void initialize() {
  new Handler().register();
  MIRun run = new MIRun();
  @Scope(IMMORTAL)
  @DefineScope(name="M", parent=IMMORTAL)
  ManagedMemory m = (ManagedMemory) ManagedMemory.getMemoryArea(this);
  m.enterPrivateMemory(2000, run);
 }
}

@Scope("M")
@DefineScope(name="MI", parent="M")
class MIRun implements SCJRunnable {
 @RunsIn("MI") void run() {...}
}
```

Fig. 1.23 CD*x* mission implementation.

initialization tasks in a sub-scope using the `MIRun` class. The `ManagedMemory` variable `m` is annotated with `@DefineScope` and `@Scope` to correctly define which scope is represented by this object. Further, notice the use of `@DefineScope` to define a new `MI` scope that will be used as a private memory for the runnable.

The `Handler` class, presented in Fig. 1.24, implements functionality that will be periodically executed throughout the mission in the `handleAsyncEvent()` method. The class is allocated in the `M` memory, defined by the `@Scope` annotation. The allocation context of its execution is the `"H"` scope, as the `@RunsIn` annotations upon the `Handler`'s methods suggest.

```
@DefineScope(name="H", parent="M")
@Scope("M") class Handler extends PeriodicEventHandler {
 Table st;
 @RunsIn("H") void handleAsyncEvent() {
      Sign s = ... ;
      @Scope("M") V3d old_pos = st.get(s);
      if (old_pos == null) {
        @Scope("M") Sign n_s = mkSign(s);
        st.put(n_s);
      } else ...
 }
 @RunsIn("H") @Scope("M") Sign mkSign(@Scope("M") Sign s) {
  @Scope(IMMORTAL) @DefineScope(name="M",parent="IMMORTAL")
  ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(s);

  @Scope("M") Sign n_s = ManagedMemory.newInstance(Sign.class);
  n_s.b = (byte[]) MemoryArea.newArrayInArea(s, byte.class, s.length);
  for (int i : s.b.length) n_s.b[i] = s.b[i];
  return n_s
 }
}
```

Fig. 1.24 CD*x* Handler implementation.

Consider the `handleAsyncEvent()` method, which implements a communication with the `Table` object allocated in the scope `M`, thus crossing scope bound-

aries. The `Table` methods are annotated as `@RunsIn(CALLER)` and `@Scope(THIS)` to enable this cross-scope communication. Consequently, the `V3d` object returned from a `@RunsIn(CALLER) get()` method is inferred to reside in `@Scope("M")`. For a newly detected aircraft, the `Sign` object is allocated in the `M` memory and inserted into the `Table`. This is implemented by the `mkSign()` method that retrieves an object representing the scope `M` and uses the `newInstance()` and `newArrayInArea()` methods to instantiate and initialize a new `Sing` object.

```
@Scope("M") class Table {
  final HashMap map;
  V3d vectors [];
  int counter = 0;
  final VRun r = new VRun();

  @RunsIn(CALLER) @Scope(THIS)
  V3d get(Sign s) {
    return (V3d) map.get(s);
  }

  @RunsIn(CALLER) void put(
        final @Scope(UNKNOWN) Sign s) {
    if (ManagedMemory.allocatedInSame(r,s))
      r.s = s;
    @Scope(IMMORTAL)
    @DefineScope(name="M",parent=IMMORTAL)
    ManagedMemory m = (ManagedMemory)
      MemoryArea.getMemoryArea(this);
    m.executeInArea(r);
  }
}

@Scope("M") class VRun
  implements SCJRunnable {

  Sign s;

  @RunsIn("M") void run() {
    if (map.get(s) != null) return;
    V3d v = vectors[counter++];
    map.put(s,v);
  }
}
```

```
ImmortalMemory   = @1
MissionMemory    = @2
PrivateMemory    = @3
CDMission        = @4
Handler          = @5
Table t          = @6
HashMap map      = @7
VRun r           = @8
V3d old_pos      = @9
Sign n_s         = @10
Sign s           = @11
```

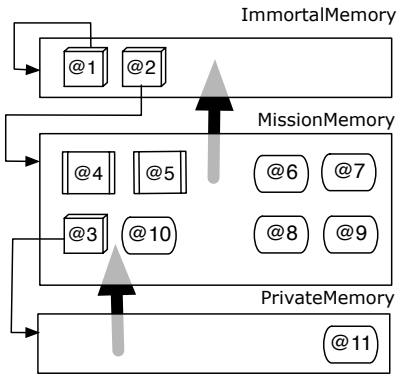


Fig. 1.25 CDx Table implementation.

The implementation of the `Table` is presented in Fig. 1.25. The figure further shows a graphical representation of memory areas in the system together with objects allocated in each of the areas. The immortal memory contains only an object representing an instance of the `MissionMemory`. The mission memory area contains the two schedulable objects of the application – `Mission` and `Handler`, an instance representing `PrivateMemory`, and objects allocated by the application itself – the `Table`, a hashmap holding `V3d` and `Sign` instances, and runnable objects used to switch allocation context between memory areas. The private memory holds temporary allocated `Sign` objects.

The `Table` class, presented in Fig. 1.25 on the left side, implements several `@RunsIn(CALLER)` methods that are called from the `Handler`. The `put()` method was modified to meet the restrictions of the annotation system, the argument

is `UNKNOWN` because themethod can potentially be called from any subscope. In the method, a dynamic guard is used to guarantee that the `Sign` object being passed as an argument is allocated in the same scope as the `Table`. After passing the dynamic guard, the `Sign` can be stored into a field of the `VectorRunnable` object. This runnable is consequently used to change allocation context by being passed to the `executeInArea()`. Inside the runnable, the `Sign` is then stored into the `map` that is managed by the `Table` class. After calling `executeInArea()`, the execution context is changed to `M` and the object `s` can be stored into the `map`. Finally, a proper `HashMap` implementation annotated with `@RunsIn(CALLER)` annotations is necessary to complement the `Table` implementation.

1.6 Related Work

The Aonix PERC Pico virtual machine introduces stack-allocated scopes, an annotation system, and an integrated static analysis system to verify scope safety and analyze memory requirements. The PERC type system [**?**] introduces annotations indicating the scope area in which a given object is allocated. A byte-code verifier interpreting the annotations proves the absence of scoped memory protocol errors. The PERC Pico annotations do not introduce absolute scopes identifiers. Instead, they emphasize scope relationships (e.g. argument `A` resides in a scope that encloses the scope of argument `B`). This allows more generic reuse of classes and methods in many different scopes, rather than requiring duplication of classes for each distinct scope context at the cost of a higher annotation burden. The PERC annotations address sizing requirements which are not considered here.

The authors of [**?**] proposed a type system for Real-Time Java. Although the work is applied to a more general scenario of RTSJ-based applications, it shows that a type system makes it possible to eliminate runtime checks. In comparison to the approach in this chapter, the proposed type system provides a richer but a more complex solution.

Scoped Types [**?**, **?**] introduce a type system for RTSJ which ensures that no runtime errors due to memory access checks will occur. Furthermore, Scoped Types capture the runtime hierarchy of scopes and subscopes in the program text by the static hierarchy of Java packages and by two dedicated Java annotations. The authors demonstrates that it is possible to statically maintain the invariants that the RTSJ checks dynamically, yet syntactic overhead upon programmers is small. The solution presented by the authors is a direct ancestor of the system described by this chapter.

1.7 Conclusion

This chapter has presented the SCJ memory management API which provides increased safety by simplifying the memory model it inherited from the RTSJ. Furthermore, we have presented a set of metadata annotations which can be used to statically prove that SCJ compliant programs are free of memory errors caused by illegal assignments. This greatly increases the reliability of safety critical applications and reduce the cost of certification.

Acknowledgements The author thanks the JSR-302 expert group (Doug Locke, B. Scott Andersen, Ben Brosgol, Mike Fulton, Thomas Henties, James Hunt, Johan Nielsen, Kelvin Nilsen, Martin Schoeberl, Joyce Tokar, Andy Wellings) for their work on the SCJ specification and their input and comments on the memory safety annotations presented in this chapter.

References

1. A. Corsaro, C. Santoro. The Analysis and Evaluation of Design Patterns for Distributed Real-Time Java Software. *16th IEEE International Conference on Emerging Technologies and Factory Automation*, 2005.
2. C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped types and aspects for real-time Java memory management. *Realtime Systems Journal*, 37(1), 2007.
3. E. Benowitz and A. Niessner. A patterns catalog for RTSJ software designs. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, pages 497–507, 2003.
4. G. Bollella, T. Canham, V. Carson, V. Champlin, D. Dvorak, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reinholtz. Programming with non-heap memory in the real time specification for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.
5. G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull****This is probably incorrect*****. The real-time specification for Java 1.0.2. Available at: http://www.rtsj.org/specjavadoc/book_index.html.
6. C. Boyapati, A. Salcianu, Jr. W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time java. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 324–337, New York, NY, USA, 2003. ACM Press.
7. Víctor A. Braberman, Federico Javier Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements. In *Symposium on Memory Management (ISMM)*, 2008.
8. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java programming language. October 1998.
9. Zhiqun Chen. *Java Card technology for Smart Cards: architecture and programmer's guide*. Addison-Wesley, 2000.
10. T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, and J. Vitek B. Titzer and. CDx: A Family of Real-time Java Benchmarks. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, 2009.
11. K. Nilsen. A type system to assure scope safety within safety-critical Java modules. In *Workshop on Java technologies for real-time and embedded systems (JTRES)*, 2006.
12. F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time Java scoped memory: Design patterns and semantics. In *Proceedings of the 7th IEEE International Symposium on, Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 101–110, 2004.

13. RTCA and EUROCAE. Software considerations in airborne systems and equipment certification. Radio Technical Commision for Aeronatics (RTCA), European Organization for Civil Aviation Electronics (EUROCAE), DO178-B, 1992.
14. T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 241–251, Washington, DC, USA, 2004. IEEE Computer Society.