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

Asynchronous Event and their Handlers

Lecture aims:

- To motivate the needs for asynchronous events
- To introduce the basic RTSJ model
- To show how to implement Asynchronous events with parameters
- To consider event handlers and program termination

Time Triggered Systems

- Within embedded system design, the controllers for real world objects such as conveyor belts, engines and robots are usually represented as threads
- The interaction between the real world objects and their controllers can be either time-triggered or event-triggered
- In a time-triggered systems, the controller is activated periodically; it senses the environment in order to determining the status of the real-time objects it is controlling
- It writes to actuators which affect the behavior of the objects
- *E.g, a robot controller determines the position of the robot via a sensor and decide that it must cut the power to a motor thereby bringing the robot to a halt*

Event Triggered Systems

- In an event triggered system, sensors in the environment are activated when the real world object enters into certain states
- The events are signaled to the controller via interrupts
- *Eg, a robot may trip a switch when it reaches a certain position; this is a signal to the controller that the power to the motor should be turn off, thereby bringing the robot to a halt*
- Event triggered systems are often more flexible whereas time triggered systems are more predictable
- In either case, the controller is usually represented as a thread

Threads Considered Harmful

- There are occasions where threads are not appropriate:
  - the external objects are many and their control algorithms are simple and non-blocking
    - using a thread per controller leads to a proliferation of threads along with the associated per-thread overhead
  - the external objects are inter-related and their collective control requires significant communication/synchronization between controllers
    - complex communication and synchronization protocols are needed which can be difficult to design correctly and may lead to deadlock or unbounded blocking

Event-based Programming

- An alternative to thread-based programming
- Each event has an associated handler; when events occur, they are queued and one or more server thread takes an event from the queue and executes it associated handler
- When the handler has finished, the server takes another event from the queue, executes the handler and so on
- There is no need for explicit communication between the handlers as they can simply read/write from shared objects without contention

Disadvantages of Events

- Disadvantages of controlling all external objects by event handlers:
  - it is difficult to have tight deadlines associated with event handlers as a long-lived and non-blocking handler must terminate before the server can execute any newly arrived high-priority handler, and
  - it is difficult to execute the handlers on a multiprocessor system as the handlers assume no contention for shared resources

The RTSJ Approach

- Attempt to provide the flexibility of threads and the efficiency of event handling via the notion of real-time asynchronous events (AE) and handlers (AEH)
- AEs are data-less happenings either fired by the program or associated with the occurrence of interrupts in the environment
- One or more AEH can be associated with a single event, and a single AEH can be associated with one or more events
- The association between AEHs and AEs is dynamic
- Each AEH has a count of the number of outstanding firings. When an event is fired, the count is atomically incremented
- The handler is then released

The AE Class

```
public class AsyncEvent {

  public void addHandler(AsyncEventHandler handler);
  public void removeHandler(AsyncEventHandler handler);
  public void setHandler(AsyncEventHandler handler);
  public boolean handledBy(AsyncEventHandler target);
  public void bindTo(String happening)
          throws UnknownHappeningException;
  public void unBindTo(String happening)
          throws UnknownHappeningException;
  public ReleaseParameters createReleaseParameters();
  public void fire();
}
```


The AEH Class

```
public class AsyncEventHandler implements Schedulable {
  public AsyncEventHandler();
  public AsyncEventHandler(Runnable logic);
  public AsyncEventHandler(boolean nonheap);
  public AsyncEventHandler(boolean nonheap, Runnable logic);
  public AsyncEventHandler(SchedulingParameters s,
     ReleaseParameters r, MemoryParameters m, MemoryArea a);
  ... // various other combinations

  protected final int getAndClearPendingFireCount();
  protected int getAndDecrementPendingFireCount();
  protected int getAndIncrementPendingFireCount();
  protected final int getPendingFireCount();

  public void handleAsyncEvent();
  public final void run();
```

ASEH

- A set of protected methods allow the fire count to be manipulated
- They can only be called by creating a subclass and overriding the `handlerAsyncEvent` method
- The default code for `handleAsyncEvent` is null unless a `Runnable` object has been supplied with the constructor, in which case, the `run` method of the `Runnable` object is called
- The `run` method of the `AsyncEventHandler` class itself is the method that will be called by the underlying system when the object is first released
- It will call `handleAsyncEvent` repeatedly whenever the fire count > 0

Bound Event Handlers

- Both event handlers and threads are schedulable objects
- Threads provide the vehicles for execution of event handlers
- Therefore, an event handler is to a server real-time thread
- For `AsyncEventHandler` objects binding is dynamic
- There is some overhead with doing this and `BoundEvent-Handler` objects are supplied to eliminate this overhead
- Bound event handlers are permanently associated with a dedicated server real-time thread

BASEH

```
class BoundAsyncEventHandler extends AsyncEventHandler {
  public BoundAsyncEventHandler();
  public BoundAsyncEventHandler(
      SchedulingParameters scheduling,
      ReleaseParameters release, MemoryParameters memory,
      MemoryArea area, ProcessingGroupParameters group,
      boolean nonheap);
}
```


Timers

- The abstract `Timer` class defines the base class from which timer events can be generated
- All timers are based on a clock; a null clock values indicates that the `RealtimeClock` should be used
- A timer has a time at which it should fire; that is release its associated handlers
- This time may be an absolute or relative time value
- If no handlers have been associated with the timer, nothing will happen when the timer fires

Timer

```
public abstract class Timer extends AsyncEvent {
  protected Timer(HighResolutionTime time, Clock clock,
                  AsyncEventHandler handler);

  public ReleaseParameters createReleaseParameters();
  public void destroy();
  public void disable();
  public void enable();
  public Clock getClock();
  public AbsoluteTime getFireTime();
  public void reschedule(HighResolutionTime time);
  public void start();
  public boolean stop();
}
```


Timers

- Once created a timer can be explicitly destroyed, disabled (which allows the timer to continue counting down but prevents it from firing) and enabled (after it has been disabled)
- If a timer is enabled after its firing time has passed, the firing is lost
- The `reschedule` method allows the firing time to be changed
- Finally the `start` method, starts the timer going
- Any relative time given in the constructor is converted to an absolute time at this point; if an absolute time was given in the constructor, and the time has passed, the timer fires immediately

One Shot Timer

```
public class OneShotTimer extends Timer {
  public OneShotTimer(HighResolutionTime fireTime,
                      AsyncEventHandler handler);
  // assumes the default real-time clock

  public OneShotTimer(HighResolutionTime fireTime,
                      Clock clock, AsyncEventHandler handler);
```


Periodic Timer

```
public class PeriodicTimer extends Timer {
  public PeriodicTimer(HighResolutionTime start,
       RelativeTime interval, AsyncEventHandler handler);
  public PeriodicTimer(HighResolutionTime start,
       RelativeTime interval, Clock clock,
       AsyncEventHandler handler);

  public ReleaseParameters createReleaseParameters();
  public void fire(); // deprecated
  public AbsoluteTime getFireTime();
  public RelativeTime getInterval();
  public void setInterval(RelativeTime interval);
```

Example: A Panic Button

- Consider a computerized hospital intensive care unit
- A patient's vital signs are automatically monitored and if there is cause for concern, a duty doctor is paged automatically
- There is also a bed-side "panic" button which can be pushed by the patient or a visitor should they feel it is necessary
- The "panic" button is mainly for the patient/visitor's benefit; if the patient's life is really in danger, other sensors will have detected the problem

Panic Button

- To be on the safe side, the system responds to a press of the panic button in the following way:
  - if there is no paging of the doctor in the last five minutes, test to see if the patient's vital signs are strong, if they are weak, the duty doctor is paged immediately
  - if the vital signs are strong and a nurse has been paged in the last ten minutes, the button is ignored
  - if the vital signs are strong and a nurse has not been paged in the last ten minutes, the duty nurse is paged

Panic Button

- The press of the "panic" button is an external happening
- It is identified by the string `"PanicButton"`
- A pager is represented by an asynchronous event; `dutyDoctor` and `dutyNurse` are the events for the doctor's and nurse's pages respectively
- A firing of the appropriate event results in the associated handler initiating the paging call
- First the event handler for the "panic button" can be defined
- The constructor attaches itself to the "panic button" event
- Note also that the handler clears the fire count as it is possible that the patient/visitor has pressed the button multiple times

PanicButtonHandler I

```
class PanicButtonHandler extends AsyncEventHandler {
  private AbsoluteTime lastPage = new AbsoluteTime(0,0);
  private Clock clock = Clock.getRealtimeClock();
  private final long nursePagesGap = 600000; // 10 mins
  private final long doctorPagesGap = 300000;// 5 mins
  private AsyncEvent nursePager,doctorPager;
  private PatientVitalSignsMonitor patient;

  PanicButtonHandler(AsyncEvent button, AsyncEvent n,
                     AsyncEvent d, PatientMonitor s) {
    nursePager = n;doctorPager = d; patient = s;
    button.addHandler(this);
  }
```

PanicButtonHandler II

```
void handleAsyncEvent() {

    RelativeTime last=clock.getTime().subtract(lastPage);
    if(last.getMilliseconds() > doctorPagesGap) {
      if(!patient.vitalSignsGood()) {
        lastPage = clock.getTime(); doctorPager.fire();
      } else if(last.getMilliseconds()>nursePagesGap){
          lastPage = clock.getTime(); nursePager.fire();
      }
    getAndClearPendingFireCount();
}
```


Configuration

```
AsyncEvent nursePager = new AsynEvent();
AsyncEvent doctorPager = new AsynEvent();
PatientMonitor signs = new ... ;
PriorityParameters priority = new ...;
AsyncEvent panicButton;
ImmortalMemory im = ImmortalMemory.instance();
im.enter( new Runnable() { public void run(){
     panicButton  = new AsyncEvent();
     AsyncEventHandler handler = new PanicButtonHandler(
            panicButton,nursePager,doctorPager,signs);
     handler.setSchedulingParameters(
           new PriorityParameters(priority));
     handler.setReleaseParameters(
           panicButton.createReleaseParameters());
     if(!handler.addToFeasibility()) { outputwarning }
     panicButton.bindTo("PanicButton");//start monitoring
} } );
```

Configuration

- Note, as the asynchronous event is being bound to an external happening, the object is created in immortal memory
- The panicButton reference can be in any scope convenient for the program
- However, as the event will need to reference the handler, the handler too must be in immortal memory
- Given that the handler is a Schedulable object, its scheduling and release parameters must be defined
- The release parameters will be aperiodic in this case

Event Handlers and Termination

- Many real-time systems do not terminate. However, it is necessary to define under what conditions a program terminates
- In Java, threads are classified as being daemon or user threads; the program terminates when all user threads have terminated; the daemon threads are destroyed
- The server threads used to execute asynchronous event handlers are daemon threads
- This means that when all user real-time threads are terminated, the program will still terminate
- However, where events are bound to happenings in the environment or timers, the program may not execute as the programmer intended

Summary

- Event-based systems are supported by the `AsyncEvent` and `AsyncEventHandler` class hierarchies
- Event handlers are schedulable entities and consequently can be given release and scheduling parameters
- Periodic, one-shot timers along with interrupt handlers are supported
- Care must be taken as an implementation may use daemon server threads; these may, therefore, be terminated when the programmer does not expect it

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

Real-Time Threads

Lecture aims:

- To introduce the basic RTSJ model for real-time threads
- To explain the concept of a NoHeapRealtimeThread
- To evaluate the support for periodic, sporadic and aperiodic threads
- Example: monitoring deadline miss

Introduction

- For real-time systems, it is necessary to model the activities in the controlled system with concurrent entities in the program
- Standard Java supports the notion of a thread, however, in practice Java threads are both too general and yet not expressive enough to capture the properties of real-time activities
- Real-time activities are also usually characterized by their execution patterns: being periodic, sporadic or aperiodic; representing these in standard Java can only be done by coding conventions in the program
- Such conventions are error prone and obscure the true nature of the application

The Basic Model

- Two classes: `RealtimeThread` and `NoHeapRealtimeThread`
- Real-time threads are schedulable objects and, therefore, can have associated release, scheduling, memory and processing group parameters
- A real-time thread can also have its memory area set

By default, a real-time thread inherits the parameters of its parent. If the parent has no scheduling parameters (because it was a plain Java thread), the scheduler's default value is used. For the priority scheduler, this is the normal priority.

The RealtimeThread Class

```
class RealtimeThread extends Thread implements Schedulable {
  public RealtimeThread(SchedulingParameters s);
  public RealtimeThread(SchedulingParameters s,
                        ReleaseParameters r);
  public RealtimeThread(SchedulingParameters s,
      ReleaseParameters r, MemoryParameters m,
      MemoryArea a, ProcessingGroupParameters g,
      Runnable logic);
  public static MemoryArea getCurrentMemoryArea();
  public MemoryArea getMemoryArea();
  public static MemoryArea getOuterMemoryArea(int index);
  public static int getInitialMemoryAreaIndex();
  public static int getMemoryAreaStackDepth();
  public static void sleep(Clock clock, HighResolutionTime t)
                                    throws InterruptedException;
  public static void sleep(HighResolutionTime time) throws …
  public void start();
  public static RealtimeThread currentRealtimeThread();
```

The RealtimeThread Class II

```
public class RealtimeThread extends ...
  ...
  public boolean waitForNextPeriod()
         throws IllegalThreadStateException;
  public void deschedulePeriodic();
  public void schedulePeriodic();
```

The meaning of these methods depends on the thread's scheduler. The following definitions are for the base priority scheduler.

Support for Periodic Threads

- The **`waitForNextPeriod`** method suspends the thread until its next release time (unless the thread has missed its deadline)
- The call returns true when the thread is next released; if the thread is not a periodic thread, an exception is thrown
- The **`deschedulePeriodic`** method will cause the associated thread to block at the end of its current release (when it calls `wFNP`); it will then remain descheduled until **`schedulerPeriodic`** is called
- When a periodic thread is "rescheduled" in this manner, the scheduler is informed so that it can remove or add the thread to the list of schedulable objects it is managing

Support for Periodic Threads

- The `ReleaseParameters` associated with a real-time thread can specify asynchronous event handlers which are scheduled by the system if the thread misses its deadline or overruns its cost allocation
- For deadline miss, no action is immediately performed on the thread itself.
- It is up to the handlers to undertake recovery operations
- If no handlers have been defined, a count is kept of the number of missed deadlines

Support for Periodic Threads

- The `waitForNextPeriod` (wFNP) method is for use by real-time threads that have periodic release parameters
- Its behavior can be described in terms of the following attributes:
  - `lastReturn` — indicates the last return value from wFNP
  - `missCount` — indicates the how many deadlines have been missed (for which no event handler has been released)
  - `descheduled` — indicates the thread should be descheduled at the end of its current release
  - `pendingReleases`— indicates number of releases that are pending

Support for Periodic Threads

- Schedulable objects (SO) have 3 states:
  - Blocked means the SO cannot be selected to have its state changed to executing; the reason may be blocked-for-I/O-completion,
    - blocked-for-release-event
    - blocked-for-reschedule
    - blocked-for-cost-replenishment
  - Eligible-for-execution means the SO can have its state changed to executing
  - Executing means the program counter in a processor holds an instruction address within the SO

Support for Periodic Threads

- On each deadline miss:
  - if the thread has a deadline miss handler, `descheduled` := true and the deadline miss handler is released with a `fireCount` increased by `missCount+1`
  - Otherwise, the `missCount` is incremented

Support for Periodic Threads

- On each cost overrun:
  - if the thread has an overrun handler, it is released
  - if the next release event has not already occurred (`pendingReleases == 0`)
    - and the thread is Eligible-for-execution or Executing , the thread becomes blocked (blocked_for_cost_replenishment)
    - otherwise, it is already blocked, so the state transition is deferred
  - otherwise (a release has occurred), the cost is replenished

Support for Periodic Threads

- When each period is due:
  - if the thread is waiting for its next release (`blocked_for_release_event`)
    - if **`descheduled`** == true, nothing happens
    - otherwise, the thread is made eligible for execution, the cost budget is replenished and `pendingReleases` is incremented
  - Otherwise: `pendingReleases` is incremented, and
    - if the thread is blocked_for_cost_replenishment, it is made Eligible-for-execution and rescheduled

Support for Periodic Threads

- When the thread's `schedulePeriodic` method is invoked:
  - `descheduled` is set to false
  - if the thread is blocked-for-reschedule, `pendingReleases` is set to zero and it is made blocked-for-release-event
- When the thread's `deschedulePeriodic` method is invoked:
  - `descheduled` is set to true

Support for Periodic Threads

- When the thread's cost parameter changes:
  - if the thread's cost budget is depleted and the thread is currently eligible for execution, a cost overrun for the thread is triggered
  - if the cost budget is not depleted and the thread is currently blocked-for-cost-replenishment, the thread is made eligible for execution

Support for Periodic Threads

- The `waitForNextPeriod` method has three possible behaviors depending on the state of `missCount`, `descheduled` and `pendingReleases`
- If `missCount >0:`
  - `missCount--`
  - if `lastReturn` is false, `pendingReleases` is decremented and false is returned (this indicates that the next release has already missed its deadline), a new cost budget is allocated
  - `lastReturn` is set to false and false is returned (this indicates that the current release has missed its deadline)

Support for Periodic Threads

- else, if **`descheduled`** is true:
  - the thread is made blocked-for-reschedule until it is notified by a call to `schedulePeriodic`
  - then it becomes blocked-for-release-event, when the release occurs the thread becomes eligible for execution
  - `pendingReleases--`
  - `lastReturn` is set to true and true is returned
- Otherwise,
  - if **`pendingReleases>=0`**:
    - the thread becomes blocked-for-release-event, when the release occurs the thread becomes eligible for execution
  - `lastReturn` = true, `pendingReleases--`, true is returned

Periodic Threads Summary

- If there are no handlers, `wFNP` will not block the thread in the event of a deadline miss. It is assumed that in this situation the thread itself will take some corrective action
- Where the handler is available, it is assumed that the handlers will take some corrective action and then reschedule the thread by calling **`schedulePeriodic`**

NoHeapRealtimeThread

- One of the main weaknesses with standard Java, from a real-time perspective, is that threads can be arbitrarily delayed by the action of the garbage collector
- The RTSJ has attacked this problem by allowing objects to be created in memory areas other than the heap
- These areas are not subject to GC
- A no-heap real-time thread NHRT is a real-time thread which only ever accesses non-heap memory areas
- Hence, it can safely be executed even when GC is occurring

NoHeapRealtimeThread I

- The `NHRT` constructors contain references to a memory area; all memory allocation performed by the thread will be from within this memory area
- An unchecked exception is thrown if the `HeapMemory` area is passed
- The `start` method is redefined; its goal is to check that the `NHRT` has not been allocated on the heap and that it has obtained no heap-allocated parameters
- If either of these requirements have been violated, an unchecked exception is thrown

NoHeapRealtimeThread Class

```
public class NoHeapRealtimeThread extends RealtimeThread {
  public NoHeapRealtimeThread(
      SchedulingParameters scheduling, MemoryArea area);
  public NoHeapRealtimeThread(
      SchedulingParameters scheduling,
      ReleaseParameters release, MemoryArea area);
  public NoHeapRealtimeThread(
      SchedulingParameters scheduling,
      ReleaseParameters release, MemoryParameters memory,
      MemoryArea area, ProcessingGroupParameters group,
      java.lang.Runnable logic);
```

A Simple Model of Periodic Threads

```
class Periodic extends RealtimeThread {

  Periodic(PriorityParameters pp, PeriodicParameters P)
  { super(pp, p); };

  public void run() {
   boolean noProblems = true;
   while(noProblems) { // code to be run each period
     ...
     noProblems = waitForNextPeriod();
   }
   // a deadline has been missed, and there is no handler
   ...
  }
}
```


The Model of Sporadic/Aperiodic Threads

- Unlike periodic threads, their release parameters have no start time, so they can be considered to be released as soon as they are started
- However, how do they indicate to the scheduler that they have completed their execution and how are they re-released?
- There is no equivalent of wFNP (contrast this with sporadic and asynchronous event handlers which have a **`fire`** method and a **`handleAsyncEvent`** method executed for each call of **`fire`**)
- The answer to this question appears to be that you have to provide your own!

Monitoring Deadline Miss I

- In many soft real-time systems, applications will want to monitor any deadline misses and cost overruns but take no action unless a certain threshold is reached
- Consider: deadline miss monitor

```
public class HealthMonitor{
  public void persistentDeadlineMiss(Schedulable s);
}
```


Monitoring Deadline Miss II

```
class DeadlineMissHandler extends AsyncEventHandler {
 private RealtimeThread myrt;
 private int missed, myThreshold;
 private HealthMonitor myhm;

 DeadlineMissHandler(HealthMonitor mon, int threshold) {
   super(new PriorityParameters(
         PriorityScheduler.MAX_PRIORITY),
         null, null, null, null, null);
   myhm = mon; myThreshold = threshold;
 }
 public void setThread(RealtimeThread rt){ myrt = rt;}

 public void handleAsyncEvent() {
   if(++missed < myThreshold)
       myrt.schedulePeriodic();
   else myhm.persistentDeadlineMiss(myrt);
 }
```

Monitoring Deadline Miss

The deadline miss handler is as schedulable object. Consequently, it will compete with its associated schedulable object according to their priority. Hence, for the handler to have an impact on the errant real-time thread, it must have a priority higher than the thread. If the priority is lower, it will not run until the errant thread blocks.

Summary

- The RTSJ supports the notion of schedulable objects with various types of release characteristics
- We have reviewed two type of schedulable objects: real-time threads and no-heap real-time threads
- We have illustrated that periodic activities are well catered for
- However, the support for aperiodic and sporadic activities is lacking
- It is currently not possible for the RTSJ to detect either deadline miss or cost overruns for these activities, as there is no notion of a release event
- Indeed, programmers are best advised to use event handlers to represent non-periodic activities

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

Asynchronous Transfer of Control

- Lecture aims:
- To introduce the application requirements for ATC
- To explain the basic RTSJ Asynchronous Transfer of Control (ATC) model

Introduction

- An asynchronous transfer of control (ATC) is where the point of execution of one schedulable object is changed by the action of another schedulable object
- Consequently, a SO may be executing one method and then suddenly, through no action of its own, find itself executing another
- Controversial because
  - complicates the language's semantics
  - makes it difficult to write correct code as the code may be subject to interference
  - increases the complexity of JVM
  - may slow down the execution of code which doesn't use the feature

The Application Requirements for ATC

- Fundamental requirement: to enable a process to respond quickly to a condition detected by another process
- Error recovery — to support coordinated error recovery between real-time threads
  - Where several threads are collectively solving a problem, an error detected by one thread may need to be quickly and safely communicated to the other threads
  - These types of activities are often called atomic actions
  - An error detected in one thread requires all other threads to participate in the recovery
  - *E.g, a hardware fault detected by a thread may mean that other threads will never finish executing because the preconditions under which they started no longer hold; they may never reach reach their polling point*

Mode Changes

- Mode changes — where changes between modes are expected but cannot be planned
  - a fault may lead to an aircraft abandoning its take-off and entering into an emergency mode of operation
  - an accident in a manufacturing process may require an immediate mode change to ensure an orderly shutdown of the plant
- The processes must be quickly and safely informed that the mode in which they are operating has changed, and that they now need to undertake a different set of actions

Scheduling and Interrupts

- Scheduling using partial/imprecise computations — there are many algorithms where the accuracy of the results depends on how much time can be allocated to their calculation
  - numerical computations, statistical estimations and heuristic searches may all produce an initial estimation of the required result, and then refine that result to a greater accuracy
  - at run-time, a certain amount of time can be allocated to an algorithm, and then, when that time has been used, the process must be interrupted to stop further refinement of the result
- User interrupts — In an interactive environment, users often wish to stop the current processing because they have detected an error condition and wish to start again

Polling ands Aborts

Polling for the notification is too slow. One approach to ATC is to destroy the thread and allow another thread to perform some recovery. All operating systems and most concurrent programming languages provide such a facility. However, destroying a thread can be expensive and is often an extreme response to many error conditions. Furthermore, it may leave the system in an inconsistent state (for example, monitor locks may not be released). Consequently, some form of controlled ATC mechanism is required.

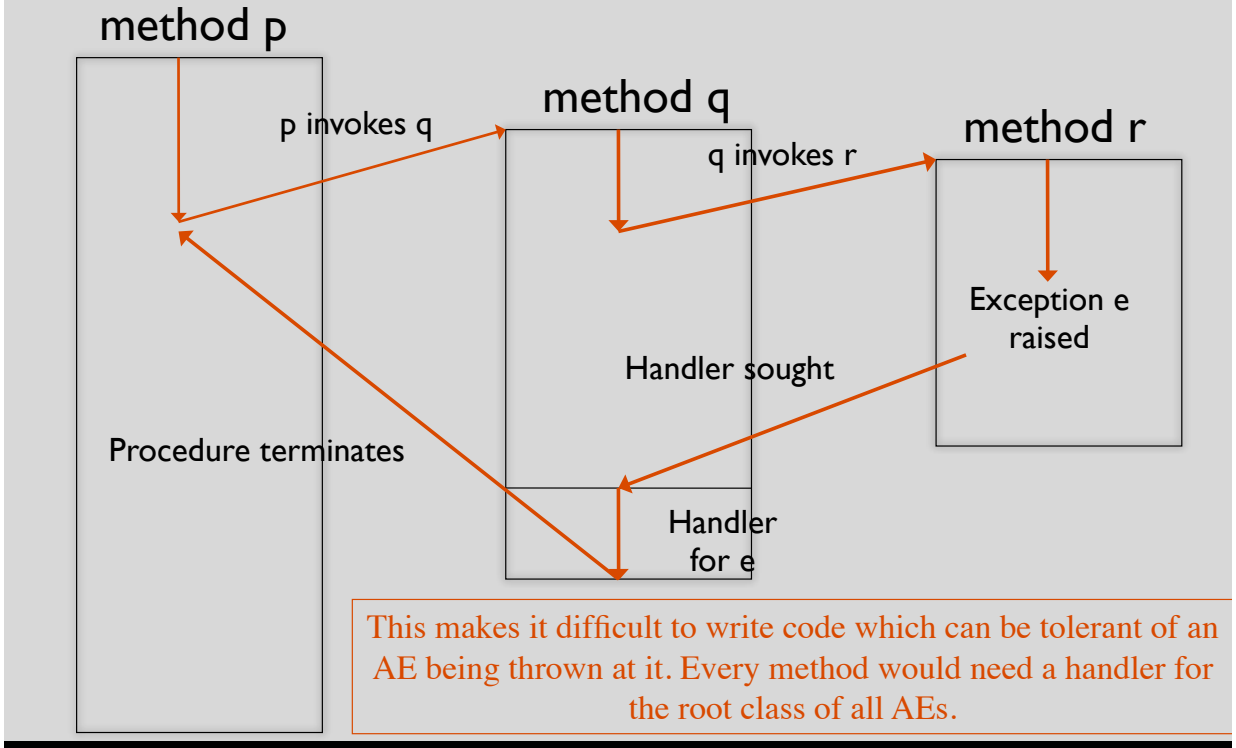
The Basic Model

- Brings together the Java exception handling model and an extension of thread interruption
- When a real-time thread is interrupted, an asynchronous exception is thrown at the thread rather than the thread having to poll for the interruption as with standard Java
- The notion of an asynchronous exception is not new and has been explored in previous languages

Problems with Asynchronous Exceptions

- The problem is how to program safely in their presence
- Most exception handling mechanisms have exception propagation with a termination model
- Consider a method `p`, which has called `q` which called `r`
- When an exception is raised within `r`, if there is no local handler, the call to `r` is terminated and a handler is sought in `q` (the exception propagates up the call chain)
- If no handler is found in `q`, the exception is propagated to `p`.
- When a handler is found, it is executed and the program continues to execute in the handler's context. There is no return to the context where the original exception was thrown.

The Termination Model



This makes it difficult to write code which can be tolerant of an AE being thrown at it. Every method would need a handler for the root class of all AEs.

ATC

- The RTSJ solution is to require that all methods, which are writen to handle an asynchronous exception, place the exception in their **`throws`** list
- These are called AI-methods (Asynchronously Interruptible)
- If a method does not do this then the asynchronous exception is not delivered but held pending until the thread is in a method which has the appropriate `throws` clause
- Hence, code written without being concerned with ATC can execute safely in an environment where ATCs are used

ATC

- To ensure that ATC can be handled safely, the RTSJ requires that
  - ATCs are deferred during the execution of synchronized methods or statements (to ensure that any shared data is left in a consistent state); these sections of code and methods which are not AI methods are called ATC-deferred sections
  - an ATC only be handled from within code that is an ATC-deferred section; this is to avoid an ATC handler being interrupted by another ATC being thrown

ATC

- Use of ATC requires
  - declaring an `AsynchronouslyInterruptedException` (AIE)
  - identifying methods which can be interrupted using a throws clause
  - signaling an AIE to a schedulable object
- Calling `interrupt` "throws" the system's generic AIE

AIE

```
public class AsynchronouslyInterruptedException extends
              InterruptedException {
  ...
  public boolean enable();
  public void disable();
  public boolean doInterruptible (Interruptible logic);

  public boolean fire();
  public boolean clear();

  public static AsynchronouslyInterruptedException getGeneric();
  // returns the AsynchronouslyInterruptedException which
  // is generated when RealtimeThread.interrupt() is invoked
```

Example of ATC

```
import nonInterruptibleServices.*;

public class InterruptibleService {
  public boolean Service() throws AIE {
    //code interspersed with calls to nonInterruptibleServices
  }
  ...
}

public InterruptibleService IS = new InterruptibleService();

// code of real-time thread, t
if(IS.Service()) { ... } else { ... };

// now another real-time thread interrupts t:

t.interrupt();
```


Semantics: when AIE is fired

- If `t` within an ATC-deferred section the AIE is marked as pending
- If `t` is in a method which does not declare AIE in its throws list, the AIE is marked as pending
- A pending AIE is thrown as soon as `t` returns to (or enters) a method with an AIE declared in its throws list
- If `t` is blocked inside a **`sleep`** or **`join`** method called from within an AI-method, `t` is rescheduled and the AIE is thrown.
- If `t` is blocked inside a **`wait`** method or the **`sleep`** or **`join`** methods called from within an ATC-deferred region, `t` is rescheduled and the AIE is thrown as a synchronous exception and it is also marked as pending

Semantics

- Although AIEs appear integrated into the Java exception handling mechanism, the normal Java rules to not apply
- Only the naming of the AIE class in a throw clause indicates the thread is interruptible. It is not possible to use a subclass. Consequently, catch clauses for AIEs must name the class AIE explicitly and not a subclass
- Handlers for AIEs do not automatically stop the propagation of the AIE. It is necessary to call the clear method in the AIE class
- Although catch clauses in ATC-deferred regions that name the **`InterruptedException`** or **`Exception`** classes will handle an AIE this will not stop the propagation of the AIE

ECS 2009

Semantics

- Although AIE is a subclass of `InterruptedException` which is a subclass of `Exception`, catch clauses which name these classes in AI-methods will not catch an AIE
- Finally clauses that are declared in AI-methods are not executed when an ATC is thrown
- Where a synchronous exception is propagating into an AI-method, and there is a pending AIE, the synchronous exception is lost when the AIE is thrown

Catching an AIE

- Once an ATC has been thrown and control is passed to an exception handler, it is necessary to ascertain whether the caught ATC is the one expected by the interrupted thread
  - If it is, the exception can be handled.
  - If it is not, the exception should be propagated to the caller
- The `clear` method defined in the class `AsynchronouslyInterruptedException` is used for this purpose

Example Continued

```
import nonInterruptibleServices.*;

public class InterruptibleService {
  ...

  private AIE stopNow;

  public void useService() {
    stopNow = AIE.getGeneric();
    try {
      // call with calls to InterruptibleService
    } catch (AIE AI) {
      if(stopNow.clear()) { //handle the ATC }
      // No else clause, if the current exception is not
      // stopNow, it will remain pending
    }
  }
}
```

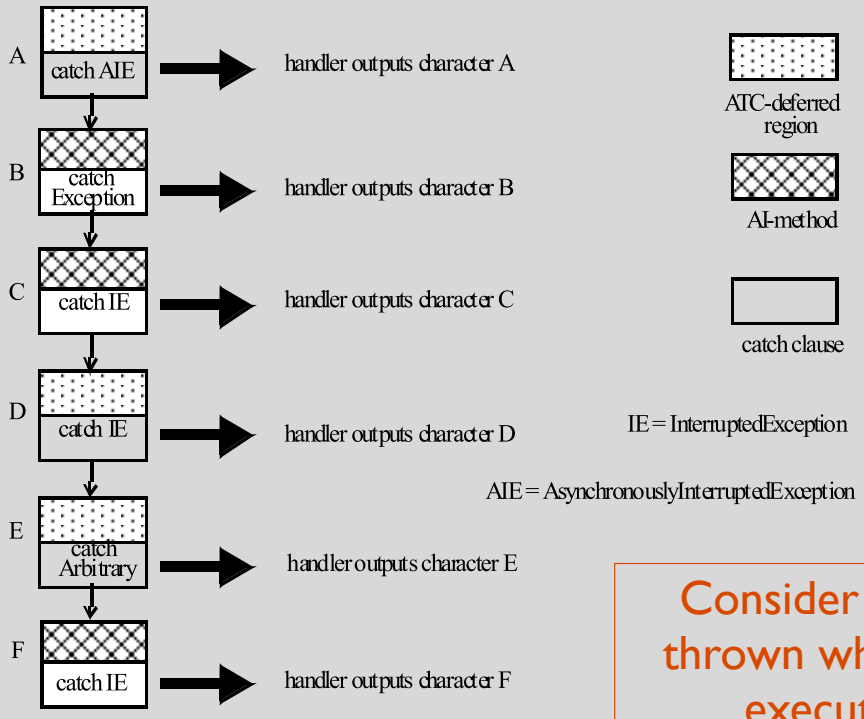
Alternative Handler

```
catch AIE AI {
  if(stopNow.clear()) {
    //handle the ATC
  } else {
    //cleanup
  }
}
```


Semantics of Clear

- If the AIE is the current AIE, the AIE is no longer pending; return true
- If the AIE isn't the current AIE, the AIE is still pending; return false

Example 1



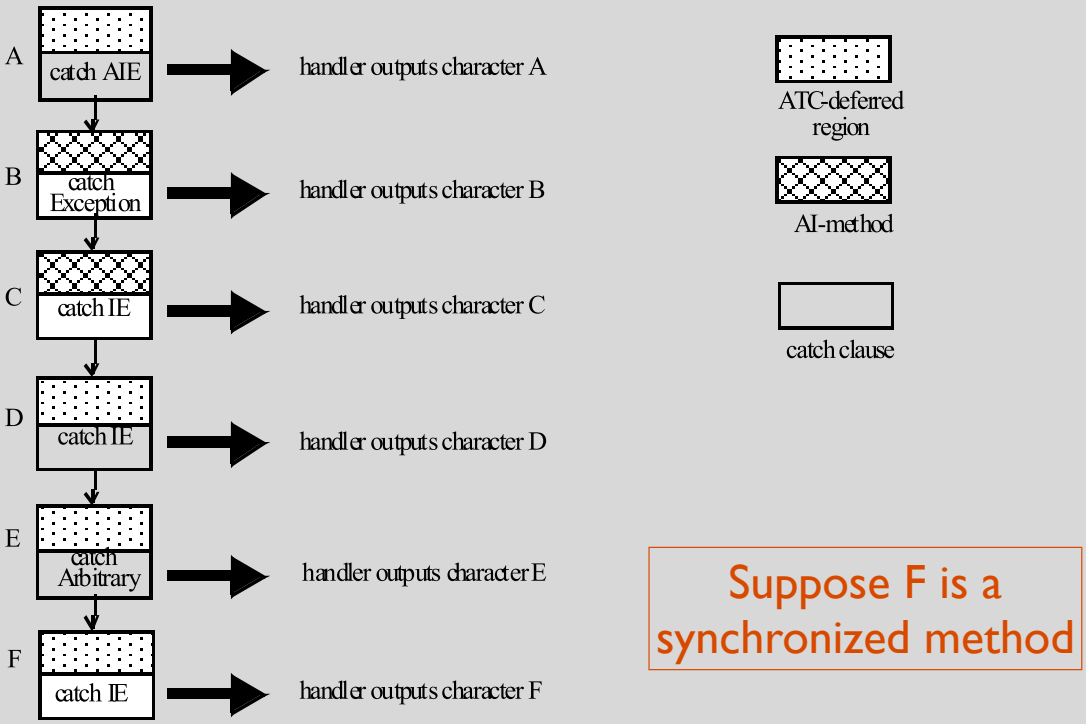
Consider an AIE thrown whilst F is executing

ECS 2009

Example 1

- The method F is terminated
- E is ATC-deferred but it doesn't have an AIE handler, E is terminated
- D is ATC-deferred and has a handler for the IE; the handler executes, however, the AIE remains pending
- When control returns to C, the AIE is thrown, C is terminated
- B is an AI-method so the exception propagates through it
- Control returns to A which is ATC-deferred with a AIE catch clause; this executes; if A calls `clear` on the current AIE, control will return to whoever called A and the AIE will have been handled and no longer pending. If the handler for the AIE is for a different AIE, the current AIE will remain pending
- The output will be the character 'D' followed by 'A'

Example 2



Suppose F is a synchronized method

Example2

- When the ATC is thrown, the AIE is marked as pending
- The AIE remains pending until control returns to C
- The AIE is then thrown and C is terminated as is the call to B
- Finally, the handler in A is executed; the output is 'A'
- If method F had been blocked on a call to the wait method when the ATC was thrown, the AIE is marked as pending and is thrown immediately
- This would be handled by the local IE catch clause
- However, the AIE is still marked as pending and will be thrown when control returns normally to method C
- When the AIE is thrown in C, the nearest handler is in A
- Output will, be the characters 'F' and 'A'

Summary I

- The ability to quickly gain the attention of a SO asynchronously is a requirement from the real-time community
- The modern approach to meeting this requirements is via an asynchronous transfer of control (ATC) facility
- RTSJ combines thread interruption with exception handling and introduces the notion of asynchronously interrupted exceptions
- A **`throws`** AIE in a method's signature indicates that the method is prepared to allow asynchronous transfers of control
- RTSJ requires that ATCs are deferred during the execution of `synchronized` methods or statements, and that an ATC only be handled from within code that is an ATC-deferred section

Summary II

- Java exception handling rules to not apply because:
  - Only AIE in a throw clause indicates the SO is interruptible
  - Handlers for AIE do not automatically stop the propagation of the AIE; it is necessary to call the clear method in the AIE class
  - Although `catch` clauses in ATC-deferred regions that name the IE or `Exception` classes handle an AIE this won't stop propagation
  - Although AIE is a subclass of IE and `Exception`, `catch` clauses which name these classes in AI-methods will not catch an AIE
  - **`finally`** clauses in AI-methods are not executed when an ATC is thrown
  - Where a synchronous exception is propagating into an AI-method, with a pending AIE, the synchronous exception is lost when the AIE is thrown

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

Asynchronous Transfer of Control

Lecture aims

- To explain the high level model and how to use the Interruptible interface
- To consider multiple `AsynchronouslyInterruptedExceptions` (AIE)
- To introduce the Timed class and show an example

ATC Summary

- All methods that are prepared to allow the delivery of an asynchronous exception must place the **`AIE`** exception in their **`throws`** list
- These methods are AI-methods (Asynchronously Interruptible)
- Other methods are called ATC-deferred and the asynchronous exception is not delivered but held pending until the thread is in a method which has the appropriate throws clause
- **`synchronized`** blocks are ATC deferred

AIE

```
public class AIE extends InterruptedException {
  ...
  public synchronized void disable();
  public boolean doInterruptible (Interruptible logic);

  public synchronized boolean enable();
  public synchronized boolean fire();

  public boolean clear ();

  public static AIE getGeneric();
  // returns the AIE which
  // generated when RealtimeThread.interrupt() is invoked
```

Interruptible

```
public interface Interruptible{
  public void interruptAction(AIE exception);
  public void run(AIE exception) throws AIE;
}
```

- An object which wishes to provide an interruptible method does so by implementing the **`Interruptible`** interface
- The **`run`** method is the method that is interruptible; the **`interruptedAction`** method is called by the system if the **`run`** method is interrupted

Interruptible

- Once this interface is implemented, the implementation can be passed as a parameter to the **`doInterruptible`** method in the AIE class
- The method can then be interrupted by calling the **`fire`** method in the AIE class
- Further control over the AIE is given by
  - **`disable`**
  - **`enable`**
  - **`isEnabled`**

Only one SO can be executing a **`doInterruptible`** at once

Warning

Note, the firing of an AIE has no effect if there is no currently active `doInterruptible`.

The firing is NOT persistent. Hence, care must be taken as there may be a race condition between one thread calling a doInterruptible and another thread calling fire on the same AIE.

To help cope with this race condition, fire will return false, if there is no currently active doInterruptible or the AIE is disabled.

Mode Change Example I

```
public class ModeA implements Interruptible {

  public void run(AIE aie) throws AIE  {
    // operation performed in Mode A
  }

  public void interruptAction(AIE aie) {
    // reset any internal state, so that when Mode A
    // becomes current, it can contine
  }
}

// similary for ModeB
```

Mode Change Example II

```
class ModeChanger extends AIE {
  static final int MODE_A = 0,MODE_B = 1;
  private int mode;

  ModeChanger(int initial) {
    mode = (initial==MODE_A)? MODE_A : MODE_B;
  }

  synchronized int currentMode(){ return mode; }
  synchronized void setMode(int nextMode) {
    if(nextMode == MODE_A | nextMode == MODE_B)
      mode = nextMode;
    else throw new IllegalArgumentException();
  }
  synchronized void toggleMode() {
    mode = (mode==MODE_A)?MODE_B:MODE_A;
  }
}
```


Mode Change Example IV

```
ModeChanger modeChange = newModeChanger(ModeChanger.MODE_A);

public void run() { // inside a RT thread with PeriodicParameters
  ModeA modeA = new ModeA();
  ModeB modeB = new ModeB();
  boolean ok = true;
  while(ok) {
    boolean _ =(modeChange.currentMode()==ModeChanger.MODE_A)?
         modeChange.doInterruptible(modeA):
         modeChange.doInterruptible(modeB); //throw away result
    ok = waitForNextPeriod();
  }
}
```

signaller:

```
modeChange.toggleMode();
modeChange.fire();
```

A Persistent AIE

```
public class PersistentAIE extends AIE {
  private volatile boolean outstandingAIE;

  public boolean fire() {
    if(super.fire()) return true;
    outstandingAIE = true;
    return false;
  }
 public boolean doInterruptible(Interruptible logic) {
    if(outstandingAIE) {
      outstandingAIE = false;
      logic.interruptAction(this);
      return true;
    }
    return super.doInterruptible(logic);
  }
}
```

Why does this fail?

A Persistent AIE

```
public boolean doInterruptible(Interruptible logic) {
  if(outstandingAIE) {
    outstandingAIE = false;
    logic.interruptAction(this);
    return true;
  }
  // fire comes in here!!
  return super.doInterruptible(logic);
}
```

A Persistent AIE

```
class PersistentAIE  extends AIE implements Interruptible {

  boolean fire(){ /* As before */ }
  public boolean doInterruptible(Interruptible logic) {
    this.logic = logic;
    return super.doInterruptible(this);
  }
  public void run(AIE aie) throws AIE {
    if(outstandingAIE) {
      outstandingAIE = false;
      super.fire();
    } else { logic.run(aie); }
  }
  public void interruptAction(AIE aie)
  { logic.interruptAction(aie); }

  private volatile boolean outstandingAIE;
  private Interruptible logic;
}
```


Nested ATCs

- Once the ATC-deferred region is exited, the currently pending AIE is thrown
- This AIE will be caught by the interruptAction of the inner most nested AIE first
- However, it will typically propagate the ATC

Nested ATC Example

```
class NestedATC {
  AIE AIE1 = new AIE();
  // similarly for AIE2 and AIE 3
  void method1()  { /* ATC-deferred region */ }

  void method2() throws AIE {
    AIE1.doInterruptible(new Interruptible(){
      public void run(AIE e) throws AIE {  method1(); }
      public void interruptAction( AIE e) {/* recovery*/}
    });
  }

  // similarly for method3, whose run method calls method2,
  // and for method4, whose run method calls method3
```


The Timed Class

- With Real-Time Java, there is a subclass of `AIE` called `Timed`
- Both absolute and Relative times can be used

```
public class Timed extends AsynchronouslyInterruptedException {
 public Timed(HighResolutionTime time)
             throws  IllegalArgumentException;
 public boolean doInterruptible(Interruptible logic);
 public void resetTime(HighResolutionTime time);
}
```

The Timed Class

- When an instance of the `Timed` class is created, a timer is created and associated with the time value passed as a parameter
- A timer value in the past results in the AIE being fired immediately the `doInterruptible` is called.
- When a time value is passed which is not in the past, the timer is started when the `doInterruptible` is called
- The timer can be reset for the next call to doInterruptible by use of the `resetTime` method

Imprecise Computation

- The algorithm consists of a mandatory part which computes an adequate, but imprecise, result
- An optional part then iteratively refines the results
- The optional part can be executed as part of a `doInterruptible` attached to a `Timed` object
- The `run` method updates the result from within a `synchronized` statement so that it is not interrupted

Imprecise Computation III

```
class ImpreciseResult {
  int value; // the result
  boolean precise; //is the value precise
}

class ImpreciseComputation {
private HighResolutionTime CompletionTime;
  private ImpreciseResult result;
  ImpreciseComputation(HighResolutionTime T) { {
    CompletionTime = T; result = new ImpreciseResult();
  }
  private int compulsoryPart() { … }
```


Imprecise Computation III

```
class ImpreciseComputation {
  ImpreciseResult service() { // public service
    result.precise = false;
    result.value = compulsoryPart();
    Interruptible I = new Interruptible() {
      public void run(AIE exception) throws AIE {
        // optional part which improves on the compulsory part
         boolean canBeImproved = true;
         while(canBeImproved) synchronized(result) {…}
         result.precise = true;
      }
      public void interruptAction(AIE exp) {
        result.precise = false; }
    });
    Timed t = new Timed(CompletionTime);
    boolean res = t.doInterruptible(I));
    // execute optional part, throw away result of doInterruptible
    return result;
  }
```

Summary

- The RTSJ combines thread interruption with exception handling and introduces the notion of an asynchronously interrupted exception (AIE)
- The presence of a throws `AsynchronouslyInterruptedException` in a method's signature indicates that the method is prepared to allow asynchronous transfers of control
- The high-level approach involves creating objects which implement the Interruptible interface
- Note, the firing of an `AsynchronouslyInterruptedException` has no effect if there is no currently active `doInterruptible`
- The firing is NOT persistent