Speculative Parallelism

Prolog also lends itself nicely to speculative parallelism. In this form of parallelism, we “guess” or speculate that some computation may be needed in the future and start it early. This speculative computation can often be done in parallel with the main (non-speculative) computation.

Recall our example of

\[
\text{member}(X, \text{list1}), \\
\quad \text{member1}(X, \text{list2}), \quad \text{isPrime}(X).
\]

After \text{member}(X, \text{list1}) has generated a preliminary solution for x, it is tested (perhaps in parallel) by \text{member1}(X, \text{list2}) and \text{isPrime}(X).

But this value of x may be rejected by one or both of these tests. If it is,
we’ll ask `member(x, list1)` to find a new binding for `x`. If we wish, this next binding can be generated speculatively, while the current value of `x` is being tested. In this way if the current value of `x` is rejected, we’ll have a new value ready to try (or know that no other binding of `x` is possible).

If the current value of `x` is accepted, the extra speculative work we did is ignored. It wasn’t needed, but was useful insurance in case further `x` bindings were needed.
Reading Assignment

- Java for C++ Programmers
  (linked from class web page)
- Roosta: Section 9.3
- Webber: Chapters 13, 15 and 17
Java & Object-Oriented Programming

Java is a fairly new and very popular programming language designed to support secure, platform-independent programming.

It is a good alternative to C or C++, trading a bit of efficiency for easier programming, debugging and maintenance.

Java is routinely interpreted (at the byte-code level), making it significantly slower than compiled C or C++. However true Java compilers exist, and are becoming more widespread. (IBM’s Jalapeno project is a good example). When compiled, Java’s execution speed is close to that of C or C++.
Subclassing in Java

When a new class is defined in terms of an existing class, the new class extends the existing class. The new class inherits all public and protected members of its parent (or base) class. The new class may add new methods or fields. It may also redefine inherited methods or fields.

class Point {
    int x, y;
    Point(int xin, int yin) {
        x = xin; y = yin;
    }
    static float dist(Point P1, Point P2) {
        return (float) Math.sqrt((P1.x-P2.x)*(P1.x-P2.x) + (P1.y-P2.y)*(P1.y-P2.y));
    }
}
class Point3 extends Point {
    int z;
    Point3(int xin, int yin, int zin) {
        super(xin, yin); z=zin;
    }
    static float dist(Point3 P1, Point3 P2) {
        float d=Point.dist(P1, P2);
        return (float) Math.sqrt((P1.z-P2.z)*(P1.z-P2.z)+
                                 d*d);
    }
}

Note that although Point3 redefines dist, the old definition of dist is still available by using the parent class as a qualifier (Point.dist). The same is true for fields that are hidden when a field in a parent is redeclared.
Non-static methods are automatically virtual: a redefined method is automatically used in all inherited methods including those defined in parent classes that think they are using an earlier definition of the class.

Example:

class C {
    void DoIt() {PrintIt();}
    void PrintIt()
        {println("C rules!");}
}

class D extends C {
    void PrintIt()
        {println("D rules!");}
    void TestIt() {DoIt();}
}
D dvar = new D();
dvar.TestIt();
D rules! is printed.
Static methods in Java are not virtual (this can make them easier to implement efficiently).
Abstract Classes and Methods

Sometimes a Java class is not meant to be used by itself because it is intentionally incomplete.
Rather, the class is meant to be starting point for the creation (via subclassing) of more complete classes.
Such classes are abstract.
Example:
abstract class Shape {
    Point location;
}
class Circle extends Shape {
    float radius;
}
Methods can also be made abstract to indicate that their actual definition will appear in subclasses:

```java
abstract class Shape {
    Point location;
    abstract float area();
}

class Circle extends Shape {
    float radius;
    float area() {
        return Math.pi * radius * radius;
    }
}
```
Subtyping and Inheritance

We can use a subtyping mechanism, as found in C++ or Java, for two different purposes:

- We may wish to inherit the actual implementations of classes and members to use as the basis of a more complete or extended class.

  To inherit an implementation, we say a given class “extends” an existing class:

  ```
  class Derived extends Base
  { ... }
  
  Class Derived contains all of the members of Base plus any others it cares to add.
  ```
We may wish to inherit an interface—a set of method names and values that will be available for use. To inherit (or claim) an interface, we use a Java interface definition. An interface doesn’t implement anything; rather, it gives a name to a set of operations or values that may be available within one or more classes.
Why are Interfaces Important?

Many classes, although very different, share a common subset of values or operations. We may be willing to use any such class as long as only interface values or operations are used.

For example, many objects can be ordered (or at least partially-ordered) using a “less than” operation.

If we always implement less than the same way, for example,

```java
boolean lessThan(Object o1, Object o2);
```

then we can create an interface that admits all classes that know about the lessThan function:
interface Compare {
    boolean lessThan(Object o1,
                     Object o2);
}

Now different classes can each implement the Compare interface, proclaiming to the world that they know how to compare objects of the class they define:

class IntCompare implements Compare {
    public boolean lessThan(Object i1,
                             Object i2){
        return ((Integer)i1).intValue() <
               ((Integer)i2).intValue();
    }
}
class StringCompare implements Compare {
    public boolean lessThan(Object i1,
                             Object i2){
        return
               ((String)i1).compareTo((String)i2)<0;
    }
}
The advantage of using interfaces is that we can now define a method or class that only depends on the given interface, and which will accept any type that implements that interface.

```java
class PrintCompare {
    public static void printAns(
        Object v1, Object v2, Compare c) {
        System.out.println(
            v1.toString() + " < " +
            v2.toString() + " is " +
            new Boolean(c.lessThan(v1, v2))
                .toString());
    }
}

class Test {
    public static void main(String args[]) {
        Integer i1 = new Integer(2);
        Integer i2 = new Integer(1);
        PrintCompare.printAns(
            i1, i2, new IntCompare());
        String s2 = "abcdef";
        String s1 = "xyzaa";
        PrintCompare.printAns(
            s1, s2, new StringCompare());
    }
}
```
Since classes may have many methods and modes of use or operation, a given class may implement many different interfaces. For example, many classes support the `Clonable` interface, which states that objects of the class may be duplicated (cloned).
Multiple Inheritance

We have seen that a class may be derived from a given parent class. It is sometimes useful to allow a class to be derived from more than one parent, inheriting members of all parents. This is multiple inheritance; it is allowed by C++ and Python, but not by Java or C#.

The basic idea is that sometimes we want a “composite” object formed from more than one source. Hence a Computer object can be viewed as both a PhysicalObject (with height, weight, color, cost, etc.) and also a CPUImplementation (with memory size, processor design, processor speed, I/O ports, etc.)
Using multiple inheritance we merge aspects of a PhysicalObject and a CPUImplementation, and perhaps add additional data:

class PhysicalObject {
    float height, width, weight;
    Color outsideColor;
    ...
}
class CPUImplementation {
    CPUClass CPUKind;
    int memorySize, CPUSpeed;
    ...
}

class Computer: PhysicalObject, CPUImplementation {
    String myURL; ...
}

The advantages of multiple inheritance are obvious—you can build a class from many sources rather than just one.
There are problems though:

- If the same name appears in more than one parent, which is used? For example, if both parents contain a “copyright” field, which do you get? C++ forbids access to fields common to several parents, though accidental clashes of member names are certainly possible. Python relies on order of specification of the parents (which can be somewhat arbitrary).

- Access to fields and methods can be less efficient. A method in a parent class can’t know how fields in derived class will be allocated when multiple parents may exist. Hence some form of indirection may be needed. For example, in class PhysicalObject, we may believe
that height is the first field allocated, while in
CPUImplementation we may believe that CPUKind is allocated first. But in class Computer, which contains both height and CPUKind, both fields can’t come first.
Threads and Parallelism in Java

Java is one of the few “main stream” programming languages to explicitly provide for user-programmed parallelism in the form of threads. A Java programmer may organize a program as several threads that may execute concurrently.

Even if the program is run on a uni-processor, use of threads may improve performance. This is because the threads can be multi-programmed, with threads switched automatically on I/O delays, page faults or even cache misses.

A program that is designed to support multiple threads is also “prepared” for future upgrades to multiprocessors or multi-threaded processors.
Java Threads

In Java any class that implements the Runnable interface can be started as a concurrent thread:

```java
interface Runnable {
    public void run();
}
```

When the thread is started, the method `run` begins to execute (perhaps concurrently with other threads of the main program).

You create a thread using the `Thread` constructor:

```java
new Thread(RunnableObject)
```

Creating a thread does not start it; you must execute the `start` method within the `Thread` object.
When this is done, the `run` method immediately starts, and continues until that method terminates normally, or it throws an uncaught exception, or it is explicitly stopped, or the main program stops.

On uniprocessors, thread execution is interleaved; on multiprocessors or multithreaded architectures execution can be concurrent.

class DoSort implements Runnable {
    int [] data;
    DoSort(int[] in) {data=in;}
    public void run(){
        // sort the data array;
    }
}

class Test {
    public static void
    main(String args[]) {
        DoSort d =
            new DoSort(new int[1000]);
        Thread t1 = new Thread(d);
        t1.start();
        // We can continue while t1
        // does its sort
    }
}
We can start multiple threads, and the threads can use `sleep` to delay or synchronize their execution:

```java
class PingPong
    implements Runnable {
    int delay; String word;
    PingPong(String s, int i) {
        delay = i; word = s;
    }
    public void run() {
        try {
            while (true) {
                System.out.print(word + " ");
                Thread.sleep(delay);
            }
        } catch (InterruptedException e) {
            //
        }
    }
}
```
public static void
main(String args[]){
    Thread t1 = new Thread(
        new PingPong("ping",33));
    Thread t2 = new Thread(
        new PingPong("PONG",100));
    t1.start();
    t2.start();
}

ping PONG ping ping ping PONG ping ping ping PONG ping ping ...

Synchronization in Java

We often want threads to co-operate, typically in how they access shared data structures.

Since thread execution is asynchronous, the details of how threads interact can be unpredictable.

Consider a method

```java
update() {
    n = n+1;
    val = f(n);
}
```

that updates fields of an object.

If two or more threads execute `update` concurrently, we might get unexpected or even illegal behavior. (Why?)
A Java method may be synchronized, which guarantees that at most one thread can execute the method at a time. Other threads wishing access, are forced to wait until the currently executing thread completes.

Thus

```java
void synchronized update() { ... }
```
can safely be used to update an object, even if multiple threads are active.

There is also a synchronized statement in Java that forces threads to execute a block of code sequentially.

```java
synchronized(obj) {
    obj.n = obj.n+1;
    obj.val = f(obj.n);
}
```
Synchronization Primitives

The following operations are provided to allow threads to safely interact:

wait()  Sleep until awakened
wait(n)  Sleep until awakened or until n milliseconds pass
notify()  Wake up one sleeping thread
notifyAll()  Wake up all sleeping threads

Using these primitives, correct concurrent access to a shared data structure can be programmed.
Consider a Buffer class in which independent threads may try to store or fetch data objects:

class Buffer {
    private Queue q;
    Buffer() { q = new Queue(); }
    public synchronized void put(Object obj) {
        q.enqueue(obj);
        notify(); //Why is this needed?
    }
    public synchronized Object get() {
        while (q.isEmpty()) {
            //Why a while loop?
            wait();
        }
        return q.dequeue();
    }
}
Locks, Semaphores and Monitors

Java’s synchronization mechanisms are based upon the notion of a lock. A lock is a special value that can be held by at most one thread.

If a thread holds a lock, it has permission to do some “critical” operation like writing a shared variable or restructuring a shared data object.

If a thread wants to do an operation but doesn’t hold the necessary lock, it must wait until it gets the lock.

In Java there is a lock associated with each run-time object.
Lock Granularity and Access

Though each Java object has a lock, you often don’t want to lock and unlock each object you access.

If you are manipulating a large data structure (like a tree or hash table), acquiring the lock for each object in the tree or table can be costly and error-prone.

Instead, it is common to create a lock corresponding to a group of objects. Hence holding the lock to the root of a tree may give you permission to access the whole tree.

There is a danger though—if all or most of a large structure is held by one thread, then other threads won’t be able to access that structure concurrently.
For example, a large shared data base (like one used to record current bookings for an airline) shouldn’t be held exclusively by one thread—hundreds of concurrent threads may want to access it at any time. An intermediate lock, like all reservations for a single fight, is more reasonable. There is also an issue of how long you hold a lock. The ideal is to have exclusive access for as short a period as is necessary. Work that is not subject to interference by other threads (like computations using local variables) should be done before a lock is obtained. Hence Java’s synchronized statement allows a method to get exclusive access to an object for a limited region, enhancing shared access.