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:

```java
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 uniprocessor, 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.

```java
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:

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 ping PONG
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.
**Deadlock**

A variety of programming problems appear in concurrent programs that don’t exist in ordinary sequential programs.

The most serious of these is deadlock:

Two or more threads hold locks that other threads require. Each waits for the other thread to release a needed lock, and no thread is able to execute.

As an example of how deadlock may occur, consider two threads, \( t_1 \) and \( t_2 \). Each requires two files, a master file and a log file. Since these files are shared, each has a lock.

Assume \( t_1 \) gets the lock for the master file while \( t_2 \) (at the same instant) gets the lock for the log file.
Now each is stuck. Each has one file, and will wait forever for the other file to be released.

In Java deadlock avoidance is wholly up to the programmer. There are no language-level guarantees that deadlock can’t happen.

Some languages have experimented with ways to help programmers avoid deadlock:

- If all locks must be claimed at once, deadlock can be avoided. You either get all of them or none, but you can’t block other threads while making no progress yourself.

- Locks (and the resources they control) can be ordered, with the rule that you must acquire locks in the
proper order. Now two threads can’t each hold locks the other needs.

- The language can require that the largest set of locks ever needed be declared in advance. When locks are requested, the operating system can track what’s claimed and what may be needed, and refuse to honor unsafe requests.
Fairness & Starvation

When one thread has a lock, other threads who want the lock will be suspended until the lock is released. It can happen that a waiting thread may be forced to wait indefinitely to acquire a lock, due to an unfair waiting policy. A waiting thread that never gets a lock it needs due to unfair lock allocation faces starvation.

As an example, if we place waiting threads on a stack, newly arrived threads will get access to a lock before earlier arrivals. This can lead to starvation. Most thread managers try to be fair and guarantee that all waiting threads have a fair chance to acquire a lock.
How are Locks Implemented?

Internally, Java needs operations to acquire a lock and release a lock. These operations can be implemented using the notion of a semaphore. A semaphore is an integer value (often just a single bit) with two atomic operations: up and down.

up(s) increments s atomically.

down(s) decrements s atomically. But if s is already zero, the process doing the down operation is put in a wait state until s becomes positive (eventually some other process should do an up operation).

Now locks are easy to implement. You do a down(lock) to claim a lock. If someone else has it, you are forced
to wait until the lock is released. If the lock value is > 0 you get it and all others are “locked out.”

When you want to release a lock, you do `up(lock)`, which makes `lock` non-zero and eligible for another thread to claim.

In fact, since only one thread will ever have a lock, the lock value needs to be only one bit, with 1 meaning currently free and unlocked and 0 meaning currently claimed and locked.
Monitors

Direct manipulation of semaphores is tedious and error-prone. If you acquire a lock but forget to release it, threads may be blocked forever.

Depending on where `down` and `up` operations are placed, it may be difficult to understand how synchronization is being performed.

Few modern languages allow direct use of semaphores. Instead, semaphores are used in the implementation of higher-level constructs like monitors.

A monitor is a language construct that guarantees that a block of code will be executed synchronously (one thread at a time).
The Java `synchronized` statement is a form of monitor.

When

```java
synchronized(obj) { ... }
```

is executed, “invisible” `getLock` and `freeLock` operations are added:

```java
synchronized(obj) {
    getLock(obj)
    ...
    freeLock(obj);
}
```

This allows the body of the `synchronized` statement to execute only when it has the lock for `obj`. Thus two different threads can never simultaneously execute the body of a `synchronized` statement because two threads can’t simultaneously hold `obj`’s lock.
In fact, synchronized methods are really just methods whose bodies are enclosed in an invisible synchronized statement.

If we execute

```java
obj.method()
```

where `method` is synchronized, `method`'s body is executed as if it were of the form

```java
synchronized(obj) {
    body of method
}
```

Operations like `sleep`, `wait`, `notify` and `notifyAll` also implicitly cause threads to release locks, allowing other threads to proceed.