

Introduction to Symbian OS

Marco Aiello

VITALab --- TUWien (A)

&

DIT --- Universita' di Trento (I)

aiellom@ieee.org

Material based on original material from the author, from the <http://www.symbian.com> website, and from the book Symbian OS Explained by Jo Stichbury, Wiley (2005).



Symbian OS

- Introduction to Symbian OS
- Characteristics of Symbian v.7.0s
- Enhancements in Symbian v.9.2
- Leaves vs. Exceptions
- Event-drive multitasking using Active Objects
- Active Objects scheduler
- Threads and Processes





Symbian OS

- An operating system for mobile devices with limited resources, multitasking needs and soft real time requirements. Based on a modular, layered, micro-kernel approach.
- Requirements:
 - Resilient power-management
 - Careful use of memory resources
 - Multi-tasking (e.g, phone calls, messages, alarms, games, wap browser, camera, bluetooth app, etc.)
 - Soft Real-Time (e.g., incoming phone call)
 - Robust
- Symbian OS was designed for mobile devices, from its earliest incarnation as EPOC32 in the Psion Series 5.
- Symbian OS is a consortium formed in 1998 and owned by Psion, Nokia, Sony Ericsson, Panasonic (Matsushita), and Siemens.
- The shareholders are licensees of Symbian OS. In addition, because Symbian OS is an open platform, any manufacturer can license it.





Symbian OS on the following HW

Motorola A1000

FOMA F880iES

Nokia 3230

BenQ P30

FOMA F900iC

Nokia 7710

Sony Ericsson P910

FOMA F901iC

Arima U300

Nokia 7610

Panasonic X700

Lenovo P930

FOMA F900i

FOMA F900iT

FOMA F900iC

Nokia 7710

Sony Ericsson P910

FOMA F901iC

Lenovo P930

FOMA F900i

FOMA F900iT

Panasonic X800

kia 6600

FOMA F700i

Motorola A1010

Nokia N-Gage QD

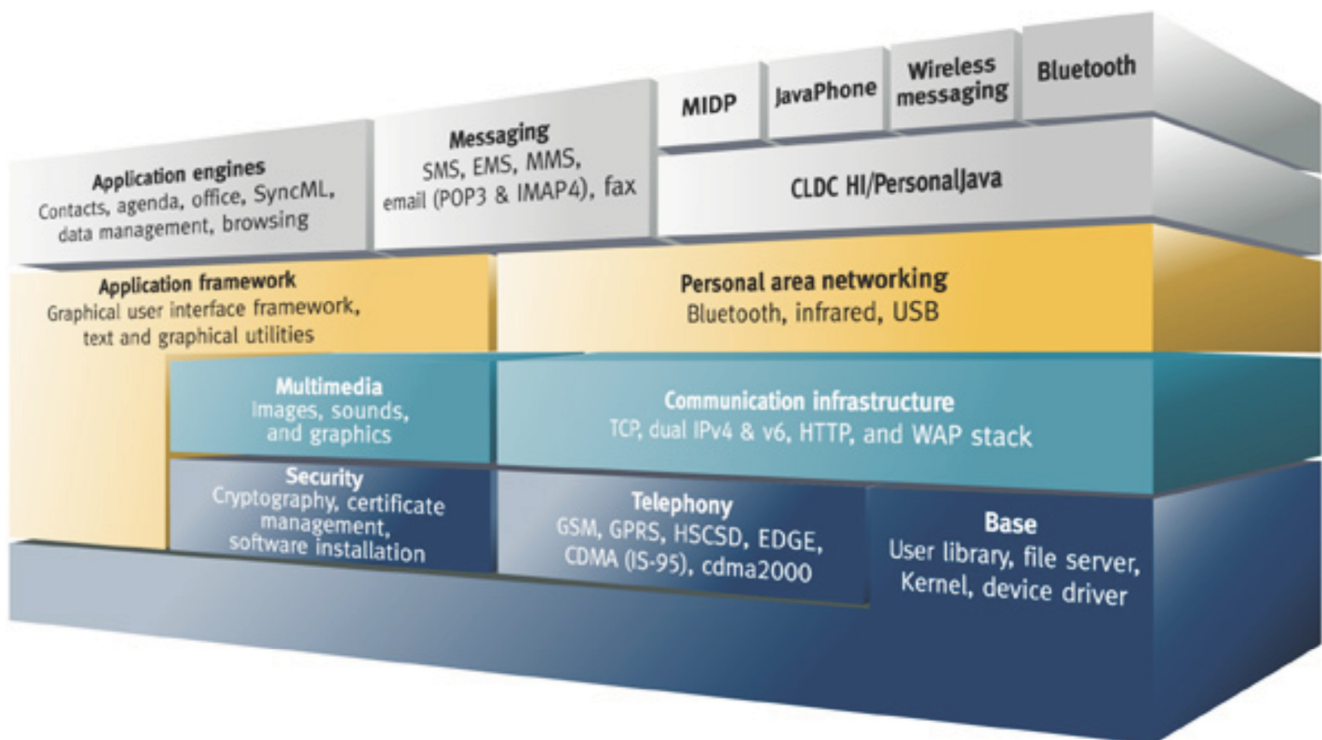
and many more...

From <http://www.symbian.com/phones/> April 2005





Layers in Symbian OS v 7.0s





Key features of Symbian OS v 7.0s

Rich suite of application engines

the suite includes engines for contacts, schedule, messaging, browsing, utility and system control; OBEX for exchanging objects such as appointments (using the industry standard vCalendar) and business cards (vCard); integrated APIs for data management, text, clipboard and graphics

Browsing

a WAP stack is provided with support for WAP 1.2.1 for mobile browsing

Messaging

multimedia messaging (MMS), enhanced messaging (EMS) and SMS; internet mail using POP3, IMAP4, SMTP and MHTML; attachments; fax

Multimedia

audio and video support for recording, playback and streaming; image conversion

Graphics

direct access to screen and keyboard for high performance; graphics accelerator API





Key features of Symbian OS v 7.0s

Communications protocols

wide-area networking stacks including TCP/IP (dual mode IPv4/v6) and WAP, personal area networking support include infrared (IrDA), Bluetooth and USB; support is also provided for multihoming capabilities and link layer Quality-of-Service (QoS) on GPRS/UMTS

Mobile telephony

Symbian OS v7.0s is ready for the 3G market with support for GSM circuit switched voice and data (CSD and EDGE ECSD) and packet-based data (GPRS and EDGE EGPRS); CDMA circuit switched voice, data and packet-based data (IS-95, cdma2000 1x, and WCDMA); SIM, RUIM and UICC Toolkit; Other standards can be implemented by licensees through extensible APIs of the telephony subsystem.

International support

conforms to the Unicode Standard version 3.0

Data synchronization

over-the-air (OTA) synchronization support using SyncML; PC-based synchronization over serial, Bluetooth, Infrared and USB; a PC Connectivity framework providing the ability to transfer files and synchronize PIM data.





Key features of Symbian OS v 7.0s

Security

full encryption and certificate management, secure protocols (HTTPS, WTLS and SSL and TLS), WIM framework and certificate-based application installation

Developing for Symbian OS

content development options include: C++, Java (J2ME) MIDP 2.0 and PersonalJava 1.1.1a (with JavaPhone 1.0 option), and WAP; tools are available for building C++ and Java applications and ROMs with support for on-target debugging

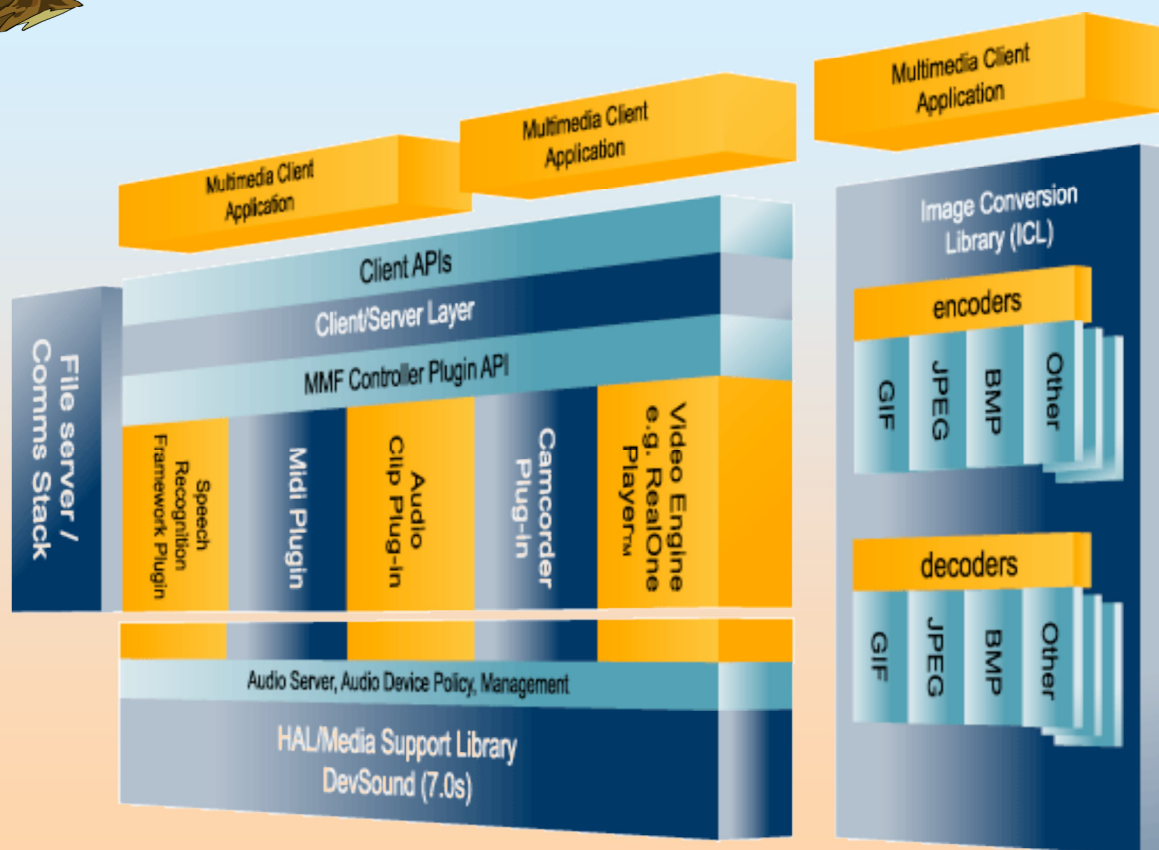
User Inputs

generic input mechanism supporting full keyboard, 0-9*# (numeric mobile phone keypad), voice, handwriting recognition and predictive text input 2.1





MultiMedia Framework





MultiMedia Framework

- The **Multimedia Framework (MMF)** provides a lightweight, multi-threaded framework for handling multimedia data. The framework provides audio recording and playback, audio streaming and image related functionality. Support is provided for video recording, playback and streaming.
- *The image conversion library* is a lightweight, optionally multithreaded, client-side framework for still image codecs and conversion; Plug-ins supplied by Symbian include JPEG, GIF, BMP, MBM, SMS, WBMP, PNG, TIFF, WMF and ICO.
- *Third party plug-ins* are enabled through the extensible nature of the framework.
- *Camera support:* An onboard camera API providing a consistent interface to basic camera functions.





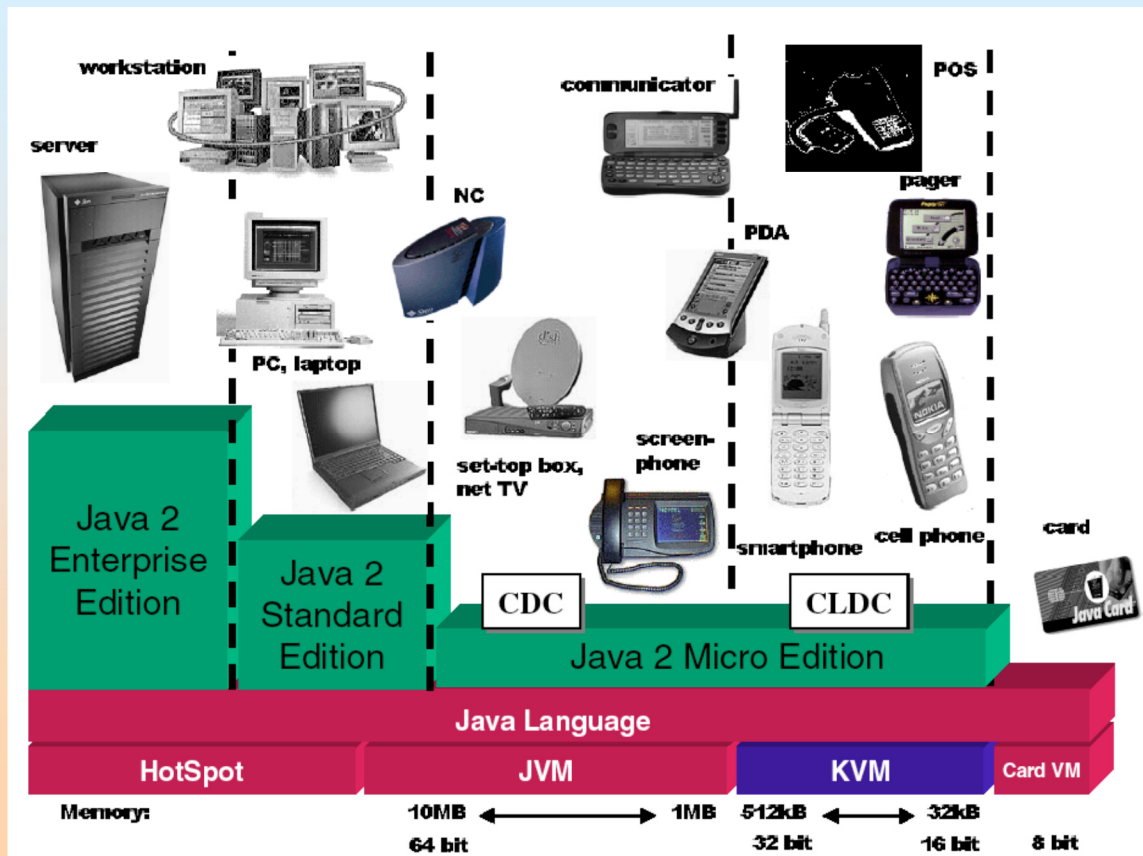
Java Micro Edition

Java can run also on small devices. But a full Java Virtual Machine is not implementable. Therefore, a subset of the Java Virtual Machine is implemented called Micro Edition.



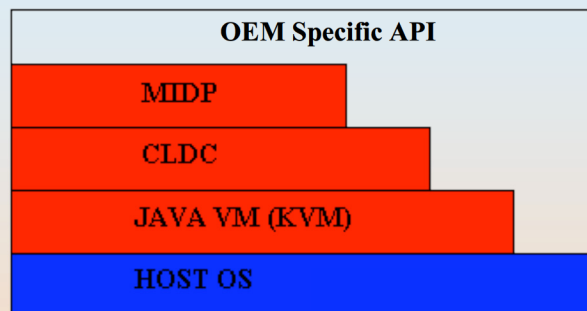


Java flavors





J2ME Architecture

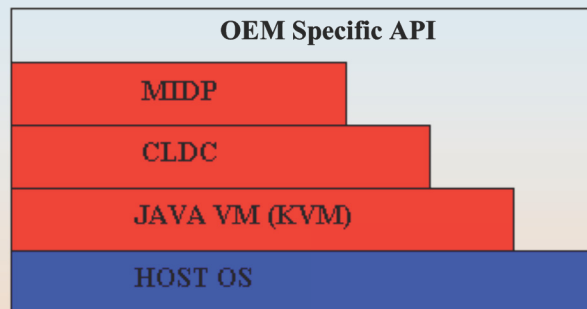


- KVM - Kilobyte Virtual Machine
 - 40 – 80 KB in size
 - For devices with 160 KB of memory and 16 or 32-bit RISC/CISC microprocessors





J2ME Architecture (cont.)

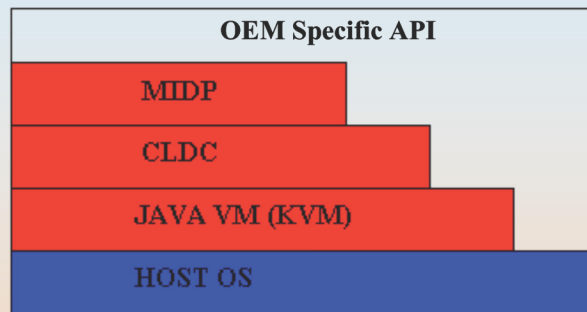


- CLDC - Connected Limited Device Configuration
 - Provides core lower level functionality
 - Bare minimum IO and utilities
 - Currently consists of java.io, java.lang, java.util, java.microedition.io





J2ME Architecture (cont.)



- MIDP – Mobile Information Device Profile
 - MIDP provides the core application functionality for mobile devices
 - ▶ Network connectivity, data storage, and user interfaces





J2ME and MIDP

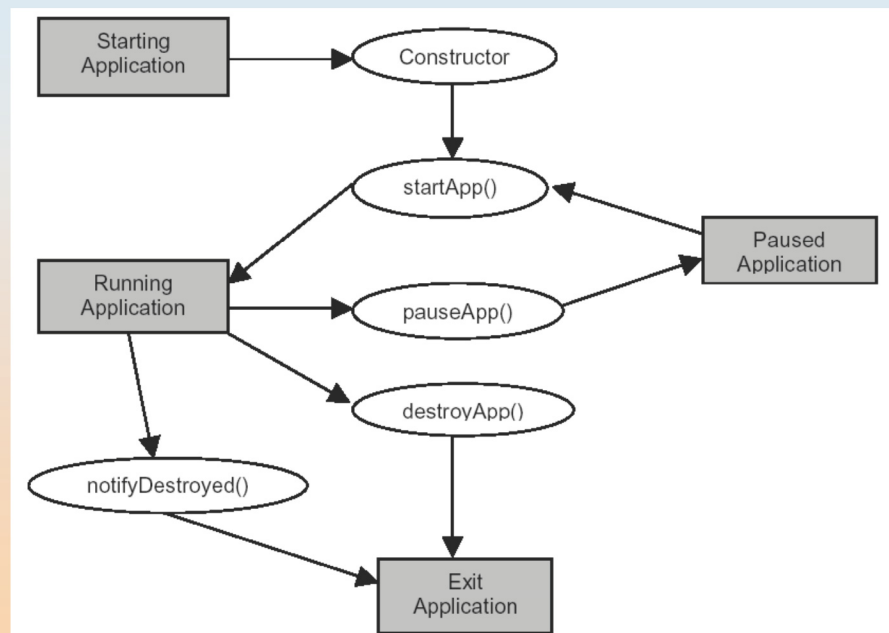
- The Mobile Information Device Profile (MIDP) is a key element of the Java 2 Platform, Mobile Edition (J2ME). When combined with the Connected Limited Device Configuration (CLDC), MIDP provides a standard Java runtime environment.
- The MIDP specification defines a platform for dynamically and securely deploying optimized, graphical, networked applications.
- CLDC and MIDP provide the core application functionality required by mobile applications, in the form of a standardized Java runtime environment and a rich set of Java APIs.





J2ME Applications

- A J2ME app is called a **Midlet**



Midlet life cycle





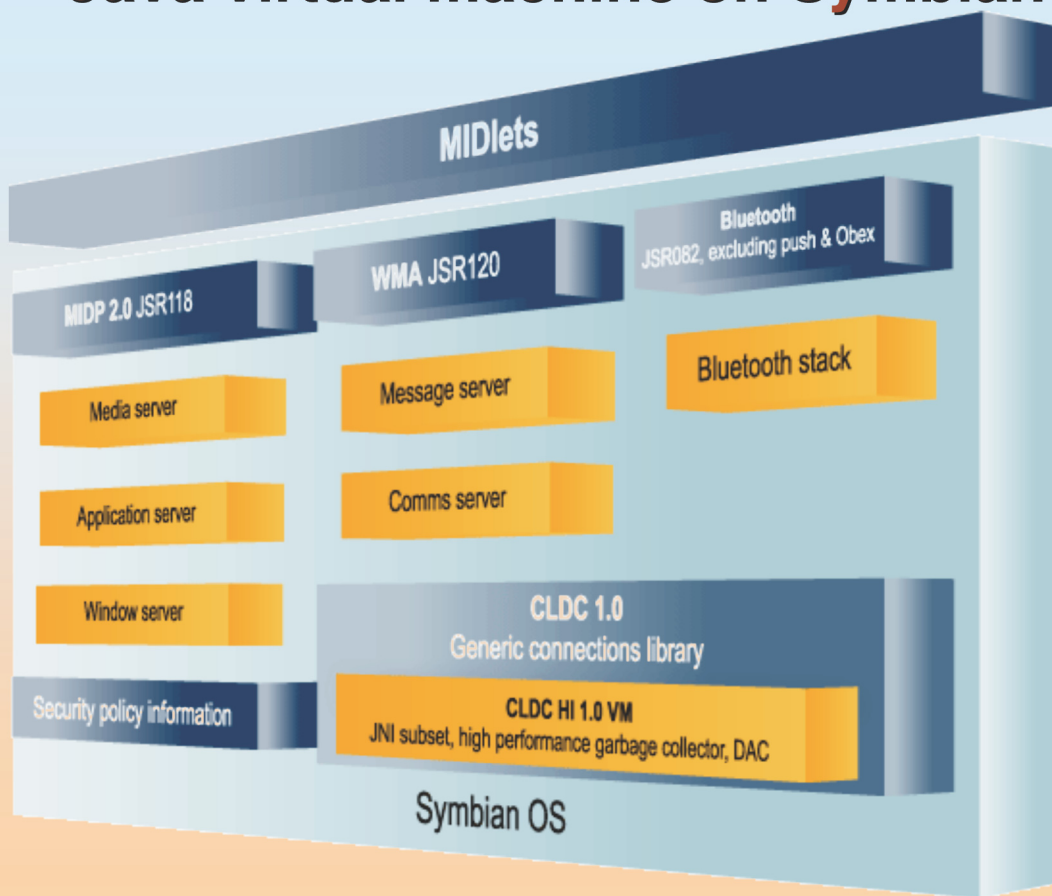
Java Micro Edition

- Minimal memory requirements:
 - ▶ 128 kilobytes of non-volatile¹ memory is available for the Java virtual machine and CLDC libraries
 - ▶ at least 32 kilobytes of volatile² memory is available for the Java runtime and object memory
- Main differences with ordinary Java and JVM:
 - ▶ No floating point support (since floating point HW is often missing)
 - ▶ No finalization (`Object.finalize()`)
 - ▶ Smaller set of exception types
 - ▶ Java Native Interface (JNI): is platform dependent
 - ▶ No support for user defined class loaders
 - ▶ No reflection
 - ▶ No support for Thread groups and daemon thread
 - ▶ No support for weak references
 - ▶ More lightweight class verifier





Java virtual machine on Symbian





Java on Symbian OS

- The Symbian OS v7.0s implementation provides MIDP 2.0, CLDC 1.0 with Sun's CLDC HI Java VM, Bluetooth 1.0, and Wireless Messaging 1.0. It also includes PersonalJava with the JavaPhone APIs.
- J2ME CLDC/MIDP2.0 provides a fast Java environment, has a tight footprint, and provides a framework that scales smoothly from constrained smartphones to more capable ones.
 - ▶ supports the OTA recommended practice document for MIDlet installation mandated by the MIDP 2.0 specification
 - ▶ heap size, code size, and persistent storage are unconstrained, allowing larger, more compelling MIDlets to be run
 - ▶ MIDlets look and behave very much as native applications: they use the native application installer and launcher, and native UI components
 - ▶ supports native color depth (4096 colors)
 - ▶ Generic Connection Framework implementation including sockets, server sockets, datagram sockets, secure sockets , HTTP and HTTPS
 - ▶ provides debugging support
 - ▶ implements Bluetooth (excluding push and OBEX)
 - ▶ implements Wireless messaging





Personal Java on Symbian OS

- PersonalJava on Symbian OS implements the 1.1.1a PersonalJava Application Environment specification.
 - ▶ support for Unicode across the Host Porting Interface
 - ▶ support for ARM JTEK (Jazelle) technology for Java hardware acceleration and ARM VTK (VMA technology kit) for Java software acceleration
 - ▶ JVMDI for remote debugging (TCP/IP over the serial link)
 - ▶ Symbian OS SDK for Java tools. Runtime customization tools





JavaPhone on Symbian OS

- The JavaPhone component delivers a set of APIs that extend the PersonalJava runtime to access important native functionality including telephony, agenda, contacts, messaging and power monitoring. Symbian OS provides the JavaPhone 1.0 reference implementation.
 - ▶ JavaPhone APIs: address book (based on the contacts application engine), calendar (based on the agenda application engine), user profile, network datagram, and power monitor (minimal implementation).
 - ▶ optional PersonalJava interfaces: serial communications, secure socket communications (HTTPS is supported, javax.net.ssl is not implemented)
 - ▶ Java Telephony API: JTAPI Core package
 - ▶ Java Telephony API (mobile): Java Mobile API Core interfaces, MobileProvider, MobileProviderEvent, MobileProviderListener, MobileAddress, MobileTerminal, MobileNetwork, MobileRadio, MobileRadioEvent and MobileRadioListener





Enhancements in Symbian v.9.2

■ Java support

- ▶ latest Java standards: including MIDP 2.0, CLDC 1.1, JTWI (JSR185), Mobile Media API (JSR135), Java API for Bluetooth (JSR082), Wireless Messaging (JSR120), Mobile 3D Graphics API (JSR184) and Personal Information Management and FileGCF APIs (JSR075)

■ Hardware support

- ▶ supports latest CPU architectures, peripherals and internal and external memory types

■ Graphics

- ▶ direct access to screen and keyboard for high performance; graphics accelerator API; increased UI flexibility (support for multiple simultaneous display, multiple display sizes and multiple display orientation)





Enhancements in Symbian v.9.2

■ Platform security

- ▶ proactive system defence mechanism based on granting and monitoring application capabilities through Symbian Signed certification. Infrastructure to allow applications to have private protected data stores.
- ▶ In addition, full encryption and certificate management, secure protocols (HTTPS, SSL and TLS) and WIM framework

■ Communications protocols

- ▶ wide area networking stacks including TCP/IP (dual mode IPv4/v6) and WAP 2.0 (Connectionless WSP and WAP Push), personal area networking support including infrared (IrDA), Bluetooth and USB; support is also provided for multihoming and link layer Quality-of-Service (QoS) on GPRS and UMTS networks

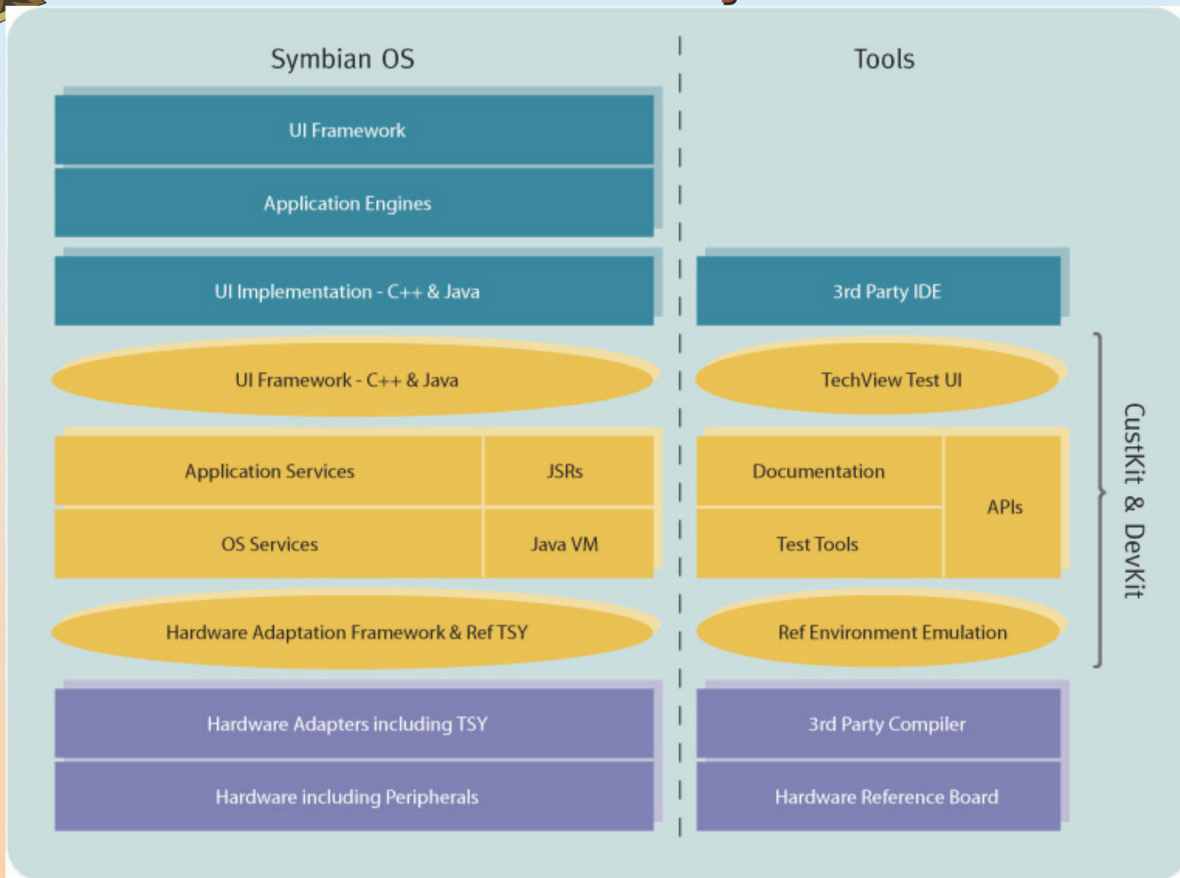
■ Developing for Symbian OS

- ▶ native system and application development in C++, supported by CodeWarrior and (from 2005H2) Eclipse-based IDEs. Java MIDP 2.0 supported by all mainstream Java tools. PC-hosted emulator for general development. Reference boards for general development, 2G, 2.5G and 3G telephony, supported by a range of JTAG probes and OEM debuggers





Enhancements in Symbian v.9.0



C++ basics in Symbian



Fundamental types

■ Fundamental built-in types:

- `TIntX` and `TUIntX` ($X=8,16,32,64$) signed and unsigned integers of increasing bit size
- `TReal32` and `TReal` ($=TReal64$) for single and double precision float (operations are slower than int operations)
- `TText8` and `TText16` for characters, corresponding to integers
- `TAny*` is a pointer to any object
- `TBool` for booleans (`ETrue=1` and `EFalse=0` integers, any non zero value is considered true!)

The above `typedef`ed types are guaranteed to be compiler-independent.





Class name conventions

- **T Classes** (T is for type as they are similar to fundamental types)
 - must not have a destructor (removed when the function is removed from the *stack*)
 - can also be created on the heap, but should be moved to the stack before potentially leaving code is executed
 - can exist without constructors, with constructors this can't be done:

```
TMyPODClass local = {2003, 2004, 2005}
```





Class name conventions

■ C Classes (classes derived from CBase)

- CBase has a virtual destructor, so C Classes can be deleted through a CBase pointer (but if TAny this is not invoked!)
- The `new` operator initializes to zero the new objects
- Objects of C class must always be allocated on the *heap*





Class name conventions

■ R Classes (representing Resource handles)

- It doesn't derive, so constructor and destrucotr must be defined
- Usually small classes invoking `open()`, `create()`, `initialize()` in the constructor and `close()`, `resent()` on destruction
- Closing a stream is never done automatically, so care must be taken in writing the destructor





Class name conventions

■ M Classes

- *Mix-ins* classes used to add “flavor” to a base class
- Abstract interface class equivalent of Java *interfaces*
- Use: inherited by a base class. Multiple inheritance is possible
- Structure:
 - No member data
 - No constructor (and no destructor, of course)



Leaves and Exceptions



Leaves: Symbian lightweight exceptions

- Exceptions increase code robustness, but also increase the length of the code and the run-time RAM overheads
- Try, catch, throw are not part of the C++ syntax for a Symbian compiler
- In Symbian there is a more lightweight alternative: Leaves





Leaves basics

- A leave may occur:
 - Calling a leaving function
 - Invoking a system call function to cause a leave
- When a leave occurs the corresponding error value is propagated through the call stack until it is caught
- Leaves are caught by a trap harness where the exception can be correctly handled
- From the implementation point of view, leaves are similar to the C *setjmp* and *longjmp* mechanisms





Leaves mechanisms and conventions

- A function can leave (leaving function) if
 1. It calls code that can leave without surrounding that code with a trap harness
 2. Calls system functions *User::Leave()* or *User::LeavelfError()*
 3. Uses operator *new (ELeave)*, overloaded operator that leaves when no memory available
- *The name of a leaving function must be suffixed with “L”*





Leaves and objects life cycle

- General rule: constructors and destructors should not leave
- Take the construction code

```
CExample *foo = new CExample();
```

- Allocates memory on the heap
- Invokes the constructor
- What if allocation goes fine, but the constructor leaves? Memory leak!

- Solution: two-phase construction





Two-phase construction

- If need to perform leaving code in the initialization or destruction phase, use the-phase idiom:
 - Provide non-leaving constructor: `CExample()`
 - Isolate leaving code in a function and expose that function to the clients: `ConstructL()`
 - Require clients to first call the leaving function and, if needed, handle leaves, or
 - Wrap both phases in a static function that take care of everything: `NewL()` or `NewLC()`





Trapping leaves: mechanisms

- Two macros allows to trap leaves:
 - TRAP
 - TRAPD (allows to store error code in a local variable)





Trapping leaves: under the hood

- Entry and exit of a TRAP macro result in a kernel executive call (system call):
 - `TTrap::Trap()`
 - `TTrap::UnTrap()`
- A struct is allocated at runtime to hold the current content of the stack and allow unwinding in case of a leave
- Therefore, trapping leaves may be expensive: try to minimize trap calls (e.g., if you have a series of calls to leaving functions, don't trap each of the calls, but combine them into one call and one trap)



The Cleanup Stack





Memory management and the cleanup stack

- Interactions of leaves management and memory management can be error-prone
 - If a local variable points to an object in the heap and a leave occurs, the stack is unwinded without calling the object destructor: memory leak!
- (Non optimal) solution: trap all leaving code to execute cleanup operations
- Need for a pattern that allows to consistently and easily handle memory when leaves can occur: the *cleanup stack*





Cleanup stack

- Objects that are not leave-safe should be placed on the cleanup stack before calling leaving code
- In case of an event, the cleanup stack manages the deallocation of the objects it contains
- The cleanup stack knows the type of objects it is dealing with and takes the appropriate actions (calls the appropriate cleanup functions/destructors)





Cleanup stack: API

- Push object on the stack:

```
static void PushL(TAny* aPtr);  
static void PushL(CBase* aPtr);  
static void PushL(TCleanupItem anItem);
```

- Pop objects from the stack

```
static void Pop();  
static void Pop(TInt aCount);
```

- Pop and destroy objects

```
static void PopAndDestroy();  
static void PopAndDestroy(TInt aCount);
```





How objects are cleaned by the cleanup stack

- If the item on the stack is a `CBase*` pointer, the object is destroyed with `delete`
- If the item on the stack is a `TAny*` pointer, the memory occupied by the object is freed with `User::Free()`, a rather limited form of cleanup — not even the C++ destructor is called
- If the item on the stack is a cleanup item, i.e., an object of type `TCleanupItem`, the cleanup operation defined during construction of the `TCleanupItem` object is invoked





PushL: What if it leaves?

- PushL is a leaving function
- So a leave may happen when pushing an object on the cleanup stack. In that case, does a memory leak occur?
- No! The cleanup stack has always store to save a pushed object. If, after storing an object, there are no more free spots, it tries to allocate more memory. If the memory allocation fails, a leave is thrown. But our objects have all been safely stored on the stack and can be cleaned up. Therefore, they are not leaked



Event-driven multitasking using Active Objects





Costs of Context Switching

- Process context switching represents a substantial *cost* to the system in terms of CPU time and can, in fact, be the most costly operation on an operating system
- Thread context switching reduces the costs by requiring that less run-time information is switched, but it is still a costly operation
- In Symbian there are processes, threads, but also active-objects within threads which have their own scheduling mechanism
- In Symbian OS, threads are scheduled pre-emptively by the kernel, which runs the highest priority thread eligible
 - Threads can be suspended, waiting for an event
 - Time-sliced thread scheduling





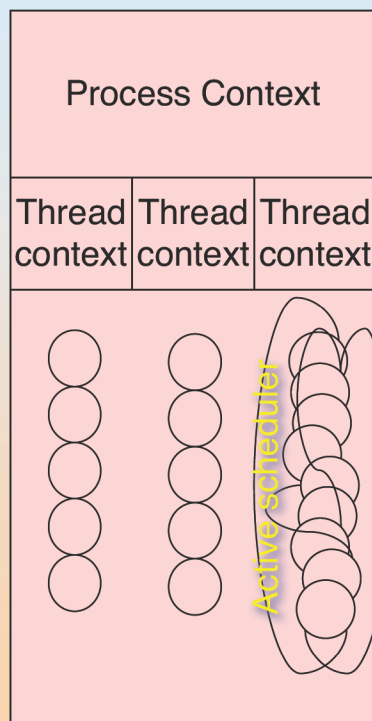
Active-Objects

- Efficient event-handling model with low response times and appropriate ordering
- Especially important for user-driven events
- Active-Object context switch can be 10 times faster than a thread context switch
- Space overhead may be few hundred bytes vs. 4-8KB for threads
- In practice, a Symbian application will be a main event-handling thread having a set of active-objects each representing a task
- Active-objects cannot be preempted, therefore should not be used for real-time tasks!





Active Objects





Cactive

- An active object class must derive from the CActive class

```
Class CActive : public CBase
{
public:
    enum TPriority
    {
        EPriorityIdle=-100,
        EPriorityLow=-20,
        EPriorityStandard=0,
        EPriorityUserInput=10,
        EPriorityHigh=20
    };

public:
    IMPORT_C ~CActive();
    IMPORT_C void Cancel();
    IMPORT_C void SetPriority(TInt aPriority);
    ...

protected:
    IMPORT_C CActive(TInt aPriority);
    IMPORT_C void SetActive();
    virtual void RunL() =0;
    virtual void DoCancel()=0;
    IMPORT_C virtual TInt RunError(TInt aError);
    ...
};
```





Life-cycle

- On construction the priority is set (not for preemption, but for scheduling)
- Active-Objects typically 'own' objects to which they issue requests asynchronously
- Each Active-Object has at most one outstanding request per owned object
 - panic: if a request is made twice due to programming error
 - refuse: if the multiple request is legal
 - cancel the outstanding request and deliver the new one
- RunL () must be implemented to handle the event (first method called when object is scheduled to handle an event)
- Cancel () calls DoCancel () (which must be implemented) which calls the cancellation of the owned object
- RunError() called if a leave occurs in the RunL()
- Destruction of an active objects by invoking Cancel()





Active Object

(1) Issues request
passing `iStatus`

(3) Calls `SetActive()`
on itself

(5) Active Scheduler calls
`RunL()` on the active
object to handle the event

`RunL()` resubmits
another request or stops
the Active Scheduler

`RunL()` cannot be pre-
empted

Asynchronous Service Provider

(2) Sets `iStatus=KRequestPending`
and starts the service



(4) Service completes.
Service provider uses
`RequestComplete()` to
notify the Active Scheduler
and post a completion result

PROCESS OR
THREAD
BOUNDARY





Threads and Active Schedulers

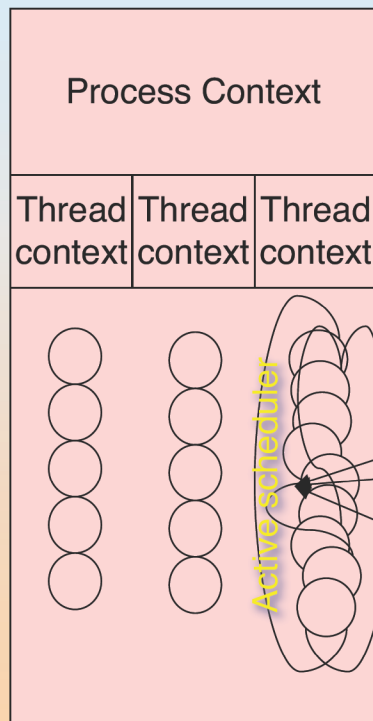
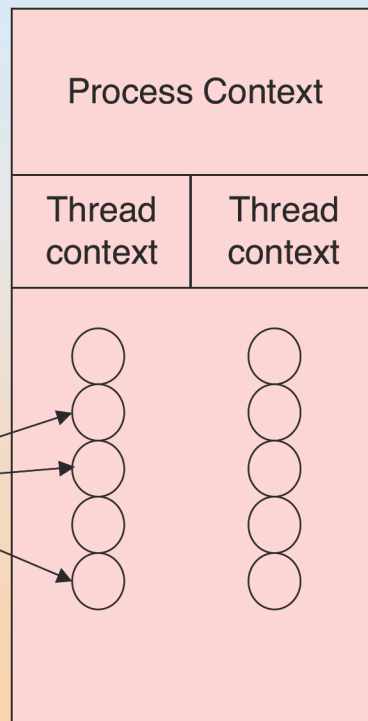
- Most threads have an active scheduler created implicitly
- But user defined threads don't and creating and starting one may be necessary
- Some threads do not have active schedulers intentionally:
 - The Java implementation
 - The C standard Library thread
 - OPL





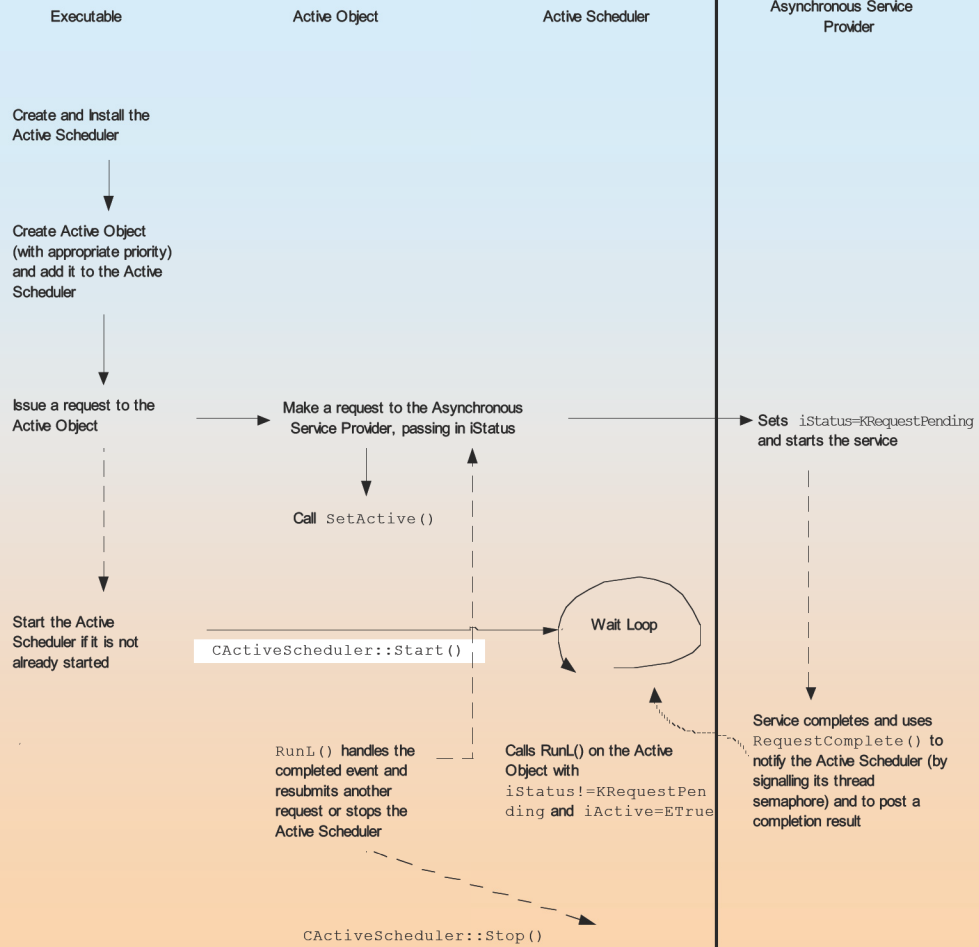
Event Handling Roles

Process with
asynchronous
multi-tasking
event
handling



Asynchronous
Service
Provider







Responsibilities of an Active Object

- Sets its own *priority*. Usually set to standard at 0. Can change dynamically
- At least one method to initiate a request then passing a `iStatus` object
- After submitting a request call `SetActive()` on itself setting the `iActive` flag (used by the active scheduler)
- Have only one outstanding request per time
- Pass the `iStatus` to the service provider, also used by the active scheduler to determine what to do with the active object
- Must implement the virtual methods `RunL()`, `DoCancel()`
- Not be destroyed while is waiting for the completion of a request. By calling `Cancel()` if this is not the case the thread panics
- Objects passed to the provider must live at least as long as the completion of the request
- If a leave can occur in `RunL()`, the class should override the default `RunError()`





Responsibilities of an Asynchronous Service Provider

- Set the `KRequestPending` to `TRequestStatus` (i.e., `iStatus`) to indicate to the scheduler that the request is ongoing
- Upon completion set `KRequestPending` to a result code and then call `RequestComplete()`
- `RequestComplete()` must be called only once for each request, otherwise there will be a panic in the thread with the requesting active object
- Must provide a cancellation method for each asynchronous request, which acts immediately and sets `TRequestStatus.KErrCancel`





Responsibilities of an Active Scheduler

- Suspend the thread calling `User::WaitForAnyRequest()`
When an event is generated it resumes the thread and searches for the active object that handles it
- Ensure that each request is handled once and only once. It resets the `iActive` flag of an active object before calling `RunL()`
- Place a TRAP harness around `RunL()` from which it calls the active objects `RunError()` if it leaves
- Raise a panic if it can't find an object for an event, i.e., one with `iActive` set and the `TRegeustStatus` indicating it has completed





Active Scheduler

- The active scheduler's wait loop is started by a call to `CActiveScheduler::Start()` most often done automatically
- Otherwise done by

```
CActiveScheduler* scheduler = new(Eleave) CActiveScheduler;  
CleanupStack::PushL(scheduler);  
CActiveScheduler::Install(Scheduler);  
scheduler.start();
```

- When the first active objects makes a request, `User::WaitForAnyRequest()` is called
- The scheduler is stopped with

```
scheduler.stop();
```





Customizing The Active Scheduler

- The active scheduler can be customized to the program's need
- This is achieved by subclassing the class `CActiveScheduler` and overwriting the `Error()` and `WaitForAnyRequest()` methods



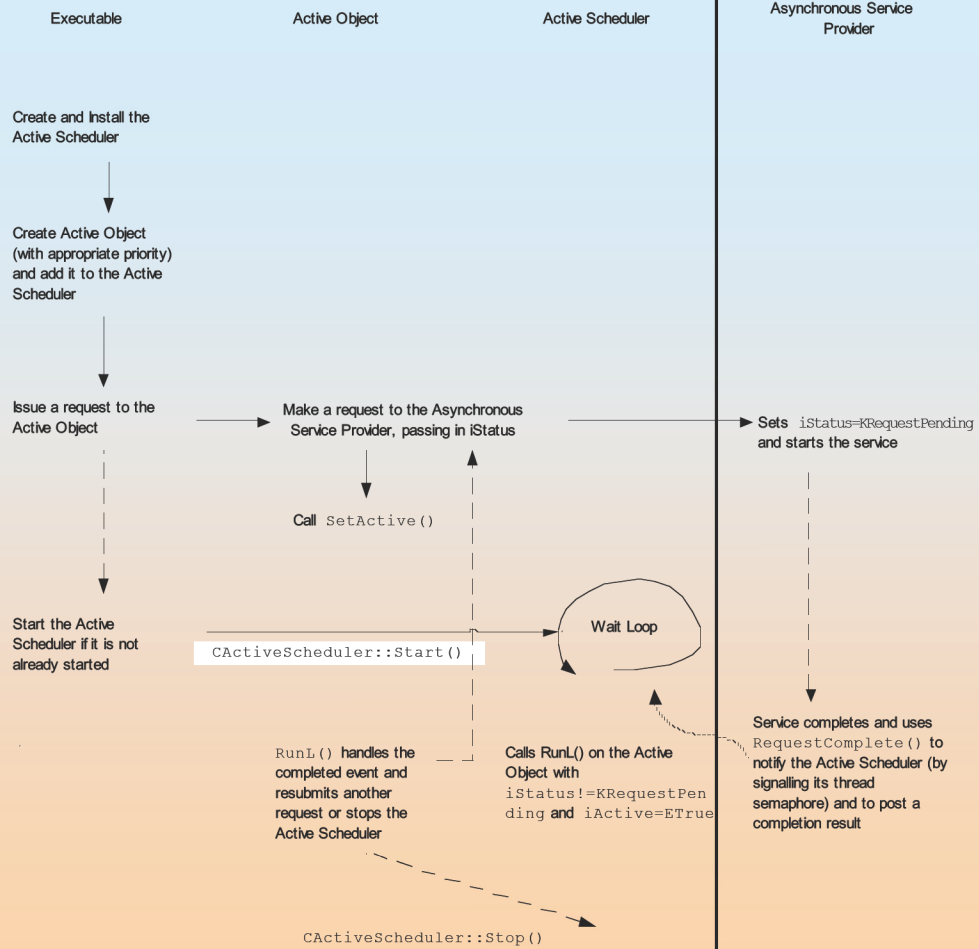


Cancellation

- Every request to an active object must complete exactly once
- If `CActive::Cancel()` is called, then determine if there is any outstanding request, if yes `DoCancel()` is called
- The service provider must provide request a cancellation method
- `DoCancel()` should be fast because `Cancel()` is a synchronous operation
- `Cancel()` resets the `iActive` flag

When an active object request is cancelled by a call to `Cancel()`, the `RunL()` event handler does not run. This means that any post cancellation cleanup must be performed in `DoCancel()` rather than in `RunL()`





Processes and Threads





Processes and Threads

- Active Objects live within threads which are scheduled with a priority based preemptive scheuling policy
- Data of threads within the same process is shared and thus must be protected with the usual synchronization primiteves such as semaphores
- Threads are manipulated via the `RThread` class
- Typical operations are
 - Suspend
 - Resume
 - Panic
 - Kill or Terminate
- Each thread has its own independent heap and stack
- The size of the stack is limited upon creation `RThread::Create()`
- Each thread has a unique identifier, returned by `RThread::Id()`
- Threads can be handled by `RThread::Open(identifer)`





Thread Priorities

- Threads are pre-emptively scheduled and the currently running thread is the highest priority thread ready to run
- If there are more thread in the above situation, then they are time-sliced on a round-robin basis
- Is starvation possible?
- Priority can be dynamically changed `RThread::SetPriority()`
- A thread has a relative and an absolute priority
- The absolute priority is either a function of the process priority, or independent from it
- All threads are created with `EPriorityNormal` by default and put into a suspended state, it runs when `RThread::Resume()` is called
- A thread is stopped with `Suspend()` or with `Kill()` and `Terminate()`. `Panic()` also stops a thread and gives a programming error
- It is possible to receive a notification when a thread dies





Panics

- When a thread is panicked the code in it stops running
- There is no recovery from a panic
- Unlike a leave, a panic can't be trapped
- A panic in a system thread will result in a reboot of the device!
- A panic in any other thread results in a "Program Closed" message box showing: the process name, panic category and an error code
- If one is working on an emulator one can break into the code causing the panic: *just-in-time debugging*
- One thread can panic another one:
 - Used by a server when a client passes a badly formed request
 - To prevent bugs, but also as self-defense (against denial-of-service attacks)





Error types

■ Faults

Raised if a critical error occurs such that the OS cannot continue. It reboots the device. It can only occur in system's code. Also known as system panic

■ Leaves

Occur under exceptional conditions, such as out of memory, or communication error. It should always be trapped by a function on the calling stack. If this is not the case it will generate a panic

■ Panics

An exception in the thread that causes the halting of the execution. It cannot be trapped and may cause the device's reboot.





Processes

- Processes are handled as threads, with one main difference, there is no suspend method
- RProcess
- Processes are not scheduled! Threads are scheduled across process boundaries. Process are used to protect memory space, rather as units of multi-tasking.
- The kernel server process is a special process that contains two thread which run at supervisor privilege level:
 - The kernel server thread: the first thread to run, the highest priority of all threads
 - The Null thread: lowest priority of all threads, used to swith the processor into idle mode to save power, by calling a callback function to shut down part of the hardware. To allow for this thread to run, threads performing input/output should wait on events rather than polling





Inter-Thread Data Transfer

- Pointer's can't be transferred between threads running on different processes. Why?
- The data transfer can be done with a dedicated API called `RMessagePtr`
- `RThread::WriteL( )` on the thread's handle
- `RThread::ReadL( )` to read data on the running thread



References



The slides are based on the following material:

- Symbian OS Explained by Jo Stichbury, Wiley (2005)
- Symbian official website: <http://www.symbian.com>:
 - Symbian OS Version 7.0s, Functional Description; By Kevin Dixon, (June 2003)
- Java official website: <http://java.sun.com>

