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
- Schedulability Analysis
- Conclusions

Memory Management

- Lecture aims:
- To motivate the need for portal objects and provide an example of their use
- To consider the real-time issues of scoped memory areas

Recall from previous lecture

- Scoped memory areas can be used two modes of operation: cooperative or competitive
- In cooperative use, the SOs are active in a scoped memory area simultaneously
  - they use the area to communicate shared objects
  - when they leave, the memory is reclaimed
- In competitive use, the goal is to make the most efficient use of memory
  - only one schedulable is active in the memory area at one time
  - the intention is that the memory can be reclaimed when each of the schedulable objects leave the area

Portals

- For cooperative sharing where there is no relationship between the SOs, how can SOs share objects created in a memory area?
- To share an object requires each SO to have a reference to it
  - A reference to an object can only be stored in an object in the same scoped area or in an object in a nested scoped area
  - It cannot be stored in the immortal or heaped memory area
- Consequently, unless there is some relationship between the SOs, one SO cannot pass a reference to an object it has just created to another SO

Portals

- Portals solve this problem; each memory area can have one object which can act as a gateway into that memory area
- SOs can use this mechanism to facilitate communication

```
public abstract class ScopedMemory extends MemoryArea {
   ...
   public Object getPortal();
   public void setPortal(Object o);
}
```

Portal Example

- Consider, an object which controls the firing of a missile
- For the missile to be launched two independent RT threads must call its fire method each with its own authorization code

```
public class FireMissile {
    public FireMissile();
    public boolean fire1(final String authorizationCode);
    public boolean fire2(final String authorizationCode);
}
```

Portal Example

- The two threads call **`fire1`** and **`fire2`** respectively
- Whichever calls in first has it authorization code checked and is held until the other thread calls its fire method
- If both threads have valid authorization codes, the missile is fired and the methods returns true, otherwise the missile is not fired and false is returned.
- In order to implement the fire methods, assume that objects need to be created in order to check the authorization

```
class Decrypt {
  boolean confirm(String code){/*check authorization*/}
}
```

Portal Example

- To synchronize, the two RT threads must communicate

```
class BarrierWithParameter {
  BarrierWithParameter(int participants);
  synchronized boolean wait(boolean go);
}
```

The threads call wait indicating whether they wish to fire. If both pass true, wait returns true

Portal Example

- The goal is to implement **`FireMissile`** without using extra memory other than that required to instantiate the class
- All memory needed by the fire methods should be created in scoped memory which can be reclaimed when the methods are inactive
- In order to implement the required firing algorithm, the class needs two **`Decrypt`** objects and one **`BarrierWithParameter`** object

Portal Example

```
public class FireMissile {
  public FireMissile() {
    ...
    SizeEstimator s = new SizeEstimator();
    s.reserve(Decrypt.class, 2);
    s.reserve(BarrierWithParameter.class, 1);
    shared = new LTMemory(s.getEstimate(),
                          s.getEstimate());
  }
  private LTMemory shared;
  ...
}
```

Portal Example

- Both threads need to access a **`BarrierWithParameter`** object, they therefore must enter into the same memory area
- The shared memory area must have a single parent
- As this class is unaware of the scoped stacks of the calling threads, it needs to create new scoped memory stacks
- It does this by using the **`executeInArea`** method to enter into immortal memory
- The **`run`** method can now enter into the scoped memory area.

The alternative approach would be to assume the calling threads have empty (or the same) scope stacks

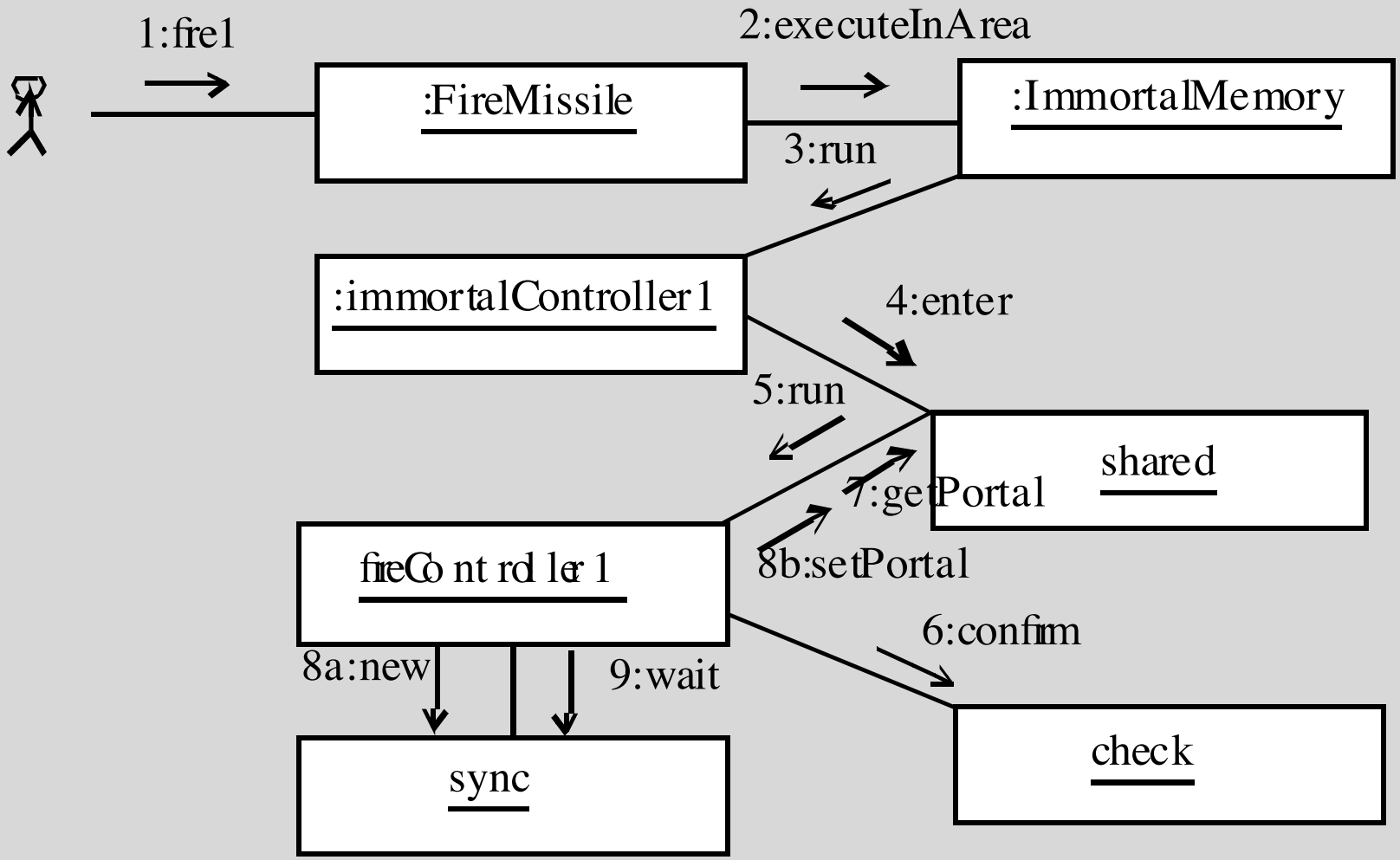
Portal Example

```
class FireMissile {
  private LTMemory shared;
  private FireAction fireController1, fireController2;
  private ImmortalAction immController1,
                         immController2;
  FireMissile() {
    fireController1 = new FireAction();
    fireController2 = new FireAction();
    immController1 = new ImmortalAction();
    immController2 = new ImmortalAction();
    SizeEstimator s = new SizeEstimator();
    s.reserve(Decrypt.class,2);
    s.reserve(BarrierWithParameter.class, 1);
    shared = new LTMemory(s.getEstimate(),
                          s.getEstimate());
  }
```

Portal Example

```
class FireMissile {
  class FireAction implements Runnable {
    String authorization;  boolean result;
    public void run() { /*coordinate missile firing */ }
  }
  class ImmortalAction implements Runnable {
    FireAction fireController;
    public void run(){ shared.enter(fireController);  }
  }
  boolean fire1(final String authorizationCode) {
    try {
      immortalController1.fireController = fireController1;
      fireController1.authorization = authorizationCode;
      ImmortalMemory.instance().executeInArea(immController1);
      return fireController1.result;
    }
    catch (InaccessibleMemoryArea ma)  {}
  }   // similarly for fire2
```

Portal Example



Portal Example

- Once in the memory area, the required objects can be created
- However, there is a problem with the shared `BarrierWithParameter` object.
  - In order for both threads to access it they must have a reference to it
  - Usually, the reference would be stored in a field declared at the class level
  - However, this is not possible, as it would break the RTSJ assignment rules as the object is in an inner scope

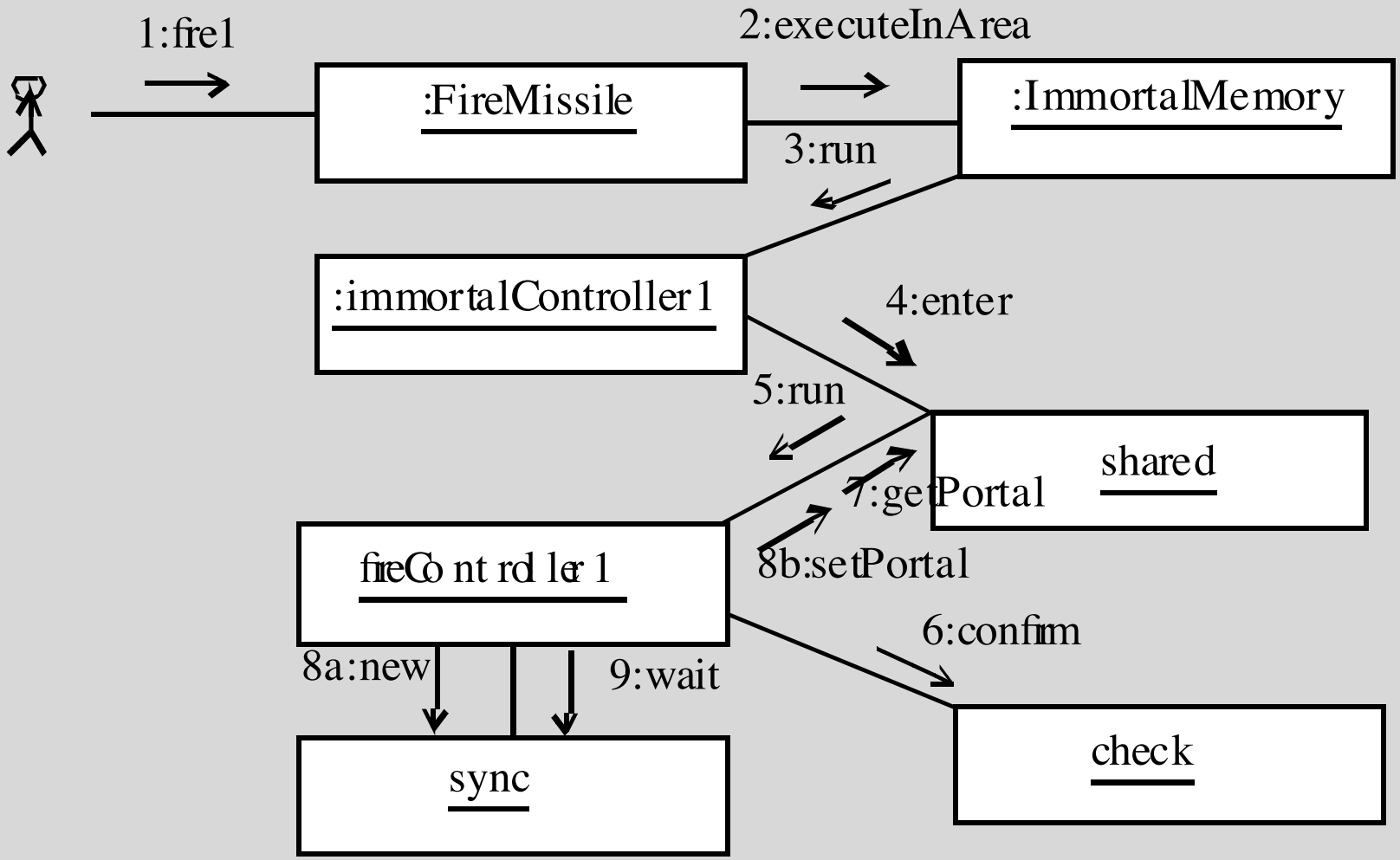
Portal Example

- The first thread to arrive attempts to obtain the portal object for the shared memory region
  - This is null, so it creates the **`BarrierWithParameter`** object and sets it up as the portal object
- The next thread to arrive can then obtain the required reference
- When both threads leave the shared memory area, all the memory is reclaimed
- However, there is a race condition between checking for the existence of the portal object and setting a newly created one
- Consequently, a lock is needed. One lock that is available at this stage is the lock associated with the memory object itself!

Portal Example

```
class FireAction implements Runnable {
  String authorization;  boolean result;
  public void run() {
    BarrierWithParameter sync;
    Decrypt check = new Decrypt();
    boolean confirmed = check.confirm(authorization);
    synchronized(RealtimeThread.getCurrentMemoryArea()) {
      sync = (BarrierWithParameter)shared.getPortal();
      if(sync == null)
        shared.setPortal(sync = new BarriewWithParameter(2));
    }
    result = sync.wait(confirmed);
  }
```

Portal Example



Using Scoped Memory Areas

- When a class `C` needs temporary memory should it take responsibility for creating and entering a scoped area?
- If `C` is intended for sequential access and is used concurrently, a `ScopedCycleException` may be thrown
  - The client SOs must either ensure that the scoped memory stack is empty or, ensure that all clients call with same stack. Clients need to have details of when a class is using scoped areas
- If `C` targets concurrent access:
  - Clients must either have the same scoped memory stacks, or `C` must "execute in" a heap or an immortal area first
  - if `C` is in the heap, it can't be used by a no-heap client; if `C` chooses immortal memory, named inner classes must be used, otherwise every time it instantiate an anonymous class, memory will be created in immortal memory and this will not be reclaimed

Real-Time Issues

- Entry to scoped memory
  - On entry into a scoped memory region, a SO may be blocked if it uses one of the **`join`** or **`joinAndEnter`** methods
  - The duration of this blocking may be difficult to bound (unless timeouts are used, but these have their own problems)
  - If the SOs using the scoped memory do not themselves block when they have entered the area, the blocking time will be the maximum time that lower priority SOs take to execute the associated **`run`** methods
  - When a SO attempts to enter into a previously active scoped memory area, it may also have to wait for the memory to be reclaimed (finalizers run, etc.)

Real-Time Issues

- Predictable scoped memory allocation
  - Memory allocation in a scoped memory should be predictable from the details of the implementation and the objects being created
  - It will consist of two components: the time taken to allocate the space for the objects (which will be proportional to their sizes, for LTMemory), and the time taken to execute the constructors for the objects created

Real-Time Issues

- Exit from scoped memory
  - The RTSJ gives a great deal of freedom on when the memory used in a scoped memory area is reclaimed
  - An implementation might decide to reclaim the memory immediately the reference count becomes zero or sometime after it becomes zero, but no later than when a new SO wishes to enter it
  - Hence, a SO on one occasion may leave the scoped memory and suffer no impact; on another occasion, it is the last object to leave and reclamation occurs immediately; alternatively it may suffer when it first enters the scope

Real-Time Issues

- Object finalization
  - Whenever scoped memory is reclaimed, the objects that have been created must have their finalization code executed before reclamation
  - The time for this to occur must be accounted for in any analysis.
- Garbage collector scans
  - The impact, if any, of the GC scanning the scoped memory area looking for heap references when those memory areas are being reclaimed concurrently

Summary

- Portal objects can be used to facilitate communication between schedulable objects using the same scoped memory area
- There is little doubt that non-heap memory management is one of the most complicated areas of the RTSJ, and one that has a major impact on the overheads of the virtual machine

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

Clocks and Time

Lecture aims

- To provide some background on clocks and time
- To explain absolute and relative time values
- To consider the support for real-time clocks
- To give some simple examples

Introduction I

- Java only supports the notion of a wall clock (calendar time); for many applications, a clock based on UTC is sufficient
- However real-time systems often require
  - A monotonic clock which progresses at a constant rate and is not subject to the insertion of extra ticks to reflect leap seconds (as UTC is). A constant rate is needed for control algorithms which want to executed on a regular basis. Many clocks are also relative to system startup and used to measure the passage of time, not calendar time
  - A count down clock which can be paused, continued or reset (e.g the clock which counts down to the launch of the Space Shuttle)
  - A CPU execution time clock which measures the amount of CPU time that is being consumed by a particular thread or object

Introduction II

- All of the above clocks need a resolution which is finer than the millisecond level
- They may all be based on the same underlying physical clock, or be supported by separate clocks
- Where more than one clock is provided, issues of the relationship between them may be importance; in particular, whether their values can drift

Basic Model

- A hierarchy of time classes rooted at **`HighResolutionTime`**
- This abstract class has three subclasses:
  - one which represents absolute time
  - one which represents relative time
  - and one which represents rational time
- The intention is to allow support for time values down to the nanosecond accuracy
- Clocks are supported through an abstract **`Clock`** class

High Resolution Time I

```
public abstract class HighResolutionTime
        implements Comparable, Cloneable {

  public abstract AbsoluteTime absolute(Clock clock);
  public int compareTo(HighResolutionTime time);
  public boolean equals(HighResolutionTime time);
  public final long getMilliseconds();
  public final int getNanoseconds();
  public abstract RelativeTime relative(Clock clock);
  public void set(HighResolutionTime time);
  public void set(long millis, int nanos);
  public static void waitForObject(Object target,
    HighResolutionTime time) throws InterruptedException;
```

High Resolution Time II

- The abstract methods allow time types that are relative to be re-expressed as absolute time values and vice versa.
- They allow clocks associated with the values to be changed
  - *absolute to absolute* — value returned has the same millisecond and nanosecond components as the encapsulated time value
  - *absolute to relative* — value returned is the value of the encapsulated absolute time minus the current time as measured from the given clock parameter
  - *relative to relative* — value returned has the same millisecond and nanosecond components as the encapsulated time value
  - *relative to absolute* — value returned is the value of current time as measured from the given clock parameter plus the encapsulated relative time

High Resolution Time III

- Changing the clock associated with a time value is potentially unsafe, particularly for absolute time values
  - This is because absolute time values are represented as a number of milliseconds and nanoseconds since an epoch
  - Different clocks may have different epochs.
- The `waitForObject` does not resolve the problem of determining if the schedulable object was woken by a notify method call or by a timeout
  - It does allow both relative and absolute time values to be specified.

Absolute Time

```
public class AbsoluteTime extends HighResolutionTime {
  public AbsoluteTime();
  public AbsoluteTime(AbsoluteTime time);
  public AbsoluteTime(Date date);
  public AbsoluteTime(long millis, int nanos);

  public AbsoluteTime absolute(Clock clock);

  public AbsoluteTime add(long millis, int nanos);
  public final AbsoluteTime add(RelativeTime time);

  public java.util.Date getDate();
  public RelativeTime relative(Clock clock);
  public void set(Date date);

  public RelativeTime subtract(AbsoluteTime time);
  public AbsoluteTime subtract(RelativeTime time);
```

Note that an absolute time can have either a positive or a negative value and that, by default, it is relative to the epoch of the real-time clock

Relative Time

```
public class RelativeTime extends HighResolutionTime {
  public RelativeTime();
  public RelativeTime(long millis, int nanos);
  public RelativeTime(RelativeTime time);

  public AbsoluteTime absolute(Clock clock);
  public RelativeTime add(long millis, int nanos);
  public RelativeTime add(RelativeTime time);
  public RelativeTime relative(Clock clock);
  public RelativeTime subtract(RelativeTime time)
```

A relative time value is an interval of time measured by some clock; for example 20 milliseconds

Clocks

- `Clock` is the abstract class from which all clocks are derived
- RTSJ allows many different types of clocks; eg, an execution-time clock which measures the amount of execution time consumed
- There is real-time clock which advances monotonically
  - can never go backwards
  - should progress uniformly and not experience insertion of leap ticks

```
public abstract class Clock {
  public Clock();
  public static Clock getRealtimeClock();
  public abstract RelativeTime getResolution();
  public AbsoluteTime getTime();
  public abstract void getTime(AbsoluteTime time);
  public abstract void setResolution(RelativeTime resolutn);
```

Example: measuring elapse time

```
Clock clock = Clock.getRealtimeClock();
AbsoluteTime oldt = clock.getTime();
...
AbsoluteTime newt = clock.getTime();
RelativeTime inter = newt.subtract(oldt);
```

This would only measure the approximate elapse time, as the schedulable object executing the code may be pre-empted after it has finished the computation and before it reads the new time

Example

- A launch clock is a clock which is initiated with a relative and an absolute time value
- The absolute time is the time at which the clock is to start ticking; the relative time is the duration of the countdown
- The count down can be stopped, restarted, or reset
- The class extends the **`Thread`** class
- The constructor saves the start time, the duration and the clock
- The resolution of the count down is one second

LaunchClock I

```
class LaunchClock extends Thread {
  private AbsoluteTime start;
  private RelativeTime remaining, tick;
  private Clock clock;
  private boolean counting = true;
  private boolean go;

  LaunchClock(AbsoluteTime at,RelativeTime count) {
    start = at; remaining = count;
    clock = Clock.getRealtimeClock();
    tick = new RelativeTime(1000,0);
  }
  RelativeTime getResolution(){ return tick; }

  synchronized AbsoluteTime getCurrentLaunchTime(){
    return new AbsoluteTime(clock.getTime().add(remaining))
    // assumes started and ticking
  }
```

LaunchClock II

```
synchronized void stopCountDown(){
  counting = false;  notifyAll();
}
synchronized void restartCountDown() {
  counting = true;  notifyAll();
}
synchronized void resetCountDown(RelativeTime to){
  remaining = to;
}

synchronized void launch() throws Exception{
  while(!go)
      try {  wait(); }
      catch(InterruptedException ie) {
          throw new Exception("Launch failed");
      }// Launch is go
}
```

LaunchClock V

```
public void run(){
  try  {
    synchronized(this) {
      while(clock.getTime().compareTo(start) < 0)
         HighResolutionTime.waitForObject(this, start);
      while(remaining.getMilliseconds() > 0) {
          while(!counting) wait();
          HighResolutionTime.waitForObject(this, tick);
          remainin.set(remaining.getMilliseconds() -
                       tick.getMilliseconds());
      }
      go = true;
      notifyAll();
    }
  } catch(InterruptedException ie) {}
}
```

Summary

- Clocks and time are fundamental to any real-time system
- The RTSJ has augmented the Java facilities with high resolution time types and a framework for supporting various clocks
- A real-time clock is guaranteed to be supported by any compliant RTSJ implementation
- This is a monotonically non-decreasing clock