**Introduction to Java**

**A First Look**

Java is a second or third generation object language

Integrates many of best features

Smalltalk

C++

Like Smalltalk

Everything is an object

Interpreted or just in time compiled

Windowed support built in

Weak and not to be confused with MS windows

Graphics support built in

Implements a garbage collector

Like C++

Must more portable than Smalltalk

Does not implement a destructor

Implements a `finalize()` method to clean up as necessary

For security reasons

Does not use obvious pointers

They’re there

Every variable needs an address

For those familiar with C or C++

Syntax, variable types, flow of control constructs

Will be similar

Like C++ or Smalltalk – or most other OO languages

Supports

Inheritance

Through inheritance mechanisms

Polymorphism and multiple inheritance

Through extended interfaces

Taken from the Smalltalk MVC

Does not support MI per se

If you know C++

Learning Java will be pretty simple

Familiar notion of Java is through the web
Yep – that’s there
Many other things as well

**A Closer Look**

**Variables**
The language is strongly typed
Java supports all the obvious
Integers
  - Byte, short, long
Floating point
  - Float, double
Characters
  - Char
Boolean
  - boolean
class, this
super
  - from Smalltalk
Shared variables / methods
  - Static
  - Adds final
    Such variables are constant
    Note they don’t occupy memory space

**Operators**
Four basic functions
Short cut operators
Auto increment and decrement
Bitwise operations
  - Java initially intended for embedded systems applications
  - And, or, exclusive or
  - Shifts left and right
  - Short cut operators

**Flow of Control**
Again Java supports all the obvious
  - if – else, do-while, for, return, switch / case, etc.
Visibility
Java supports the expected access protection qualifiers
public, protected, private
These have the same interpretation as in C++

Inheritance
Through
implements, interface, extends
Adds final
No further derivations

Exceptions
From C++
catch, try, throw(s)

Java Classes
As one might expect
Class declaration preceded by keyword class
Defines a new data type
Following declaration and definition
Can create objects of that type

General form appears exactly as in C++

```java
class className
{
    constructors
data members
    function members
}
```

Unlike C++
Declaration and definition
Appear in same file
Members are public by default
Access specified
Member by member basis
Rather than as ‘all members following…’
Member and method access
Use familiar dot notation

Example
MyClass anInstance;
anInstance.myData;

Let’s look at a simple Java program
Example
SimpleClass0.Java

Java Inheritance
Java supports single inheritance
Does not support notions of
  Private or protected inheritance

Syntax
class Y extends class X
{
    ...
}
Should look familiar to C++ programmers
As with C++
  Method declared in subclass
    With same name as one in parent class
    Overrides the parent method

Example
inherit0.java

Now we’ll execute a member function polymorphically
  Referred to in Java as dynamic method dispatch
Example
inherit1.java
**Java Interfaces**

C++ used the notion of

Virtual functions and pure virtual functions

With pure virtual functions

One could implement an abstract class

Also called an

Interface class

Abstract base class

Abstract super class

Abstract super class

Such a class could not have instances

Defined the interface

Subsequently derived classes must support

Java uses notion of interfaces to

Abstract a class’ interface from its implementation

Permits one to specify

What a class must do

Not how it should do it

Also similar to adapters in the C++ STL

Interfaces designed to support dynamic method dispatch

At runtime

This is late or dynamic binding

Similar to classes

But in a completely different hierarchy

Class can extend only one class

Can implement many different interfaces

Loosely analogous to Smalltalk’s

View – controller pair

**Syntax**

```java
access interface interfaceName
{
    data members = aValue
    function members
}
```
When variables are declared in an interface
They are implicitly final and static
    Cannot be changed by implementing class
    They must be initialized with a constant value

Let’s look at a couple of examples

Example
interface0.java

Like class inheritance
    One interface can extend another
Example
Interface1.java

Java and Multithreaded Programming

One of the cooler features of Java
    Built-in support for multithreaded programming
Such a program
    Contains two or more processes
        That can run concurrently
    Each part is called a thread
    Multithreading is specialized form of multitasking
Before proceeding let’s build a little background

Multitasking - Multiprocessing

    Multitasking system as name implies
        Supports multiple tasks running simultaneously
    Important job in multitasking system
        Exchanging data between tasks
        Synchronizing tasks
        Sharing resources
    Let’s define and clarify many of the terms we’ll be using

Some Terms
    Task
        Names process and task used interchangeably
        As we noted process comprised of two things
Program
Sometimes called \textit{text section}

State
Program counter
Registers
Stack
Sometimes called \textit{data section}

We may have several processes associated with same program
Each has separate execution sequence
Text sections may be equivalent
Data sections will be different

Process may spawn other processes during execution
Called child processes
Spawning process called parent

Child process may get resources from
Operating system
Parent process

Task or process is program that is executing
Multitasking operating system
Permits one to run two or more programs simultaneously

Thread
Also called a \textit{lightweight} process
In contrast to task which is called a \textit{heavyweight} process
Exists within a process
Smallest unit of dispatchable code

Thread
Has its own
Program counter - thread of control
Register set
Stack space

Shares
Code section
Data section
System resources
Multitask

Multiple jobs in memory
Tasks appear to be executing simultaneously
Appear used since unless multiprocessing system
Only single processor
Only single task can really be executing
Context switch sufficiently quick
Gives illusion of simultaneous execution
Each task is typically operating
Independently of others
In has independent
Address space
Program counter
Stack pointer
Context switch expensive
Switching from one task to another

Multithread

Multiple threads within a single process
Multiple threads of control associated with
Several shared resources
Require fewer resources than multitasking processes

Java Threads

Thread States

Threads in Java may be in any of several states
Ready to run – waiting for the CPU
Running – has the CPU
Suspended – temporarily relinquished the CPU
Resumed – reacquired the CPU
Terminated – finished execution

Synchronization

Threads introduce asynchronous behaviour into a system
Must be means to enforce synchronicity when necessary
Java uses what is called a monitor
Monitor
Concept developed by C.A.R. Hoare
Control mechanism that ensures
Once thread enters monitor
Protected region
All other threads must wait (to enter) until it exits
Java uses notion of synchronized methods
Once thread has entered synchronized method
No other thread can invoke any other synchronized method
On same object

Using Java Threads
We use Java threads by
Inheritance
Extending thread
Interface
Implementing runnable
When Java program starts
One thread begins running immediately
Called main thread
All child threads are spawned from the main thread
It must be the last to finish execution
We can use this thread as seen in next example
Observe we must obtain reference to the thread by calling method
currentThread()
is a public static member of Thread

Example
thread0.java

Let's look first at developing our own threads

Implementing Runnable
By implementing the runnable interface
This is the easiest way
We can construct a thread on any object
That implements runnable
To do so
Only need to implement the single method `run()`.
Which has the prototype:

```java
public void run()
```

The thread will run until `run()` returns.

Next:

- Need to instantiate object of type `Thread`.
- From within the class,
- The new thread will not start running
  - Until it's `start()` method called.

Let's bring this all together in a simple example.

---

**Example**

`Thread1.java`

---

**Extending Thread**

The alternate way to create a thread:

- Create class that extends `Thread`.
- Then create instance of that class.
- Class extending thread:
  - Must override `run()` method.

We'll modify previous example to extend `Thread`.

**Example**

`Thread2.java`

---

Why two different methods for creating child threads:

- Thread class defines several methods.
- Can be overridden.

Only one must be overridden:

- `run()`.

If interested in both implementation and interface:

- Want to subsequently derive.
- Extend thread.

Otherwise:

- If we want only interface.
- Implement runnable.
Managing Threads
Main thread must be last to finish
After starting other threads
Can implement long delay
  Ensure that all are finished
Not ideal solution

isAlive()
First alternative
  Test each to see if still running
Language provides isAlive( ) method
  Test if thread still active or running

Syntax

```
final boolean isAlive( )
```

Returns
True
  If thread still running
False
  Otherwise

join()
Second alternative
  Wait until thread terminates
Language provides join( ) method
  Basic functionality
    Waits until thread called upon finishes
    Waits until called thread joins calling thread

Syntax

```
final void join( ) throws InterruptedException
```

Alternate syntax permits specification of maximum
  Wait time
Here’s an illustrating example

**Example**
Thread3.Java

Thread Priorities
Not all tasks we do are of equal importance
Time critical task
More important than routine maintenance task
Java permits user to specify priority at which a thread runs
Priorities range from 1 to 10
MIN_PRIORITY to MAX_PRIORITY
Syntax is of the form

**Syntax**

```java
final void setPriority( int level );
```

One can obtain current priority using get priority method

**Syntax**

```java
final int getPriority();
```

Thread Synchronization
When we have a system with more than one thread executing
Each may affect or be affected by another process
Several threads
May directly share logical address space
Code and data
Threads or lightweight processes
Be allowed to share data only
Through files
Concurrent access to shared data
Can result in data inconsistency
Critical Sections

Consider following problem and pseudo code fragments

Exchanging messages through bounded buffer

Allow n items in buffer

<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>if not full</td>
<td>if item</td>
</tr>
<tr>
<td>add item</td>
<td>get item</td>
</tr>
<tr>
<td>increment count</td>
<td>decrement count</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>wait until space</td>
<td>wait for item</td>
</tr>
</tbody>
</table>

Problem

Value of count

Depends upon who accesses variable

May be any of 3 different values

Variable count is critical variable

Within producer or consumer thread

Denoted critical section

Critical section in general

Section of code in which process is changing common variables

File

Table

etc

While process in critical section

Want to prevent access by all other processes

Termed mutual exclusion

Abstractly may represent code as

```
repeat
  entry section
  critical section
  exit section
  remainder section
until forever
```
One solution to critical section problem
Also view as need for synchronization
Of actions of various threads
On shared variable
Called semaphore or monitor

Synchronized Methods
Java provides such tools as built-in part of language
All objects have built in monitor
To enter object's monitor
Call method that has been synchronized
Use keyword \textit{synchronized}

\textbf{Syntax}

\begin{verbatim}
synchronized returnType method( args );
\end{verbatim}

While thread inside synchronized method
All other threads that try to call that method
Or any other synchronized method on same class instance
Must wait
To relinquish control
Return from method

Let's look at simple example
\textit{Example}

\begin{verbatim}
syncThread0.java
\end{verbatim}

Synchronized Blocks
Such approach covers most of cases we typically encounter
Other times
Don't have access to object internals
Have library of code purchased from third party
Still may need to control access to specific methods
Solve by putting calls to methods
Inside synchronized block
**Syntax**

```java
synchronized ( object )
{
    statements to be synchronized
}
```

Object is reference to object being synchronized

**Example**

syncThread1.Java

Interthread Communication

When working with multithreaded environment
System has appearance of multiple actions happening simultaneously
In fact typically have only one CPU
Thus in reality only one action occurring
Objective to keep CPU constantly busy
When process or thread blocked waiting on event
CPU cannot be used by other processes

Spinlock

We now see main disadvantage of monitors as described
When wait encountered
Encountering thread blocked
Must loop continuously while waiting
Called *busy waiting*
Waiting processes waste CPU cycles while waiting
Other process could use productively
Such a semaphore or monitor called *spinlock*
Because process spins while waiting for lock
Similar to polling
Advantage of spinlock
No context switch
Can take long time
If lock expected to be held for short time
Spinlock useful
To overcome need for busy waiting
  Modify definition of semaphore operations
When process executes wait operation
  If semaphore TRUE
    Must wait
Rather than wait process can block itself
  Block operation places self in waiting queue
    Associated with semaphore
    Process state changed to waiting
    Control transferred to scheduler
Blocked process should be restarted
  Some other process executes signal operation
  Process
    Restarted
      By wakeup operation
      Places process in ready state
    Placed in ready queue
Such a scheme implemented in Java using following methods
  wait( )
  notify( )
  notifyAll( )
Such methods can only be called
  From within synchronized method
wait( )
  Tells calling thread
    Give up monitor
    Go to sleep until different thread
      Enters monitor and calls notify( )
notify( )
  Wakes up first thread
    That called wait( )
      On same object instance
notifyAll( )
  Wakes up all threads
    That called wait( )
      On same object instance
Syntax

```java
final void wait() throws InterruptedException
final void notify()
final void notifyAll()
```

Let’s first look at an implementation

Producer and consumer

- Share a common one word buffer
- Are running asynchronously

Observe either consumer or producer can overrun the buffer

**Example**

`syncThread2.Java`

Now let’s look at an implementation

Same producer and consumer are synchronized
- Through the wait / notify pair

**Example**

`syncThread3.Java`