Programming Embedded Systems

Lecture 13

An Intro to Real-time Java

Wednesday Feb 22, 2012

Philipp Rümmer
Uppsala University
Philipp.Ruemmer@it.uu.se

Lecture outline

- Background of real-time Java
- Threads, tasks, scheduling, priorities
- Monitors, queues
- Asynchronous events

Why consider Java at all for embedded systems?

- Buffer overflow and underflow
- Null object dereference
- Uninitialized variable
- Inappropriate cast
- Memory leaks

- **All common problems in low-level languages (C), under control in Java**

Why consider Java for embedded systems? (2)

- Typically:
  Implementation less time consuming in Java than in C
- Java can hide hardware/OS specifics
  → Applications can be ported more easily
- Java code is normally easier to read, easier to maintain, etc.
  (than C code)

Problems with standard Java for embedded applications

- No real-time features
  - No access to OS real-time scheduler
  - No avoidance of priority inversion
- Dynamic class loading, initialisation, compilation
- Garbage collection
- Cumbersome access to low-level hardware
- Standard JVMs are too large

Real-time Java

- **RTSJ**: real-time specification for Java
  - extends other Java frameworks, for instance Java SE, Java ME, Java EE
- Various implementations
  - **RTS**: Sun's/Oracle's RTSJ JVM
  - Various independent JVMs, JDKs: e.g., Timesys, IBM, jRate
  - Available for some micro-controllers: e.g., AVR ATmega8
- There are further RTSJ-independent real-time solutions based on Java

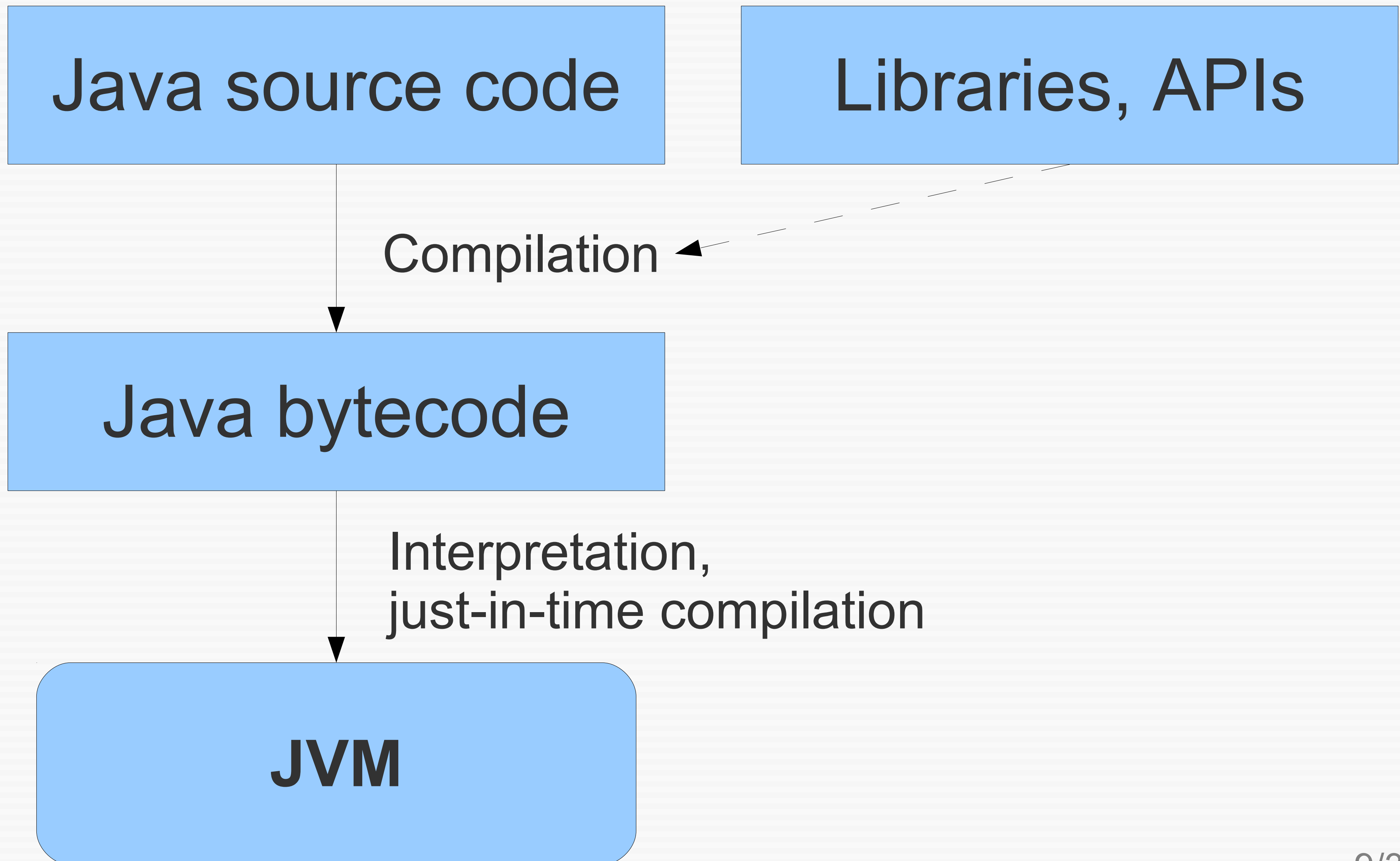
History of RTSJ

- Version 1: JSR 1
  (approved in Nov 2001)
- Version 1.1: JSR 282
  (still under development ...
  unclear if ever released)

RTSJ features/guarantees

- Real-time-specific thread types
- Real-time scheduler
- Extended memory management
- Inter-thread synchronisation/communication
- Event handling
- Clocks, timers
- **Uniform access to hardware + RTOS functionality**

Architecture of standard Java



RTSJ architecture

- RTSJ programs are **normal Java programs**, written using RTSJ library
- RTSJ implementations have two parts:
  - RTSJ **library**:
    set of real-time-specific classes
  - Tailor-made **JVM**:
    minimalist, real-time scheduler; often runs without further OS; can be partly implemented in hardware
- Sometimes: Java bytecode is upfront compiled to machine code

More information

- http://www.rtsj.org/
- Is Java ready for real-time?
  Jan Vitek at MVD'10,
  http://goedel.cs.uiowa.edu/MVD/talks/Vitek.pdf
- Peter C. Dibble, “Real-Time Java Platform Programming”

Examples

For playing: Timesys

- Provide free reference implementation of RTSJ
- Standard Java compiler + library + tailor-made JVM
- http://www.timesys.com/java/ (need to register for download)

Explicit feasibility analysis

- Task parameters are provided explicitly in code
    - Arrival times, WCET, deadlines
- RTSJ implementations can perform explicit feasibility/schedulability analysis
    - However: mostly not implemented

Synchronisation, communication

Monitors

- Main method for synchronisation in RTSJ → just like in normal Java
- Methods or blocks can be `synchronized`
    - Monitor region; only one thread can enter at a time
    - Used to implement mutual exclusion
    - Also possible: `synchronized(obj)`

Monitors (2)

- `wait`: temporarily give up monitor, wait until some other threads issue `notify`
  - Be careful: “spurious wake-ups” are possible
- `notify`: wake up some thread `wait`ing inside a monitor
- `notifyAll`: wake up all threads `wait`ing

MonitorControl

- Extension in RTSJ:
  It is possible to specify strategy for avoiding **priority inversion**
  - `PriorityInheritance`
  - `PriorityCeilingEmulation`
- Can be set globally, or on per-object basis

Queues

- `WaitFreeReadQueue`
    - reads are guaranteed not to block (mostly interesting for real-time threads); returns `null` if queue empty
    - multiple readers have to be synchronised by hand!
- `WaitFreeWriteQueue`
    - similar; writes are guaranteed not to block
- `WaitFreeDeque` (deprecated)

Asynchronous events

Events

- Encapsulation of internal or external happenings that need to be handled by program
  - E.g., timers, key presses, interrupts, etc.
- Captured by class `AsyncEvent`
  - `Timer` is a subclass of `AsyncEvent`
- Maintains a **queue** of events, to be handled successively

EventHandlers

- `AsyncEventHandler`: code to be executed when an event occurs
  - Very similar to (aperiodic) thread
  - Parameters like a thread: priority, release parameters, etc.
- Method `handleAsyncEvent` is invoked each time an event occurs
- Compare with callbacks or **(deferred) interrupt service routines** …

Memory management

(next lecture)

Conclusions so far

- Threads; scheduling; synchronisation; events
- RTSJ is a convenient, high-level API for designing real-time applications
- **Many of the features should be familiar to you!**
  (from FreeRTOS + micro-controller)
- More in next lecture +
  on home assignment