Patterns for Safety-Critical Java Memory Usage

Juan Rios[1] Kelvin Nilsen[2] Martin Schoeberl[1]

[1] Department of Informatics and Mathematical Modelling
Technical University of Denmark

[2] Atego Systems, Inc.

Java Technologies for Real-time and Embedded Systems, 2012

Outline

1. Motivation
   - Problem Outline and Motivation
   - Previous Work
2. Contribution
   - Scoped Memory Usage Patterns
   - Proposed Modifications to SCJ API

Outline

1. Motivation
   - Problem Outline and Motivation
   - Previous Work
2. Contribution
   - Scoped Memory Usage Patterns
   - Proposed Modifications to SCJ API

Problem and Motivation

- Real-time Java profiles make use of scoped memory regions
- Explicit scoping requires care from programmers
  - Deal with temporary objects
  - Pass scope-allocated objects as arguments to methods
  - Returning scope-allocated objects from methods
- Look into the expressiveness of the SCJ memory model
- Explore patterns on how to use it
  - Focus on how to pass arguments and return objects between private memories

Problem and Motivation

- Real-time Java profiles make use of scoped memory regions
- Explicit scoping requires care from programmers
  - Deal with temporary objects
  - Pass scope-allocated objects as arguments to methods
  - Returning scope-allocated objects from methods
- Look into the expressiveness of the SCJ memory model
- Explore patterns on how to use it
  - Focus on how to pass arguments and return objects between private memories

Problem and Motivation

- Real-time Java profiles make use of scoped memory regions
- Explicit scoping requires care from programmers
  - Deal with temporary objects
  - Pass scope-allocated objects as arguments to methods
  - Returning scope-allocated objects from methods
- Look into the expressiveness of the SCJ memory model
- Explore patterns on how to use it
  - Focus on how to pass arguments and return objects between private memories

Problem and Motivation

- Real-time Java profiles make use of scoped memory regions
- Explicit scoping requires care from programmers
  - Deal with temporary objects
  - Pass scope-allocated objects as arguments to methods
  - Returning scope-allocated objects from methods
- Look into the expressiveness of the SCJ memory model
- Explore patterns on how to use it
  - Focus on how to pass arguments and return objects between private memories

Outline

1. Motivation
   - Problem Outline and Motivation
   - Previous Work
2. Contribution
   - Scoped Memory Usage Patterns
   - Proposed Modifications to SCJ API

Previous Work

- Memory pools and memory blocks in Immortal and scoped memories
- Collections of design patterns for RTSJ
  - RTSJ specific
  - Features not in SCJ
  - Violations to reference assignment rules
- Mapping between memory areas and methods
  - Reduce the need for reference assignment checks
  - Restricted use of RTSJ memory API very similar to SCJ private memories

Previous Work

- Memory pools and memory blocks in Immortal and scoped memories
- Collections of design patterns for RTSJ
  - RTSJ specific
  - Features not in SCJ
  - Violations to reference assignment rules
- Mapping between memory areas and methods
  - Reduce the need for reference assignment checks
  - Restricted use of RTSJ memory API very similar to SCJ private memories

Previous Work

- Memory pools and memory blocks in Immortal and scoped memories
- Collections of design patterns for RTSJ
  - RTSJ specific
  - Features not in SCJ
  - Violations to reference assignment rules
- Mapping between memory areas and methods
  - Reduce the need for reference assignment checks
  - Restricted use of RTSJ memory API very similar to SCJ private memories

Previous Work

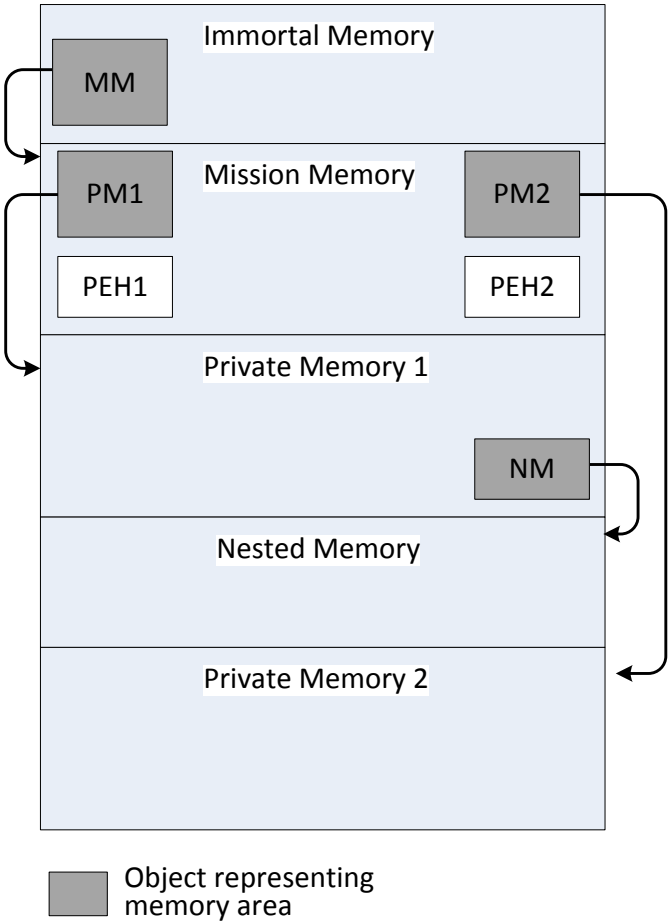
- IBM's *"Lifecycle Memory Managed Periodic Worker Threads"* pattern, with focus on object lifetimes and not on understanding scopes
  - Defines four categories of object lifetimes analogous to the lifetime of objects in SCJ scopes
    - *RETAIN_FOREVER* = Immortal Memory
    - *RETAIN_THREAD_GROUP* = Mission Memory
    - *RETAIN_THREAD* = Private Memory
    - *RETAIN_ITERATION* = Nested Private Memory
  - Relies on `java.lang.reflect` package, not part of SCJ

Outline

1. Motivation
   - Problem Outline and Motivation
   - Previous Work
2. Contribution
   - Scoped Memory Usage Patterns
   - Proposed Modifications to SCJ API

SCJ Memory API

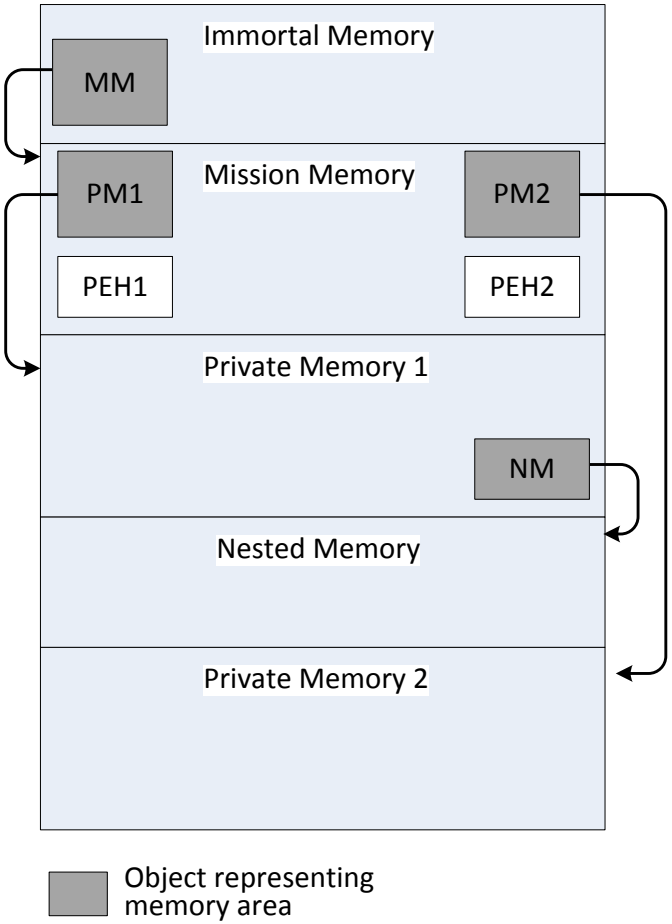
Brief description



- Memory area objects represent allocation contexts
- Memory areas are entered with:
  - `enterPrivateMemory()`
  - `executeInArea()`
- Creation of objects in an allocation context other than the current is done with:
  - `newInstance()`
  - `newArray()`
- It is possible to get references to memory areas with:
  - `getMemoryArea(Object O)`
  - `getCurrentManagedMemory()`

SCJ Memory API

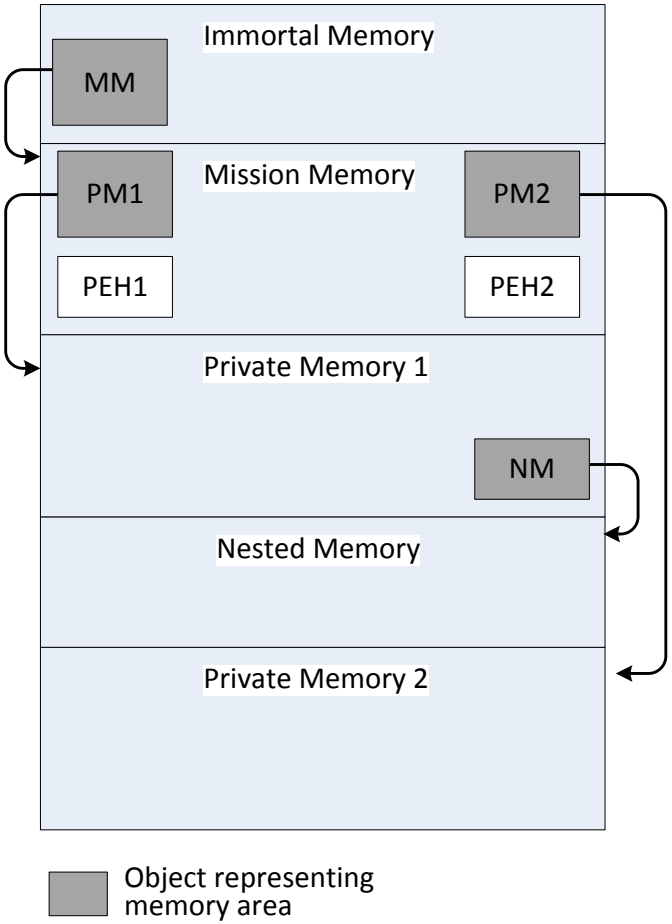
Brief description



- Memory area objects represent allocation contexts
- Memory areas are entered with:
  - `enterPrivateMemory()`
  - `executeInArea()`
- Creation of objects in an allocation context other than the current is done with:
  - `newInstance()`
  - `newArray()`
- It is possible to get references to memory areas with:
  - `getMemoryArea(Object O)`
  - `getCurrentManagedMemory()`

SCJ Memory API

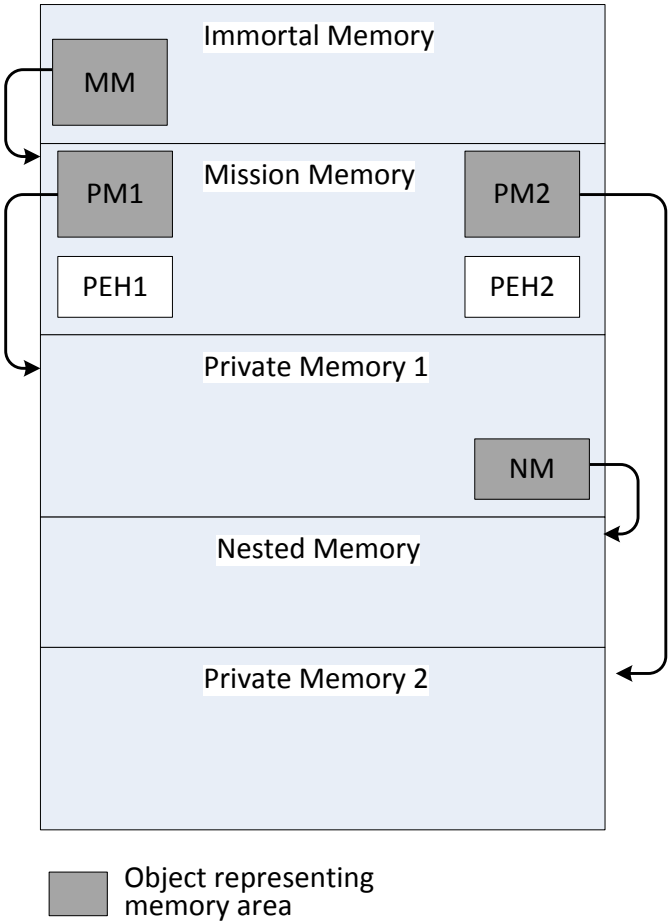
Brief description



- Memory area objects represent allocation contexts
- Memory areas are entered with:
  - `enterPrivateMemory()`
  - `executeInArea()`
- Creation of objects in an allocation context other than the current is done with:
  - `newInstance()`
  - `newArray()`
- It is possible to get references to memory areas with:
  - `getMemoryArea(Object O)`
  - `getCurrentManagedMemory()`

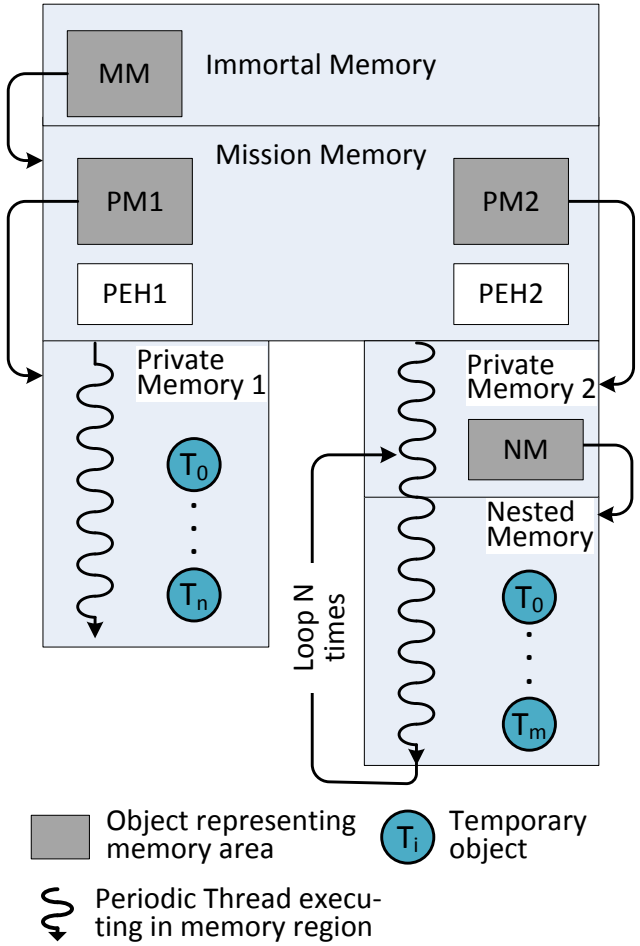
SCJ Memory API

Brief description



- Memory area objects represent allocation contexts
- Memory areas are entered with:
  - `enterPrivateMemory()`
  - `executeInArea()`
- Creation of objects in an allocation context other than the current is done with:
  - `newInstance()`
  - `newArray()`
- It is possible to get references to memory areas with:
  - `getMemoryArea(Object O)`
  - `getCurrentManagedMemory()`

Loop Pattern
Description



- Intent: Reuse of private memory space by several sub computations in a single handler release
- Condition: No need for returning results or to pass arguments
- Implementation: The `run()` method of Runnable instances are executed in private nested memory areas by calling `enterPrivateMemory()`

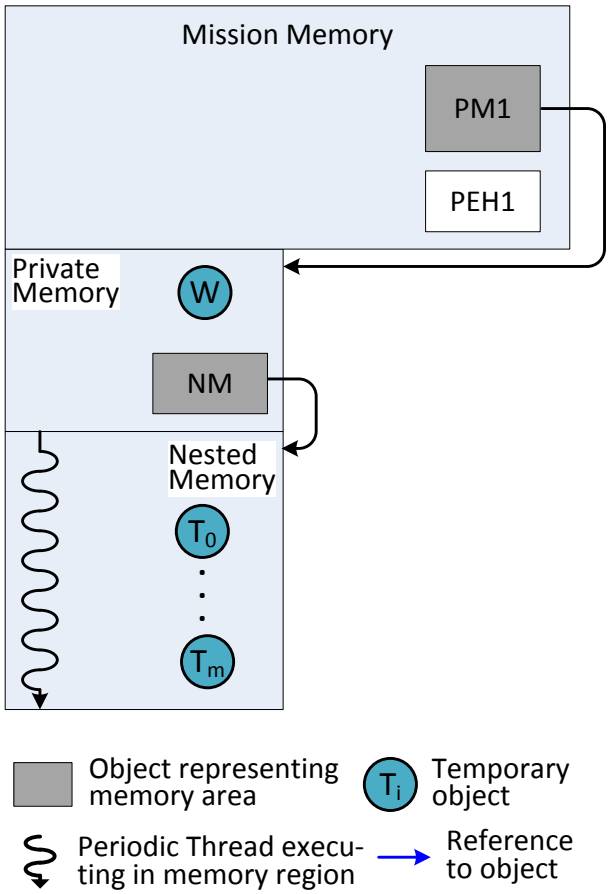
Loop Pattern
Example

```
class Worker implements Runnable {
  public void run() {
    // Your code here
  }
}

class MyHandler extends PeriodicEventHandler {

  public void handleAsyncEvent() {

    Worker w = new Worker();

    for(int i = 0; i < BLOCK_SIZE; i++){
      ManagedMemory.enterPrivateMemory(256, w);
    }
  }
}
```

- Intent: Reuse of private memory space by several sub computations in a single handler release
- Condition: No need for returning results or to pass arguments
- Implementation: The `run()` method of Runnable instances are executed in private nested memory areas by calling `enterPrivateMemory()`

Return a Newly Allocated Object

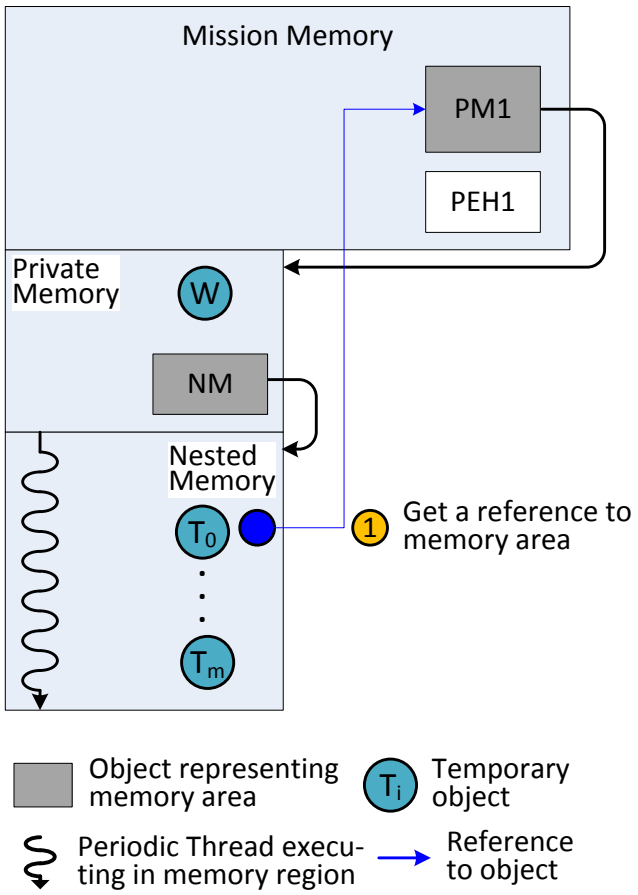
Description



- Intent: Pass parameters and preserve results within different (nested) private memory activations without using static fields
- The condition of returning only primitive values is relaxed
- Implementation: The returned object is created in a different memory area with the help of a memory reference

Return a Newly Allocated Object

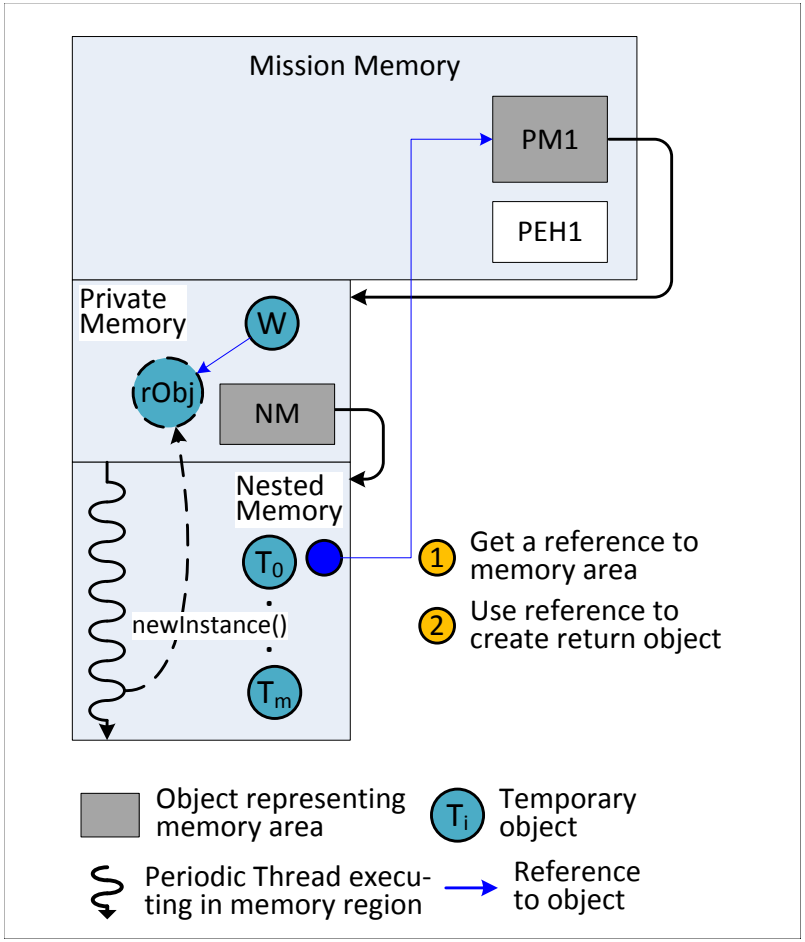
Description



- Intent: Pass parameters and preserve results within different (nested) private memory activations without using static fields
- The condition of returning only primitive values is relaxed
- Implementation: The returned object is created in a different memory area with the help of a memory reference

Return a Newly Allocated Object

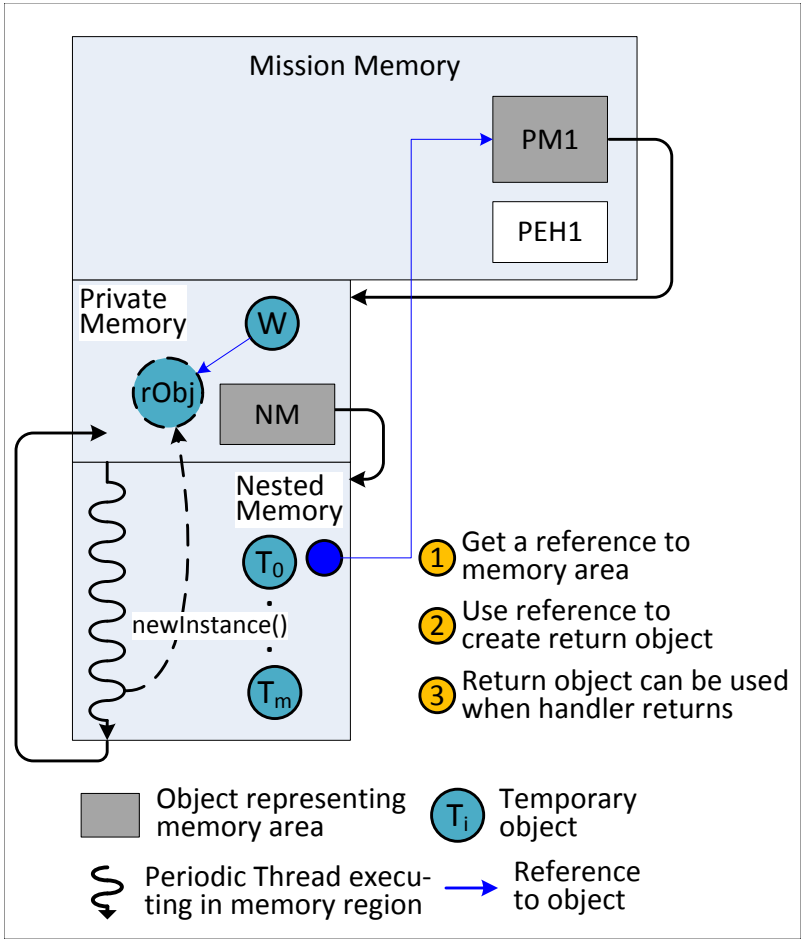
Description



- Intent: Pass parameters and preserve results within different (nested) private memory activations without using static fields
- The condition of returning only primitive values is relaxed
- Implementation: The returned object is created in a different memory area with the help of a memory reference

Return a Newly Allocated Object

Description



- Intent: Pass parameters and preserve results within different (nested) private memory activations without using static fields
- The condition of returning only primitive values is relaxed
- Implementation: The returned object is created in a different memory area with the help of a memory reference

Return a Newly Allocated Object

Example

```
class Worker implements Runnable {

  RetObject rObj;

  public void run()  {

    // Do some work...
    MemoryArea mem = MemoryArea.getMemoryArea(this);
    rObj = mem.newInstance(RetObject.class);

  }
};

class MyHandler extends PeriodicEventHandler {

  public void handleAsyncEvent(){

    Worker w = new Worker();
    ManagedMemory.enterPrivateMemory(256,w);

    // Use returned object and fields
    w.rObj ...
  }
}
```

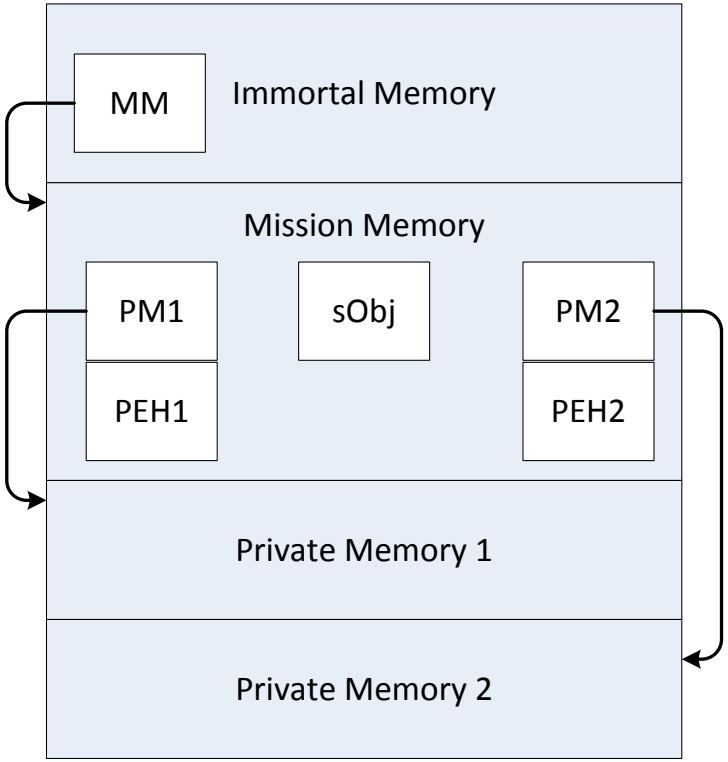
- Intent: Pass parameters and preserve results within different (nested) private memory activations without using static fields
- The condition of returning only primitive values is relaxed
- Implementation: The returned object is created in a different memory area with the help of a memory reference

Outline

1. Motivation
   - Problem Outline and Motivation
   - Previous Work
2. Contribution
   - Scoped Memory Usage Patterns
   - Proposed Modifications to SCJ API

Simplifying Allocation Context Change

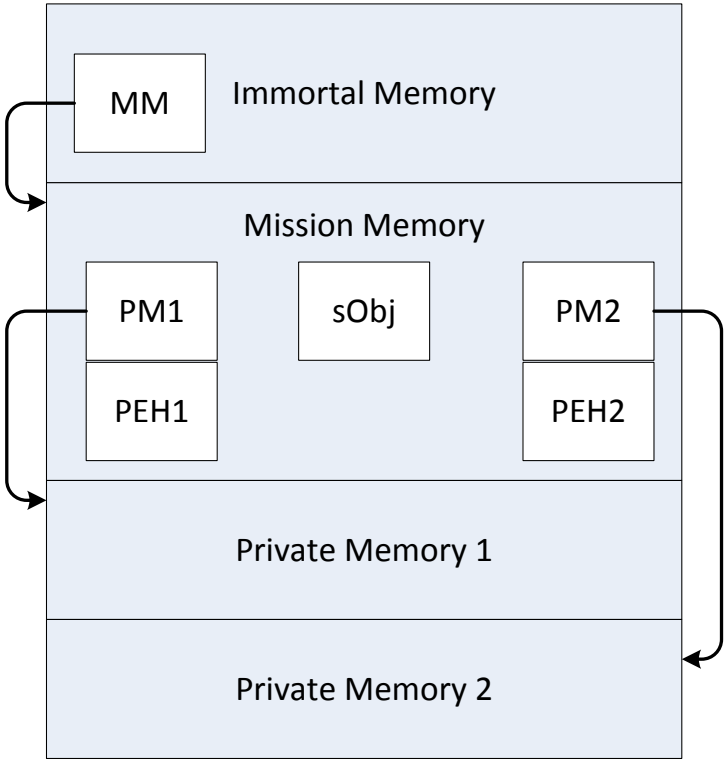
Leaking memory references between handlers



- A handler can store memory references in shared objects allocated in Mission Memory
  - By using `getCurrentManagedMemory()`
- Get a private memory reference from the handler object itself
  - By using `getMemoryArea(Object)`

Simplifying Allocation Context Change

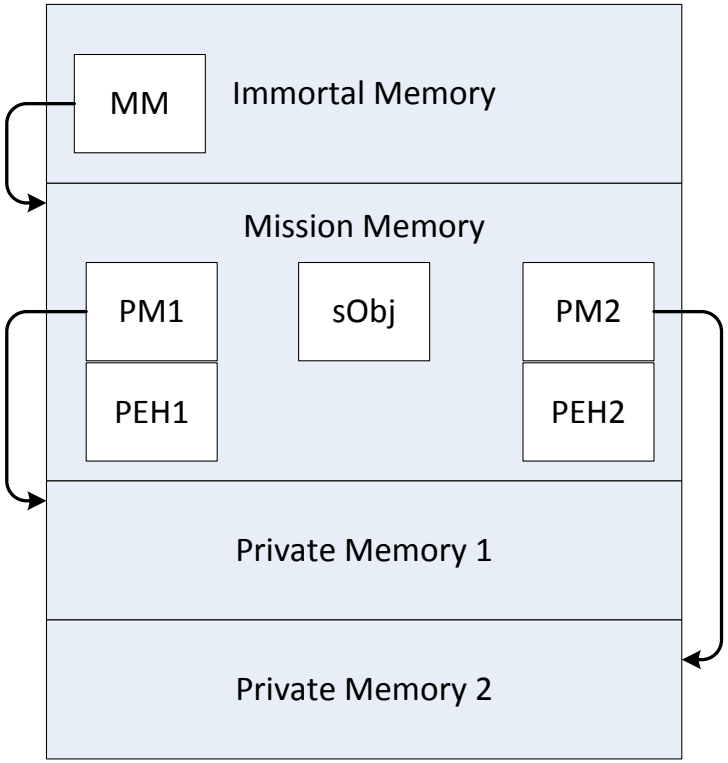
Runtime checks to prevent execution in different private scopes



- With an arbitrary memory reference, a handler can try to execute code in a different private memory
  - `executeInArea()`, `newInstance()` and `newArray()`
- RTSJ requires checking at runtime that the target memory is in the current thread scope stack

Simplifying Allocation Context Change

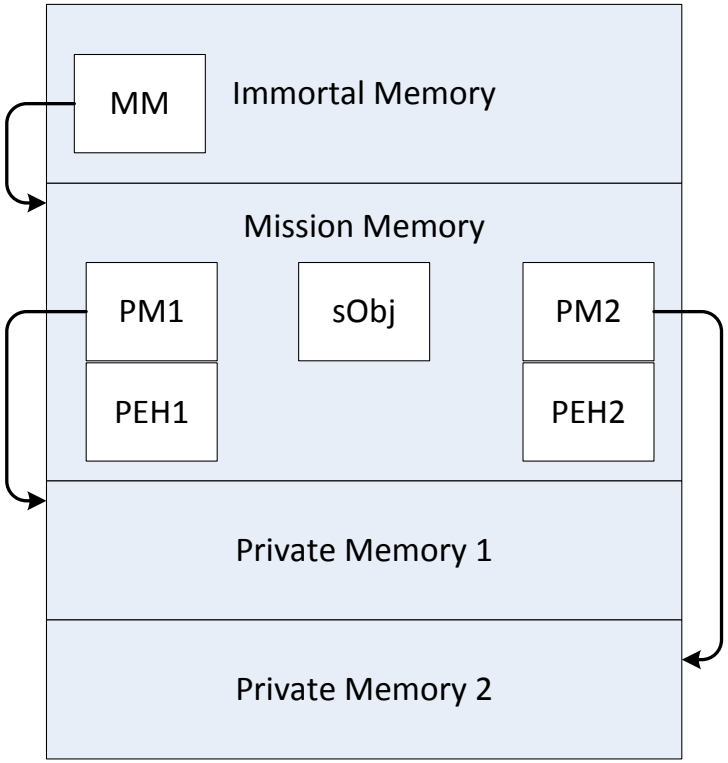
Proposed changes in the current API



- Hide the following methods from the public API:
  - `getCurrentManagedMemory()`
  - `getMemoryArea()`
  - `executeInArea()`
  - `newInstance()`
  - `newArray()`
- Replace `executeInArea()` with a new static method:
  - `executeInOuter(object, runnable)`
  - `executeInOuter(runnable)`
- New API will be semantically equivalent to the old
  - No memory reference leaks
  - No runtime checks

Simplifying Allocation Context Change

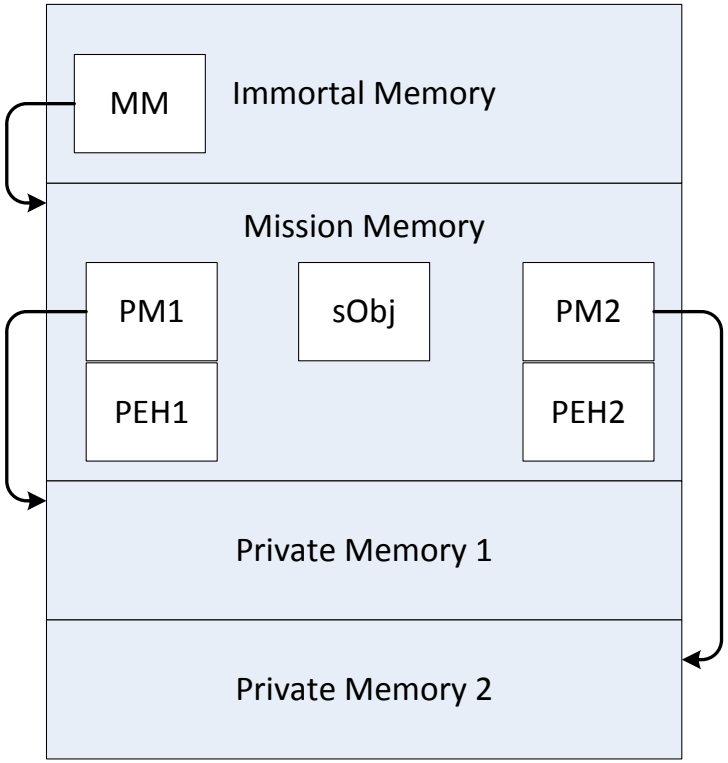
Proposed changes in the current API



- Hide the following methods from the public API:
  - `getCurrentManagedMemory()`
  - `getMemoryArea()`
  - `executeInArea()`
  - `newInstance()`
  - `newArray()`
- Replace `executeInArea()` with a new static method:
  - `executeInOuter(object, runnable)`
  - `executeInOuter(runnable)`
- New API will be semantically equivalent to the old
  - No memory reference leaks
  - No runtime checks

Simplifying Allocation Context Change

Proposed changes in the current API



- Hide the following methods from the public API:
  - `getCurrentManagedMemory()`
  - `getMemoryArea()`
  - `executeInArea()`
  - `newInstance()`
  - `newArray()`
- Replace `executeInArea()` with a new static method:
  - `executeInOuter(object, runnable)`
  - `executeInOuter(runnable)`
- New API will be semantically equivalent to the old
  - No memory reference leaks
  - No runtime checks

Summary

- Scoped memory use patterns are useful to develop SCJ applications
  - Having arguments and returning results from inner scopes is not so obvious
- An explicit change of allocation context into an outer memory eliminates:
  - Run time checks (target memory need to be in thread's scope stack)
  - The possibility to leak memory references
- Future work
  - Is it possible to obtain re-usable code by rewriting portions of Java's standard libraries to be scope-safe?

Leaking Memory References

An example

```
public class MyMission extends Mission {

  public void  initialize  ()  {
    ...
    PeriodicEventHandler PEH1 = new PeriodicEventHandler(...);
    PeriodicEventHandler PEH2 = new PeriodicEventHandler(..., PEH1);
    ...
  }
}

public class PEH2 extends PeriodicEventHandler {

  PeriodicEventHandler PEH1;

  public PEH2(..., PeriodicEventHandler peh) {
    super (...);
    PEH1 = peh;
  }

  public void handleAsyncEvent() {
    ...
    MemoryArea mem = MemoryArea.getMemoryArea(PEH1);
    mem.executeInArea(...);
    ...
  }
}
```

The Scope Stack in SCJ