FEATURES OF THE JAVA PROGRAMMING LANGUAGE FOR REAL-TIME DISTRIBUTED APPLICATIONS

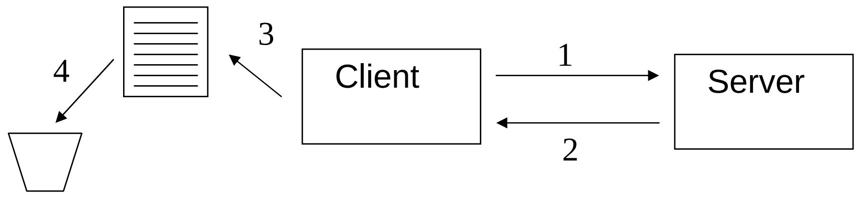
Some comparisons of the Java language compared to Ada and C++

- Java supports *single inheritance* for object classes (but multiple inheritance for interfaces) like Ada, which is more restrictive than C++.
- Java supports *packages* like Ada to collect related classes together.
- Unlike C++ and Ada, Java provides automatic garbage collection.
- Like Ada, Java provides strong data typing, array-bounds checking, exceptions, but not pointers.
- Like Ada, but unlike C++, Java supports multi-threading

Client/Server using Java via applets

The Java language started as a tool to create *applets* (mini-applications) for the World Wide Web:

1. Web browser requests an applet in an HTML page (HTML <applet> tag, which has the *class* filename).
2. Browser requests applet from server.
3. Load and execute applet (size of region on HTML page owned by the applet tag).
4. Discard applet.

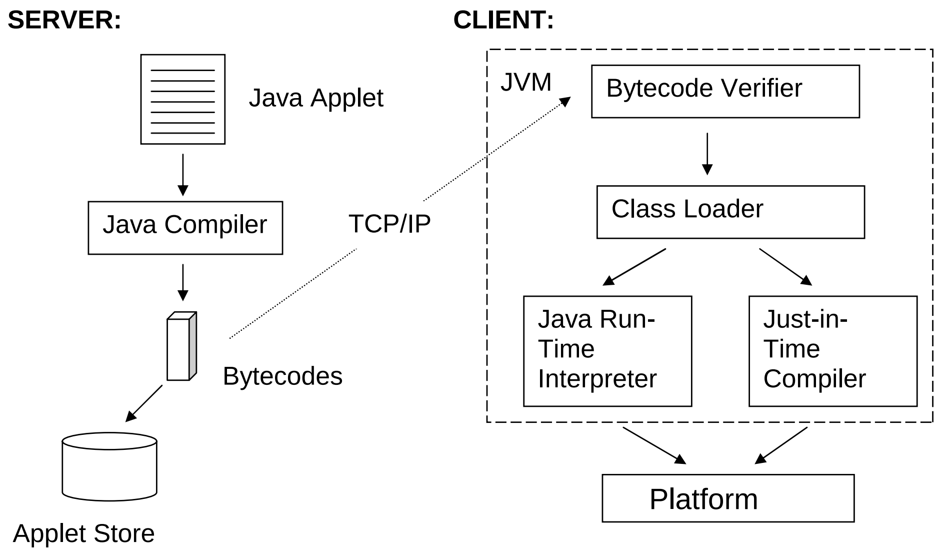


Distributed Code and Data

Usage as applets in web pages is just the most extensive application to date. Java also provides for distribution of code and data across multiple platforms. There are some important attributes required for a general mobile code system:

- Safe environment for code - precise control of applet environment such as memory access, system calls, etc.
- Platform-independent - cross platform services on variety of operating systems and hardware.
- Life cycle control - run-time support for loading, executing and unloading code and data.
- Distribution - secure transfer of code across network, certification of applets, authentication of clients and servers.

The format of distributed code is low-level but machine-independent *bytecodes*. Translation of bytecodes to machine-dependent instructions is performed by the *Java Virtual Machine*.



Java Language - Threads

For real-time distributed applications involving multiple tasks the support provided in the Java language for multi-threading is an important capability, e.g: user interaction through multiple graphical displays, acquisition of data from instrumentation, processing and display of that data, control of plant and processes in response to user requests requires multiple concurrent threads of execution.

Thread Creation

A separate thread of control is created through a thread object, e.g:

```
Thread job = new Thread();
```

To start the thread job, its `start` method is called, which spawns the new thread of control. The new thread begins execution when the JVM invokes the new thread's `run` method. Consider an example:

```
Class TwoThreads extends Thread {
  string threadname; // thread name to be printed
  int delay; // delay time msec

  TwoThreads(string name, int time) {
    threadname = name;
    delay = time;
  }

  public void run() {
    try {
      for(;;) {
        System.out.print(threadname + " ");
        sleep(delay);
      }
    } catch (InterrruptedException e) {
        return; // thread ends
    }
  }

  public static void main(String[] args) {
    new TwoThreads("Thread 1",50).start();
    new TwoThreads("Thread 2",150).start();
  }
}
```

Thread Mutual Exclusion

Synchronization is required where multiple threads need to share a resource to eliminate the possibility of corruption of the data in that resource and/or deadlock occuring. A **synchronized** method is used to *lock* an object, i.e. enforce mutually exclusive access to that object. For example, consider a multithreaded bank account object:

```
class Account {
  private double balance;
  public Account (double InitialDeposit) {
    balance = InitialDeposit;
  }
  public synchronized double getBalance() {
    return balance;
  }
  public synchronized void Deposit (double amt) {
    balance += amt;
  }
}
```

The `balance` field is protected by the synchronized access methods in the above example. An alternative approach is to use synchronized statement blocks, e.g:

```
public static void abs(int[] values) {
  synchronized (values) {
    for (int i=0; i < values.length; i++) {
      if (values[i] < 0)
        values[i] = -values[i];
    }
  }
}
```

i.e. the synchronizng lock is performed on the `values` field above so that it is cannot be changed by other threads of execution during execution of the enclosed statement block. The locked object does not need to be accessed in the statement block - it may just serve as a lock.

Thread Communication

Two methods are defined to enable threads to communicate: wait and notify, i.e. a thread may wait for an event to occur and be notified by another thread that it has occurred:

```
synchronized void doWhenCondition() {
  while (!condition)
    wait();
    ... Operations executed after wait on condition
}
```

Note the use of **`synchronized`** to ensure mutual exclusion when **`condition`** is accessed and the *atomic* release of the lock on **`condition`** when the wait is performed. Other forms of wait allow a timeout to be specified, e.g:

```
public final void wait(long timeout)
    throws InterruptedException;
        // waits timeout msecs
```

A waiting thread will be awoken by **`notify`** and if it was waiting on **`condition`** it will find it set, e.g:

```
synchronized void changeCondition() {
  condition=TRUE;
  notify();
}
```

If more than one thread is likely to be waiting, then a **`notifyAll`** should be used. Consider a multi-threaded queue example:

```
class Queue {
  Element head,tail;

  public synchronized void append (Element p) {
    if (tail == null) head = p;
    else tail.next = p;
    p.next = null;
    tail = p;
    notifyAll(); // Inform readers queue has data
  }
```

```
  public synchronized Element get() {
    try {
      while (head == null) wait();
    } catch (InterruptedException e) {
        return null;
    }
    Element p = head;    // Get queue element
    head = head.next;    // Remove from queue
    if (head == null) tail = null;
    return p;
  }
}
```

Thread Scheduling

The scheduling approach used for Java threads is *priority based run-until-blocked with round-robin*. That means that the threads at the highest priority level all get some processor time until they suspend to sleep for a time delay or execute a system or thread function that is blocked. Lower priority threads are not guaranteed processor time, but if the operating system supports *time-slicing* then lower priority threads will also receive processor time related to their priority level. **`setPriority`** is used to change thread priorities (between **`MIN_PRIORITY`** and **`MAX_PRIORITY`** from the initial default (which is the same priority as the thread spawning it) and **`getPriority`** returns the thread priority level.

Some other useful methods which influence thread scheduling are:

```
public static void sleep(long time)
    throws InterruptedException;
        // delays at least time msecs

public static void yield()
        // suspends thread until scheduler reruns
```

Consider an application that produces a list of words with each thread dedicated to producing a single word. It is run with a first parameter indicating if threads are to yield, the second parameter is the number of words to print, and the final parameter is the list of words to print:

```
class Babble extends Thread {
  static boolean doYield;
  static int howOften;
  String word;

  Babble(String whatToSay) {
    word = whatToSay;
  }

  public void run() {
    for (int i = 0; i < howOften; i++) {
      System.out.print(word + " ");
      if(doYield) yield();
    }
  }

  public static void main(String[] args) {
    howOften = Integer.parseInt(args[1]);
    doYield = new Boolean(args[0]).booleanValue();

    Thread thread = currentThread();
    thread.setPriority(Thread.MAX_PRIORITY);
    for (int i = 2; i < args.length; i++)
      new Babble(args[i]).start();
  }
}
```

Run set for not yielding/yielding, i.e:

Babble false 2 thread1 thread2

$\rightarrow$ output string could be **thread1 thread1 thread2 thread2**

Babble true 2 thread1 thread2

$\rightarrow$ output string will be **thread1 thread2 thread1 thread2**

i.e. thread yielding has ensured a fairer distribution of processor cycles.

Thread Suspension and Termination Example

```
Thread spinner;
public void userHitCancel() {
  spinner.suspend();
  if(askYesNo("Really Cancel?"))
    spinner.stop();
  else
    spinner.resume();
}
```

Thread Synchronization

Threads can synchronize through a **`join`** method, i.e. a thread can spawn another thread then wait with a join for that thread to complete, e.g:

```
class CalcThread extends Thread {
  private double Result;

  public double result() {
    return Result;
  }

  public double calculate() {
    ... Do some calculations
  }

  public void run() {
    Result = calcuate();
  }

}

class Join {
  public static void main(String[] args) {
    CalcThread calc = new CalcThread();
    calc.start();
    ... Do something else
    try {
      calc.join();
      System.out.println("result="+calc.result());
    } catch (InterruptedException e) {
        System.out.println("No answer->interrupt");
    }
  }
}
```

Note that threads can be of two types: *user* and *daemon* - all threads start as user type and can be made daemon type with the **`setDaemon(true)`** method. The application runs until the last user thread terminates, and then any remaining daemon threads are terminated.

Runnable interface

The thread class implements the **`Runnable`** interface so that any object can be made runnable by passing it to the **`Thread`** constructor. The advantage of this method of thread creation is that it is possible to turn an object that has been extended already from some class into a thread (which wouldn't otherwise be allowed due to the single inheritence restriction). Consider a previous example recast to use **`Runnable`**:

```
Class TwoThreads implements Runnable {
  string threadname; // thread name to be printed
  .
  .
  .
  public static void main(String[] args) {
     Runnable thread1 = new TwoThreads("Thread 1",50);
     Runnable thread2 = new TwoThreads("Thread 2",100);
     new Thread(thread1).start();
     new Thread(thread2).start();
  }
}
```

i.e. A new class **TwoThreads** that is runnable is created, and then two objects of this class are created (**thread1** and **thread2**) and finally new **`Thread`** objects are created for both and started.

Thread Groups

The idea behind thread groups is to provide protection from interference to a thread by threads outside its group. A hierarchy can also be established, i.e. thread groups within thread groups. The group of a thread is defined in the thread constructor:

```
public Thread(ThreadGroup group, String threadName)
```

A maximum priority can be set for a group which ensures all threads in that group cannot raise their priority above that. Along with other group specific methods, a **`checkAccess`** method is provided to determine if the current thread is allowed to modify this group.

REMOTE METHOD INVOCATION

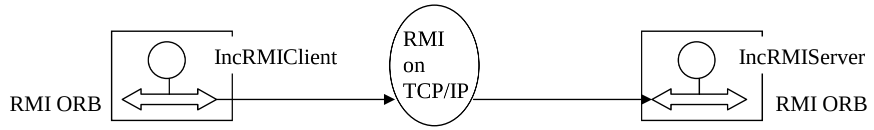
For basic communication Java supports sockets but sockets require the client and server to engage in applications-level protocols to encode and decode messages, and the design of such protocols can be error-prone.

An alternative is Remote Procedure Call (RPC), which translates poorly to distributed object systems, where communication between program-level objects residing in different address spaces is needed. To match the object invocation semantics, Remote Method Invocation (RMI) can be used.

Other RMI type systems can be adapted to handle Java objects, but these other systems, e.g. CORBA, must support a multilanguage environment and thus must have a language-neutral object model. Java's RMI can assume the environment of the Java Virtual Machine (JVM) on all systems, so advantage can be taken of the Java object model where possible.

In the Java distributed object model, a *remote object* is one whose methods can be invoked from another JVM, potentially on a different host. *A method invocation on a remote object has the same syntax as a method invocation on a local object.*

Consider a simple example with a count increment on a remote server:



```
public class IncRMIClient
{ public static void main(String args [])
  { System.setSecurityManager(new RMISecurityManager());
    try {
      IncRMI myInc = (IncRMI)Naming.lookup("rmi://"
                        +args[0] + "/" + "my IncRMI");
      myInc.increment();
    } catch(Exception e) {
      System.out.println("Exception" + e);
    }
  }
}
```

```
public interface IncRMI extends java.rmi.remote
{
  public int increment() throws java.rmi.RemoteException;
}

public class IncRMIImpl extends UnicastRemoteObject
     implements IncRMI
{
  private int count;

  .
  .
  public int increment() throws RemoteException
  {
     count++;
     return count;
  }
}

public class IncRMIServer
{ public static void main(String args [])
  { System.setSecurityManager(new RMISecurityManager());
    try {
      IncRMI myInc = new IncRMIImpl("my IncRMI");
      System.out.println("IncRMI Server ready");
    } catch(Exception e) {
      System.out.println("Exception" + e);
    }
  }
}
```

Native Methods

Java supports native methods written in other languages (typically C/C++) with obvious limitations on portability and safety protection, e.g:

```
public native void unlock() throws IO Exception;
```

The Java program must load the native language executable (which is stored as a shared or dynamic link library), e.g:

```
System.loadLibrary("Unlock");
```

The native language function or procedure can then be called as a Java method with the same syntax as other methods.

Extended Java Environments for Real-Time Distributed Systems *(ref: www.java.sun.com)*

Embedded Java (V1.1 - Jan 1999 -> J2ME Dec 2002):

The *EmbeddedJava* application environment was the Java technology aimed at devices that have dedicated functionality and limited memory. It comprised a JVM, a set of Java -based class libraries, and a set of tools. The tools could be used to configure and compile to the method level from Java class libraries a software environment that is customized to the needs of device applications. This software environment could then be burned into ROM. Now superceded by J2ME (Micro Edition) which provides enabling technologies of CLDC (Connected Limited Device Configuration) and MIDP (Mobile Information Device Profile) and various Wireless APIs.

Jini (V1.2 - Dec 2001):

Jini connection technology enables JVM based devices to plug together to form an impromptu community with each device providing services that other devices in the community can use. The *Jini* technology provides a layer above RMI and offers discovery and join protocols, a lookup service, a leasing and transaction mechanism, and the ability to move Java objects between JVMs. *Jini* Technology is made up of a core platform (JCP) and an extended platform (JXP).

JavaSpaces (V1.2 - April 2002):

JavaSpaces is a *Jini* service to connect JVM-based network resources. JavaSpaces is different to conventional databases since a more loosely coupled repository of information is provided, the identity of a client or a server is no longer relevant; data packets are treated just like any other object posted to the space as an anonymous service. In addition to data, a space can find, match and reference objects by both type and value, meaning they can store objects as information or behavior. It effectively consists of methods that enable entries to be put into a shared space.

Real-time Specification for Java (final – Jan 2002):

The approach is to introduce an API addressing the main areas of importance for a language supporting Real-time systems development, while as much as possible retaining the original "Write Once, Run Anywhere" philosophy behind the Java programming language. Other key principles were to ensure predictable execution at the expense of general purpose computing features, and to avoid extending the syntax of the Java language itself.

- Thread scheduling and dispatch: allow implementations to provide their own scheduling mechanism but provide a basic scheduler that is priority based (at least 28 levels) and pre-emptive.
- Memory management: while allowing memory management to be as automated as possible, tight control over garbage collection mechanisms must also be supported.
- Synchronization and resource sharing: support priority inversion control mechanisms.
- Asynchronous event handling, transfer of control and thread termination: extend exception handling to allow threads to change the focus of control of other threads and terminate them safely.
- Direct physical memory access: introduce a class that allows byte-level access to physical memory and allows objects to be constructed in physical memory.

The RTSJ specification is a resource for migration for better support in J2ME for RT applications.

JavaIDL (J2SE 1.4 - Dec 2001)

From JDK 1.2 the Java programming environment has included some elementary CORBA compliance. The *idlj* compiler in J2SE 1.3 provided CORBA 2.3 compliance "in some areas but not in others". The J2SE 1.4 ORB platform introduced Portable Object Adaptor (POA) and GIOP 1.2 support "along the lines of compliance with CORBA 2.3" and some parts of the CORBA 2.3.1 specification are supported.