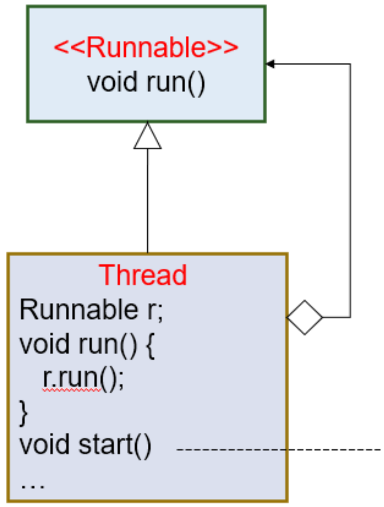
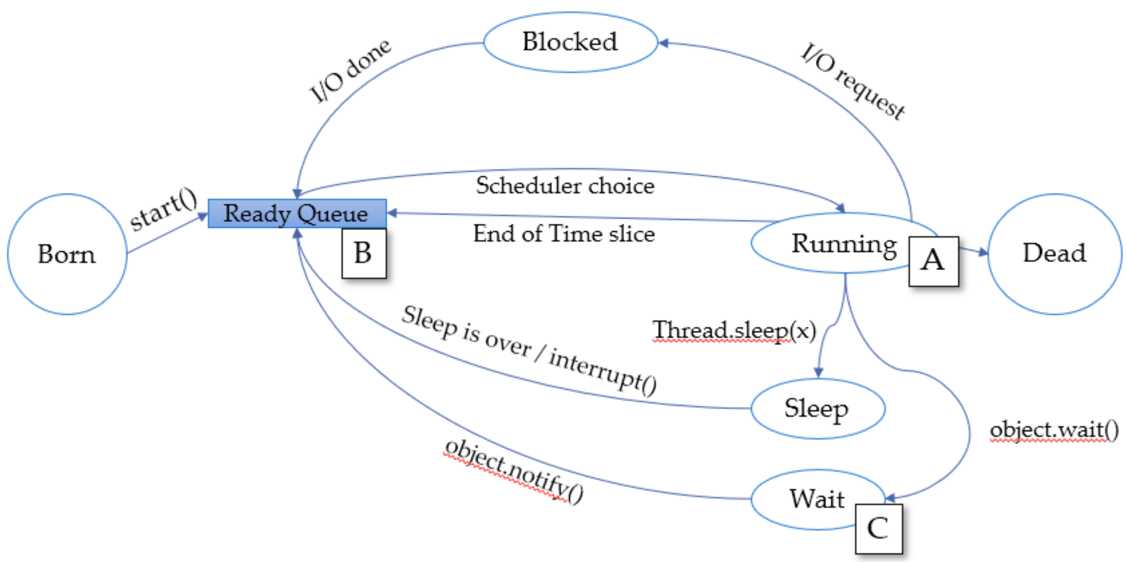
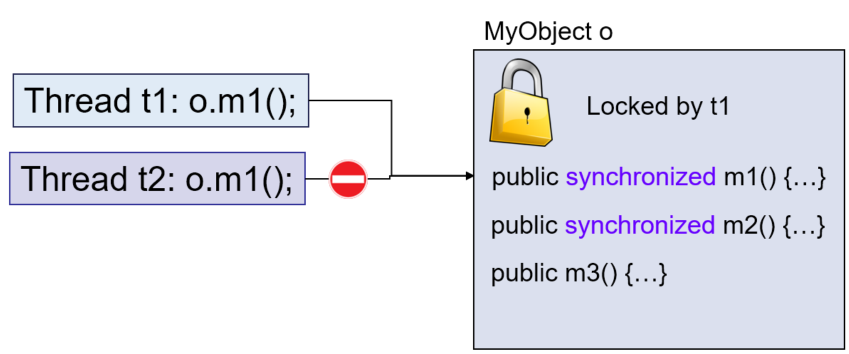
תזכורות פת"מ2







Basic Active Object

```java
class MyActiveModel implements Model {

  Maze maze;
  Solution solution;
  BlockingQueue<Runnable> dispatchQueue
     = new LinkedBlockingQueue<Runnable>();

  public MyActiveModel() {

    new Thread(new Runnable() {
      public void run() {
        while (true) {
          try {
            // take() blocks, so no busy waiting
            dispatchQueue.take().run();
          } catch (InterruptedException e) {}
        }
      }
    }).start();

  }
```

```java
void generateMaze() throws InterruptedException {
  dispatchQueue.put(new Runnable() {
    public void run() {
      maze = MazeGenerator.generateMaze(/**/);
    }
  });
}

void solve(Maze m) throws InterruptedException  {
  dispatchQueue.put(new Runnable() {
    public void run() {
      solution = searcher.search(m);
    }
  });
}
```

Thread Pool

```java
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
//...
public static void main(String[] args) {
 ExecutorService executor =
          Executors.newFixedThreadPool(2);
 executor.execute (new RunnableTask1 ());
 executor.execute (new RunnableTask2 ());
 executor.execute (new RunnableTask3 ());
}
```

The Solution – Future!

Future <V>
V value;
set(V v); V get();

```java
public class MyCallable implements Callable<Worker>{

        Worker call() throws Exception{
                // after 10 minutes or so…
                return someWorker;
        }
}
```

```java
ExecutorService executor = Executors. newFixedThreadPool (2);

Future<Worker> f = executor.submit (new MyCallable ());
// ...
Worker w = f.get(); // waits for the call() to return
```

Scheduling Tasks – with a simple Timer

```java
import java.util.Timer;
import java.util.TimerTask;
public class ThreadTest {
 private static class Ping extends TimerTask{
  public void run(){System.out.println("ping");}
 }
 private static class Pong extends TimerTask{
  public void run(){System.out.println("pong");}
 }
 public static void main(String[] args){
   Ping ping=new Ping();
   Pong pong=new Pong();
   Timer t=new Timer();
   t.scheduleAtFixedRate(ping, 0, 1000);
   t.scheduleAtFixedRate(pong, 500, 1000);
 }
}
```

Canceling tasks:

```java
int i;
while((i=System.in.read())!=13);
ping.cancel(); // canceled task
pong.cancel(); // t continues…
t.cancel(); // t is cancled
```

```java
import java.util.concurrent.atomic.AtomicInteger;
public class Count {
 AtomicInteger count = new AtomicInteger(0);
 public  void setCount(int x){count.set(x);}
 public int getCount(){return count.get();}
 public void update(){
  count.incrementAndGet();// ++count
 }
}
```

```java
public void run() {
    boolean w=W.tryLock();
    boolean r=R.tryLock();
    try{
        if(w && r){
            // do the writing...
            // do some reading...
            // do more writing...
        } else{
            // try again later...
        }
    }finally{
         if(w) W.unlock();
         if(r) R.unlock();
    }
}
```

Thread Safe Containers

- java.util.concurrent introduced **Thread Safe** containers,
- that also provides good performance!
  - ArrayBlockingQueue<E>
  - ConcurrentHashMap<K,V>
  - ConcurrentLinkedQueue<E>
  - etc…

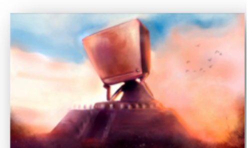
```
public class Fib_FJ extends RecursiveTask<Integer>{
 // with fork-join pool
 int num;
 public Fib_FJ(int num) { this.num=num; }

 @Override
 public Integer compute(){ // a recursive task
  if(num<=1)
   return num;
  Fib_FJ fib1= new Fib_FJ(num-1);
  fib1.fork();
  Fib_FJ fib2= new Fib_FJ(num-2);
  return fib2.compute()+fib1.join();
 }

 public static void main(String[] args) {
  Fib_FJ fib=new Fib_FJ(45);
  ForkJoinPool pool = new ForkJoinPool();
  System.out.println(pool.invoke(fib));
 }
}
```

Using CompletableFuture

```
public String deepThought(){
 // takes a really really long time...
 return "42";
}
```



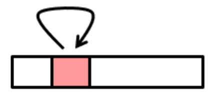
```
CompletableFuture.supplyAsync( ()->{return deepThought();})
        .thenApply(answer->Integer.parseInt(answer))
        .thenApply(x->x*2)
        .thenAccept(answer->System.out.println("answer: "+answer));
```

אופטימיזציות

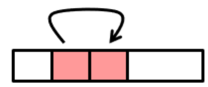
Locality

- Principle of Locality: Programs tend to use data and instructions
  - with addresses near or equal to those they have used recently
  - or same as those they have used recently

- Temporal locality:
  - Recently referenced items are likely
    to be referenced again in the near future



- Spatial locality:
  - Items with nearby addresses tend
    to be referenced close together in time



Machine-Independent Opt. Summary

Code Motion

- *Compilers are good at this for simple loop/array structures*
- *Don't do well in presence of procedure calls and memory aliasing*

Reduction in Strength

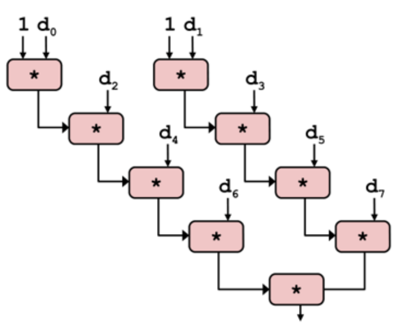
- Shift, add instead of multiply or divide
  - *compilers are (generally) good at this*
  - *Exact trade-offs machine-dependent*
- Keep data in registers rather than memory
  - *compilers are not good at this, since concerned with aliasing*

Share Common Subexpressions

- *compilers have limited algebraic reasoning capabilities*
- *Help compiler overcome aliasing, use local variables*

Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



What changed:
Two independent "streams" of operations

Overall Performance
N elements, D cycles latency/op
Should be (N/2+1)*D cycles:
CPE = D/2
CPE matches prediction!

Sometimes branches can be avoided

```
int sum(int[] a) {
  int resM = 1, resA = 0;
  for (int i = 0; i < 1000; ++i) {
       if (i % 2) resM *= a[i];
       else resA += a[i];
  }
  return resM + resA;
}
```

Prediction might be very wrong

```
int sum(int[] a) {
  int resM = 1, resA = 0;
  for (int i = 1; i < 1000; i += 2)
       resM *= a[i];

  for (int i = 0; i < 1000; i += 2)
       resA += a[i];

  return resM + resA;
}
```

Write Code Suitable for Implementation with Conditional Moves

GCC is able to generate conditional moves for code written in a more "functional" style

```
// imperative style
void minmax1(int a[], int b[], int n) {
  int i;
  for (i = 0; i < n; i++) {
        if (a[i] > b[i]) {
                int t = a[i];
                a[i] = b[i];
                b[i] = t;
        }
  }
}
```

CPE of around 14.50 for random data, and 3.00–4.00 for predictable data
a clear sign of a high misprediction penalty

```
// more "functional" style
void minmax2(int a[], int b[], int n) {
  int i;
  for (i = 0; i < n; i++) {
        int min = a[i] < b[i] ? a[i] : b[i];
        int max = a[i] < b[i] ? b[i] : a[i];
        a[i] = min;
        b[i] = max;
  }
}
```

CPE of around 5.0 regardless of whether the data are arbitrary or predictable

JVM

Optimization Tip 1: call native methods

Optimization Tip 2: use intern() to compare strings

Optimization Tip 3:
- make a smart use of *final, private,* and *static* methods
- Try to keep the number of subclasses low
- Favor composition over inheritance

Optimization Tip 4:
- put the more likely branch first
- Sort your data to make it more predictable

Optimization Tip 5: Use local objects in small scopes