Roadmap

- Overview of the RTSJ
- **Memory Management**
- Clocks and Time
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- Resource Control

Memory Management

- Lecture aims
  - To provide an introduction to memory management in the RTSJ
  - To give an overview of MemoryAreas
  - To show an example of Scoped Memory Usage
  - To consider how to estimating the size of objects

Introduction I

- An advantage of using a high-level language is that it relieves the programmer of the burden of dealing with many of the low-level resource allocation issues
- Issues such as assigning variables to registers or memory locations, allocating and freeing memory for dynamic data structures, etc., all detract the programmer from the task at hand
- Java remove many of these troublesome worries and provide high-level abstract models that the programmer can use

Introduction II

- For real-time/embedded systems programming there is a conflict
  - Use of high-level abstractions aid in software engineering
  - Yet embedded and real-time systems often only have limited resources which must be carefully managed
- Nowhere is this conflict more apparent than in the area of memory management
- Embedded systems usually have a limited amount of memory available; this is because of cost, size, power, weight or other constraints imposed by the overall system requirements
- It may be necessary to control how this memory is allocated so that it can be used effectively

Stack versus Heap Memory

- Most languages provide two data structures to help manage dynamic memory: the stack and the heap
- The stack is typically used for storing variables of basic data types (such as int, boolean and references) local to a method
- All objects, which are created from class definitions, are stored on the heap and Java requires GC
- Much work has been done on real-time GC, yet there is still a reluctance to rely on it in time-critical systems

Real-time and Garbage Collection

- GC may be performed either when the heap is full or incrementally (asynchronous garbage collection)
- GC may have significant impact on the response time of a RT thread
- Consider a time-critical periodic thread which has had all its objects pre-allocated
- Even though it may have a higher priority than a plain thread & will not require any new memory, it may still be delayed if it preempts the plain thread during GC
- It is not safe for the time-critical thread to execute until GC has finished (particularly if memory is compacted)

The RTSJ and Memory Management

- The RTSJ recognizes that it is necessary to allow memory management which is not affected by the problems of garbage collection
- It does this via the introduction of immortal and scoped memory areas
- These are areas of memory which are logically outside of the heap and, therefore, are not subject to garbage collection
- If a schedulable object is active in a memory area, all calls to new create the object in that memory area

The Basic Model

- RTSJ provides two alternatives to using the heap: immortal memory and scoped memory
- The memory associated with objects allocated in immortal memory is never subject to GC
- Objects allocated in scoped memory have a well-defined life time
- Schedulable objects may enter and leave a scoped memory area
- Whilst they are executing within that area, all memory allocations are performed from the scoped memory
- When there are no schedulable objects active inside a scoped memory area, the allocated memory is reclaimed

Memory Areas I

```
public abstract class MemoryArea {
  protected MemoryArea(long sizeInBytes);
  protected MemoryArea(long sizeInBytes, Runnable logic);
  public void enter();
```

Associate this memory area to the current schedulable object for the duration of the run method passed as a parameter to the constructor

```
  public void enter(Runnable logic);
```

Associate this memory area to the current schedulable object for the duration of the run method of the object passed as a parameter

```
  public static MemoryArea getMemoryArea(Object object);
```

Get the memory area associated with the object

```
  public long memoryConsumed();
```

Returns the number of bytes consumed

```
  public long memoryRemaining();
```

Returns the number of bytes remaining

```
  public long size();
```

Returns the current size of the memory area

Immortal Memory

- There is only one `ImmortalMemory` area, hence the class is defined as final and has only one additional method (`instance`) which will return a reference to the immortal memory area
- Immortal memory is shared among all threads in an application
- Note, there is no public constructor for this class. Hence, the size of the immortal memory is fixed by the JVM

```
public final class ImmortalMemory extends MemoryArea {
  public static ImmortalMemory instance();
}
```

Allocating Objects in Immortal Memory

- The simplest method for allocating objects in immortal memory is to use the enter method in the `MemoryArea` class and pass an object implementing the `Runnable` interface

```
ImmortalMemory.instance().enter(new Runnable(){
      public void run() {
        any memory allocation performed here using the allocator will occur in Immortal
      }
    });
```

Although memory allocated by the **`run`** method will occur from immortal memory, memory needed by the object implementing the **`run`** method will be allocated from the current memory area at the time of the call to enter

Linear Time Memory

```
public class LTMemory extends ScopedMemory {
  public LTMemory(long initialSizeInBytes,
                  long maxSizeInBytes);
  public LTMemory(long initialSizeInBytes,
                  long maxSizeInBytes, Runnable logic);
  ...
  public int getMaximumSize();
```

Linear time refers to the time it takes to allocate an object, and not the time it takes to run the constructor

Reference Counts

- Each scoped memory object has a reference count which indicates the number of times the scope has been entered
- When that reference count goes from 1 to 0, the memory allocated in the scoped memory area can be reclaimed (after running any finalization code associated with the allocated objects)

Example: Scoped Memory

- Consider a class that encapsulates a large table which contains (two by two) matrices
- The match method takes a two by two matrix and performs the dot product of this with every entry in the table
- It then counts the number of results for which the dot product is the unity matrix

Matrix

```
class MatrixExample {
  MatrixExample(int Size) { /* initialize table */ }

  int match(final int with[][])
  {/*check if "with" is in the table*/}

  private int[][] dotProduct(int[][] a, int [][] b)
  { /* calculate dot product return result */ }

  private boolean eq(int[] [] a, int[] [] b)
  { /* returns true if the matrices are equal */ }

  private int [][][] table; // 2 by 2 matrices
  private int [][] unity = {{1,1},{1,1}};
}
```


Match Method: with GC

```
public int match(final int with[][]) {
  int found = 0;
  for(int i=0; i < table.length; i++) {
    int[][] product = dotProduct(table[i], with);
    if(eq(product, unity)) found++;
  }
  return found;
}
```

- Each time around the loop, a call to `dotProduct` is made which creates and returns a new matrix object
- This object is then compared to the unity matrix
- After this comparison, the matrix object is available for GC

Match Method: LTMemory

```
public int match(final int with[][]) {// first attempt
  LTMemory myMem = new LTMemory(1000, 5000);
  int found = 0;
  for(int i=0; i < table.length; i++) {
    myMem.enter(new Runnable(){
      public void run() {
        int[][] product = dotProduct(table[i], with);
        if(eq(product, unity)) found++;
       }
      });
    }
  }
  return found;
}
```


Problems

- Accessibility of the loop parameter `i`; only **`final`** local vars can be accessed from within a class nested within a method
- To solve this: create a new **`final`** variable, `j`, inside the loop which contains the current value of `i`
- **`found`** must become a field and initialized to 0 on each call
- The anonymous class is a subclass of `Object` not the outer class, consequently, **`eq`** is not in scope
- This can be circumvented by naming the method explicitly

Matrix Version 2

```
public class MatrixExample    {
  public MatrixExample(int Size)
  { /* initialize table */ }

  public int match(final int with[][])
  {  /* check if "with" is in the table */ }

  private int[][] dotProduct(int[][] a, int [][] b)
  { /* calculate dot product return result */ }

  private boolean equals(int[] [] a, int[] [] b)
  { /* returns true if the matrices are equal */ }

  private int [][][] table; // 2 by 2 matrices
  private int [][] unity = {{1,1},{1,1}};
  public int found = 0;
  private LTMemory myMem;
}
```


Match Version 2

```
public int match(final int with[][]) {
  found = 0;
  for(int i=0; i < table.length; i++) {
    final int j = i;
    myMem.enter(new Runnable(){
      public void run() {
        int[][] product = dotProduct(table[j], with);
        if(MatrixExample.this.equals(product, unity))
          found++;
      }
    });
  }
  return found;
}
```

Problem

- We now reclaim temporary memory every time **`enter()`** returns
- However creating instance of the anonymous class still use memory in the surrounding memory area
- There will be a few bytes of objects header (e.g. a monitor, class table, hash code) and a copy of the `j` variable and a reference to the **`with`** array (this is how the compiler implements the access to the method's data)
- Although this memory usage cannot be eliminated, it can be bounded by caching the object in a field

Matrix Final Version

```
public class MatrixExample {
  ...
  Product produce = new Product();

  private class Product implements Runnable {
    int j;
    int withMatrix[][];
    int found = 0;
    public void run() {
      int[][] product=dotProduct(table[j],withMatrix);
      if(matrixEquals(product, unity)) found++;
    }
  }
  ...
}
```

Match Final Version

```
public int match(final int with[][])  {
  produce.found = 0;
  for(int i=0; i < table.length; i++) {
    produce.j = i;
    produce.withMatrix = with;
    myMem.enter(produce);
  }
  return produce.found;
}
```

- Now, there is just the initial cost of creating the produce object (performed in the constructor of `MatrixExample`)
- Note that the only way to pass parameters to the run method is via setting attributes of the object (in this case directly)
- Note also that only one thread may call match at a time

Size Estimator

```
public final class SizeEstimator {
  public SizeEstimator();
  public long getEstimate();
  public void reserve(Class c, int n);
  public void reserve(SizeEstimator s);
  public void reserve(SizeEstimator s, int n);
```

- Allows the size of a Java object to be estimated
- Again, size is the object itself, it does not include an objects created during the constructor

```
SizeEstimator s = new SizeEstimator();
s.reserve(javax.realtime.PriorityParameters.class, 1);
System.out.println("size of PP is "+s.getEstimate());
```

Summary

- The lack of confidence in RTGC is one of the main inhibitors to the widespread use of Java in real-time and embedded systems
- The RTSJ has introduced an alternative memory management facility based on the concept of memory areas
- There are two types on non-heap memory areas
  - a singleton immortal memory which is never subject to GC
  - scoped memory in to which schedulable objects can enter and leave. When there are no schedulable objects active in a scoped memory area, all the objects are destroyed and their memory reclaimed

Roadmap

- Overview of the RTSJ
- **Memory Management**
- Clocks and Time
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- Resource Control

Memory Management

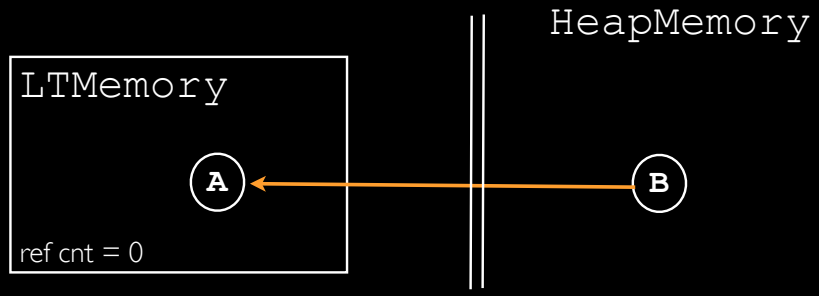
Lecture aims:

- To explain the RTSJ assignment Rules
- To illustrate the single parent rule
- To consider the sharing of memory areas between `Schedulable` objects

Memory Assignment Rules

- In the RTSJ there are four types of memory
  - heap memory: collected by the garbage collector
  - local variables: collected automatically when methods exit
  - immortal memory: never collected
  - scoped memory: collected when reference count equals zero
- Given the different collection mechanism, it is necessary to be careful when accessing memory
- Otherwise dangling references may occur
- *A dangling reference is a references to an already collected object*

Dangling Reference Example



- **A**, has been created in a scoped memory region
- A reference to **A** has been stored in object, **B**, in the heap
- The lifetime of a scoped memory is controlled by its reference count, if it goes to 0, the there will be a reference to a non-existent object

The Reference Count

The reference count of a scoped memory area is the count of the number of active calls to its **`enter`** method. It is **not** a count of the number of objects that have references to the objects allocated in the scoped memory.

The reference to object, **A**, from **B** becomes invalid when **A**'s memory area is reclaimed. Without something like GC there is no support for detecting this invalid reference, so the safety of the Java program would be compromised.

Memory Assignment Rules

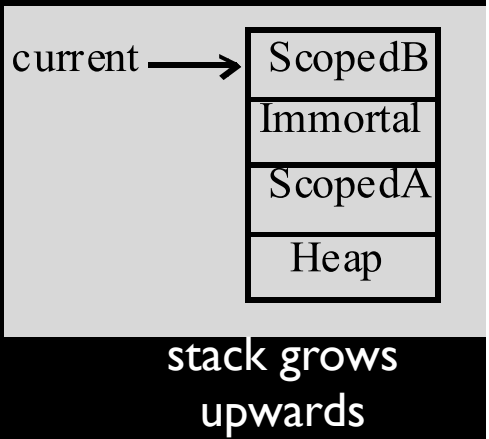
From	To Heap Memory	To Immortal Memory	To Scoped Memory
Heap Memory	allowed	allowed	forbidden
Immortal Memory	allowed	allowed	forbidden
Scoped Memory	allowed	allowed	allowed is to same scope or outer scope forbidden if to an inner scope
Local Variable	allowed	allowed	generally allowed

Note

- If the program violates the assignment rules, the unchecked exception `IllegalAssignmentError` is thrown
- One of the requirements for RTSJ was that there should be no changes to the Java language and that existing compilers can be used to compile RTSJ programs
- So these rules are enforced on every assignment at run-time by the JVM
- An RTSJ-aware compiler may optimize some check at compile-time or at class loading-time, but there is likely to be some residual checks
- In practice, the overhead of the checks is 10 - 20% of execution time if implemented efficiently

Nested Memory Areas

- The JVM keeps track of the currently active memory areas of each schedulable object
- Usually by keeping a stack of areas
- Every time a schedulable object enters an area, the identity of that area is pushed onto the stack
- When it leaves the memory area, the identity is popped off

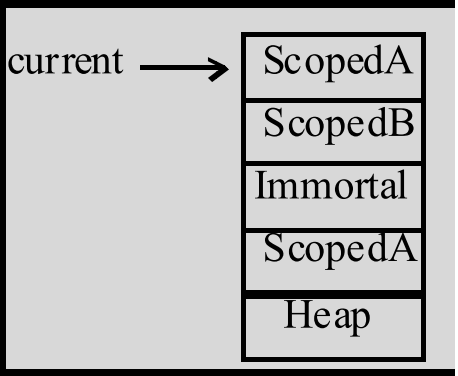


Implementation of Assignment Rules

- The stack can be used to check for invalid memory assignment to and from scoped memory areas
- Creating a reference from an object in a scoped area to an object in another scoped are below the first area in the stack is allowed
- Creating a reference from an object in one scoped memory area to an object in another area above the first area is forbidden

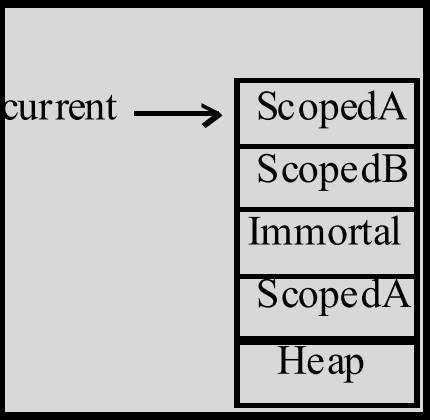
Consider

- Assume a schedulable enters the same memory area twice
- an object could be created in **`ScopedA`** which references an object in **`ScopedB`**
- when the current **`ScopedA`** memory is exited, its reference count is still **`>0`**. so its objects are not reclaimed
- when **`ScopedB`** is exited, it objects are reclaimed creating a dangling reference in **`ScopedA`**



The Single Parent Rule

- To avoid this problem, the RTSJ requires that each scoped memory area has a single parent
- The parent of an active scoped memory area is
  - If the memory is the first scoped area on the stack, its parent is termed the primordial scope area
  - For all other scoped memory areas, the parent is the first scoped area below it on the stack



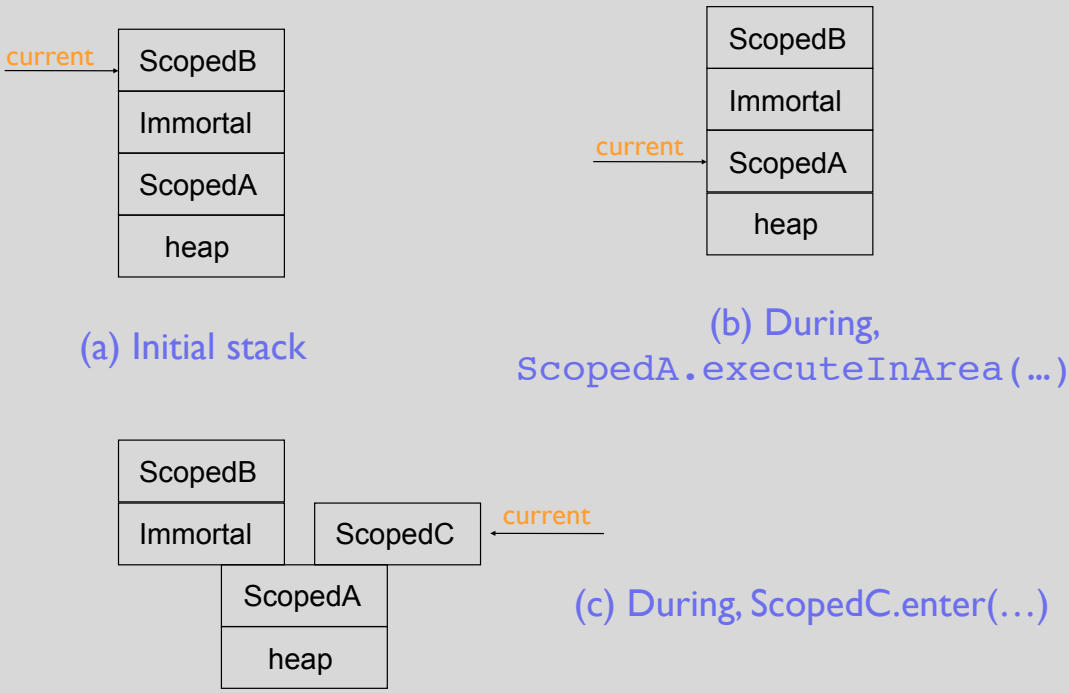
The example violates the single parent rule, therefore `ScopedCycleException` is thrown

Moving Between Memory Areas

- As it is not possible to re-enter an active scoped memory area, it is necessary to provide alternative mechanisms for moving between active memory areas

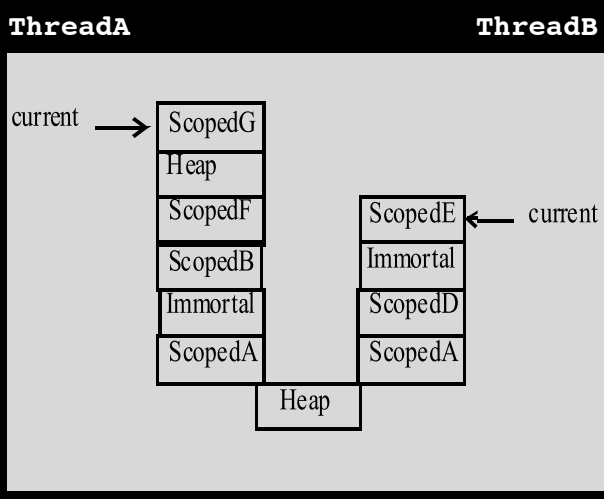
```
public abstract class MemoryArea {

  public void executeInArea(Runnable logic);

  public Object newArray(Class type, int number)throws …
  public Object newInstance(Class type) throws …
  public Object newInstance(Constructor c, Object[] args)
    throws …
```


Cactus Stack Needed



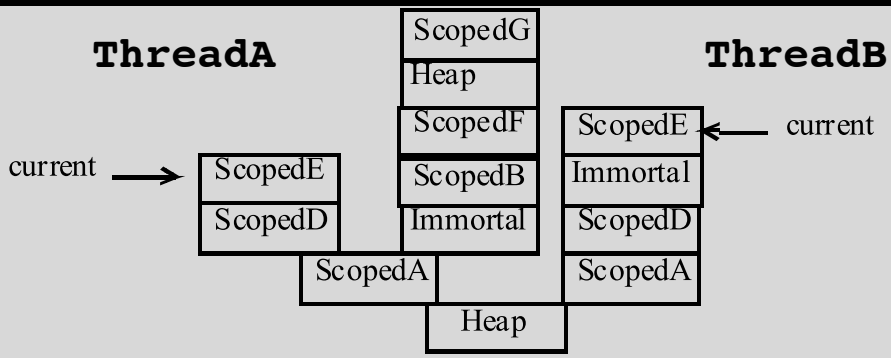
Sharing Memory Areas

- Multiple schedulable objects can access the same memory areas
- Consequently, the cactus stacks for each schedulable objects are linked together
- Here, two real-time threads (**`ThreadA`** and **`ThreadB`**) have active scoped memory stacks



Sharing Memory Areas

- Suppose **`ThreadA`** wishes to enter **`ScopedE`**; it cannot do so directly because **`ScopedE`** is active and has parent **`ScopedD`**
- To access **`ScopedE`**, **`ThreadA`** must move to **`ScopedA`** (using **`executeInArea`**), then enter **`ScopedD`** and **`ScopedE`**



Entering and Joining Scoped Memory Areas

- Memory areas can be used cooperatively or competitively
- In cooperative use, the schedulable objects aim to be active in a scoped memory simultaneously
  - they use the area to communicate shared objects
  - when they leave, the memory is reclaimed
- In competitive use, they aim for efficient use of memory
  - the schedulable objects are trying to take their memory from the same area but are not using the area for the communication
  - it is usually required for only one schedulable to be active in the area at a time
  - the intention is that the memory can be reclaimed when each of the schedulable objects leave the area

Competitive Use

- To ensure that the area becomes inactive use join.
- If timeout expires, the memory area is entered
- It is difficult to know if the timeout did expire

```
public abstract class ScopedMemory extends MemoryArea {
  public int getReferenceCount();
  public void join() throws InterruptedException;
  public void join(HighResolutionTime time) throws …
  public void joinAndEnter() throws InterruptedException;
  public void joinAndEnter(HighResolutionTime time) throws…
  public void joinAndEnter(java.lang.Runnable logic) throws…
  public void joinAndEnter(java.lang.Runnable logic,
                           HighResolutionTime time) throws…
```

Summary

- The lack of confidence in RTGC is one of the main inhibitors to the widespread use of Java in real-time and embedded systems
- The RTSJ has introduced an additional memory management facility based on the concept of memory areas
- There are two types on non-heap memory areas
  - immortal memory which is never subject to garbage collection
  - scoped memory in to which schedulable objects can enter and leave; when there are no schedulable objects active in a scoped memory area, all the objects are destroyed and their memory reclaimed.
- Due to the variety of memory areas, the RTSJ has strict assignment rules between them in order to ensure that dangling references do not occur