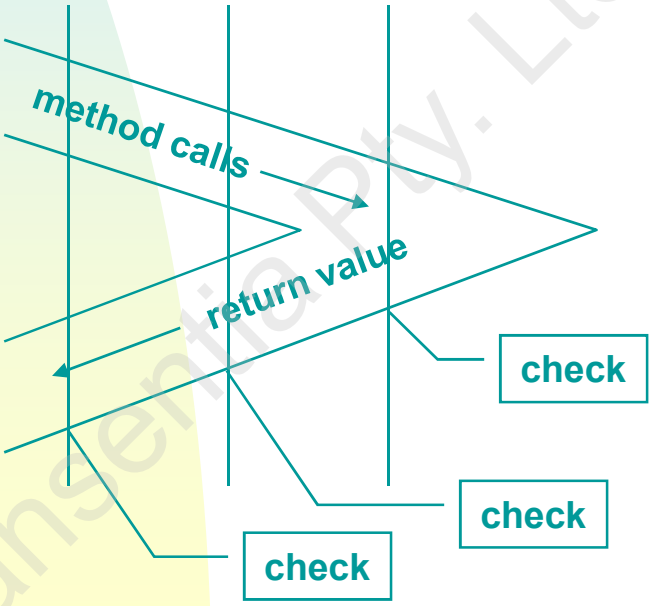
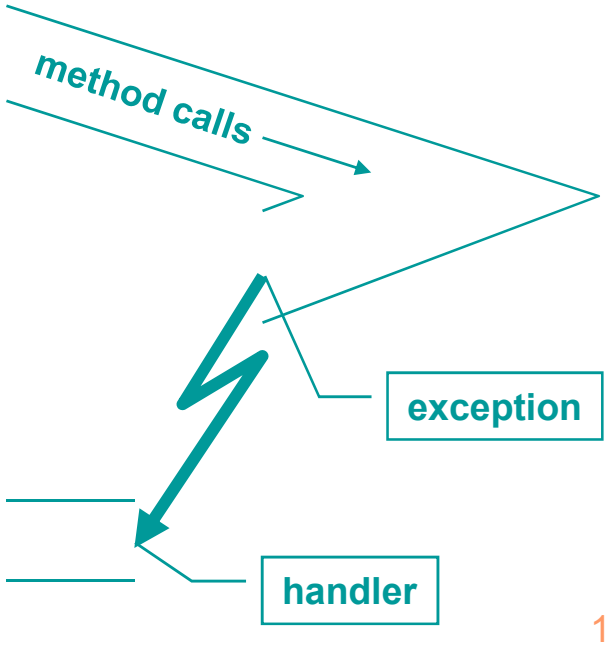
Exceptions & Threads

- Exceptions

  "If anything can go wrong, it will.
  —Finagle's Law (often incorrectly attributed to Murphy, whose law is rather different—which only goes to show that Finagle was right)"

  - a mechanism for generating "out-of-band" notifications regarding the occurrence of exceptional events





Exceptions & Threads

- Exceptions extend java.lang.Throwable
  - 2 main categories:
    - Exception: programmatic faults, etc.
    - 2 sub-categories:
      - *"if it's a* RuntimeException*, it's your fault. Fix your code."*
        - ArrayIndexOutOfBoundsException, bad cast, etc.
        - don't need to explicitly declare these; assumed to be widespread ("unchecked exceptions")
      - others
        - read past end of file, malformed URL, etc.
    - Error: internal defects and resource exhaustions
      - not for mere mortals
      - not usually explicitly mentioned

Exceptions & Threads

- Methods must state the (non-RuntimeException) exceptions they throw (define the OOB channel):

```
public int myDivide (int x, int y) throws ArithmeticException
  {
  if (y == 0)
    throw new ArithmeticException ();
  else
    return (x / y);
  }
```

- Exception/constructor isn't anything special:

```
public int myDivide (int x, int y) throws MyArithmeticException
  {
  if (y == 0)
    throw new MyArithmeticException ("Naughty, naughty!", 123);
  else
    return (x / y);
  }
```

Exceptions & Threads

```
class MyArithmeticException extends java.lang.ArithmeticException
  {
  private int errorNo;

  public MyArithmeticException (String s, int e)
    {
    super (s);
    errorNo = e;
    }

  public int getErrno ()
    {
    return (errorNo);
    }
  }
```

Exceptions & Threads

- Dealing with exceptions

```
// declare file, num, denom
try
  {
  // open file, read num, denom…
  int result = myDivide (num, denom);
  }
catch (final MyArithmeticException e)
  {
  stdout.println (e.getMessage () + " " + e.getErrno ());
  e.printStackTrace ();
  throw e;   // to enclosing environment
  }
catch (final IOException e)
  { …
  }
catch (final Exception e)
  { …
  }
finally
  {
  // close file…guaranteed regardless
  }
```

order: specific -> general

Introduced in 

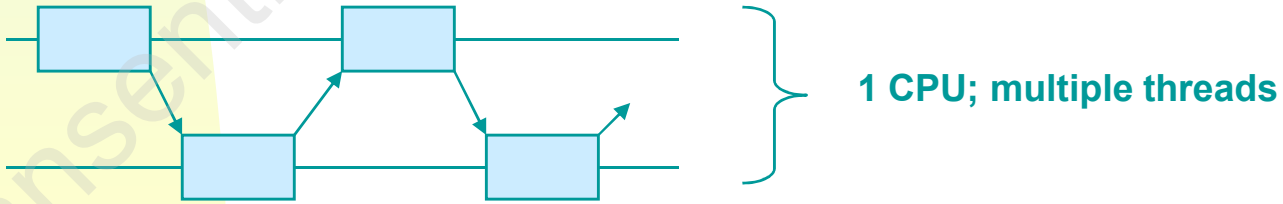
Exceptions & Threads

- Exceptions are not a panacea
  - intended for *rare*, *exceptional* events
  - slow: lots of housekeeping involved
  - don't micromanage
    - catch everything individually
  - don't squelch
    - catch everything generically
    - ignore
- Experience plays a large part in defining exceptions

  - eg. java.io.EOFException

Exceptions & Threads

- Threads
  - objects and parallelism naturally go together
    - object encapsulates data & processing *specification*
    - may as well encapsulate actual *processing* …
  - give illusion of doing > 1 thing at a time
    - better apparent performance
    - better resource utilization
    - needed for applets over a slow internet!



Exceptions & Threads

- Thread defined by:
  - implementing java.lang.Runnable
    - can be used whether or not a class extends another

```
class AnyClass extends Applet implements Runnable
  { … }
```

    - this is the ‘preferred’ method
  - extending java.lang.Thread
    - only possible if class doesn’t already extend something

```
class AnyClass extends Thread
  { … }
```

OR

Exceptions & Threads

```
public class EZTest
  {
  public static void main (String [] args)
    {
    new Thread (new EZThread (), "hickory").start ();
    new Thread (new EZThread (), "dickory").start ();
    new Thread (new EZThread (), "dock").start ();
    }
  }

class EZThread implements Runnable
  {
  public void run () // eventually called by Thread.start ()
    {
    String threadName = Thread.currentThread ().getName ();

    for (int x = 0; x < 3; x ++)
      {
      System.out.println (x + " " + threadName);
      try { Thread.sleep ((int) (Math.random () * 1000)); }
      catch (InterruptedException ie) { }
      }
    System.out.println (threadName + " has expired");
    }
  }
```

Exceptions & Threads

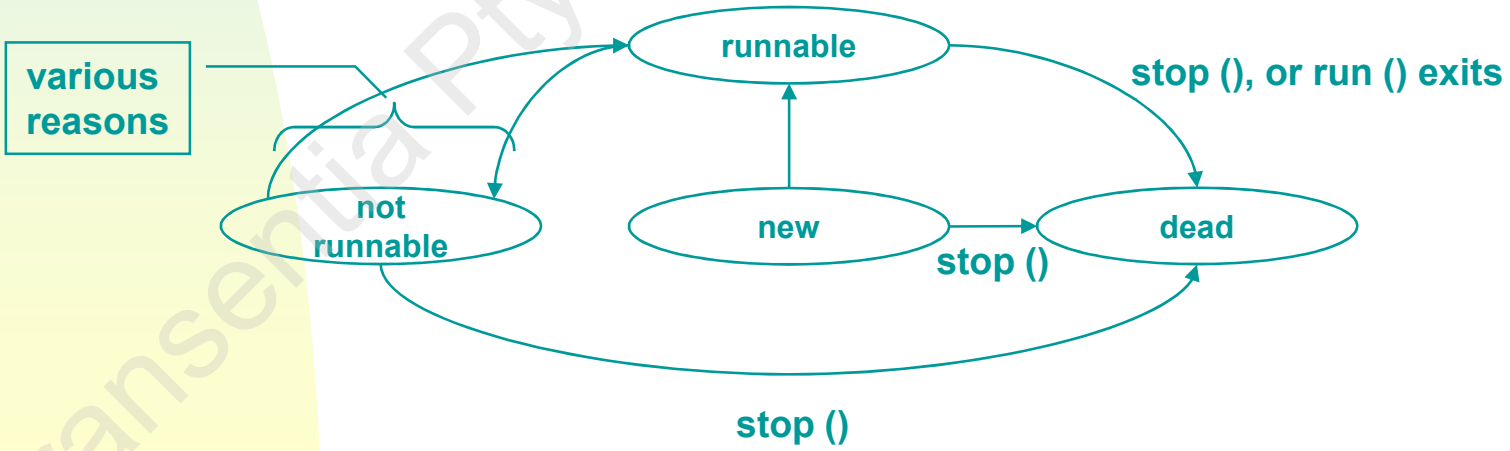
```
public class EZTest
  {
  public static void main (String [] args)
    {
    new EZThread ("hickory").start ();
    new EZThread ("dickory").start ();
    new EZThread ("dock").start ();
    }
  }

class EZThread extends Thread
  {
  public EZThread (String str)
    { super (str); }

  public void run () // called by start ()
    {
    for (int x = 0; x < 3; x ++)
      {
      System.out.println (x + " " + getName ());
      try { sleep ((int) (Math.random () * 500)); }
      catch (final InterruptedException ie) {}
      }
    System.out.println (getName () + " has expired");
    }
  }
```

```
D:\Bob\Java\EZThread>java EZTest
0 hickory
0 dickory
0 dock
1 hickory
1 dock
1 dickory
2 hickory
2 dickory
dickory has expired
2 dock
hickory has expired
dock has expired
```

note *random, undefined* ordering

Exceptions & Threads

- 4 possible thread states (lifecycle)
  - new
  - runnable
  - not runnable
    - waiting, suspended, sleeping, blocked on I/O, etc.
  - dead



Exceptions & Threads

- Thread exceptions
  - IllegalThreadStateException
    - e.g. resume () on sleeping thread
  - InterruptedException
  - IllegalMonitorStateException
    - try to wait but not in synchronized code
  - ThreadDeath (extends *java.lang.Error*)
    - sent to kill a thread
    - can be caught to allow cleanup but *must* be re-thrown
  - etc.

Exceptions & Threads

- Some thread methods
  - sleep
  - yield
    - accounts for “green threads” implementations
  - isAlive
    - coarse: can’t distinguish between various states
  - setPriority
    - at any given time, the “runnable” thread with the highest priority will be running
  - setDaemon
    - background thread
    - Java machine will exit if only daemon threads alive

Exceptions & Threads

- Thread groups
  - all threads are members of a group
  - ∃ default group
  - provides a way of dealing with related threads in one go
    - set priority bands
    - start, suspend, etc.
    - apply access privileges
    - etc.

Exceptions & Threads

- Synchronization
  - races, deadlocks, etc. ***highly*** problematic
  - Java provides Hoare monitors
    - every object has a lock
    - synchronized methods use lock to ensure only 1 thread active within a monitor instance at any time

```
synchronized int myMethod ()
  { … }
```

    - synchronizing whole method inefficient

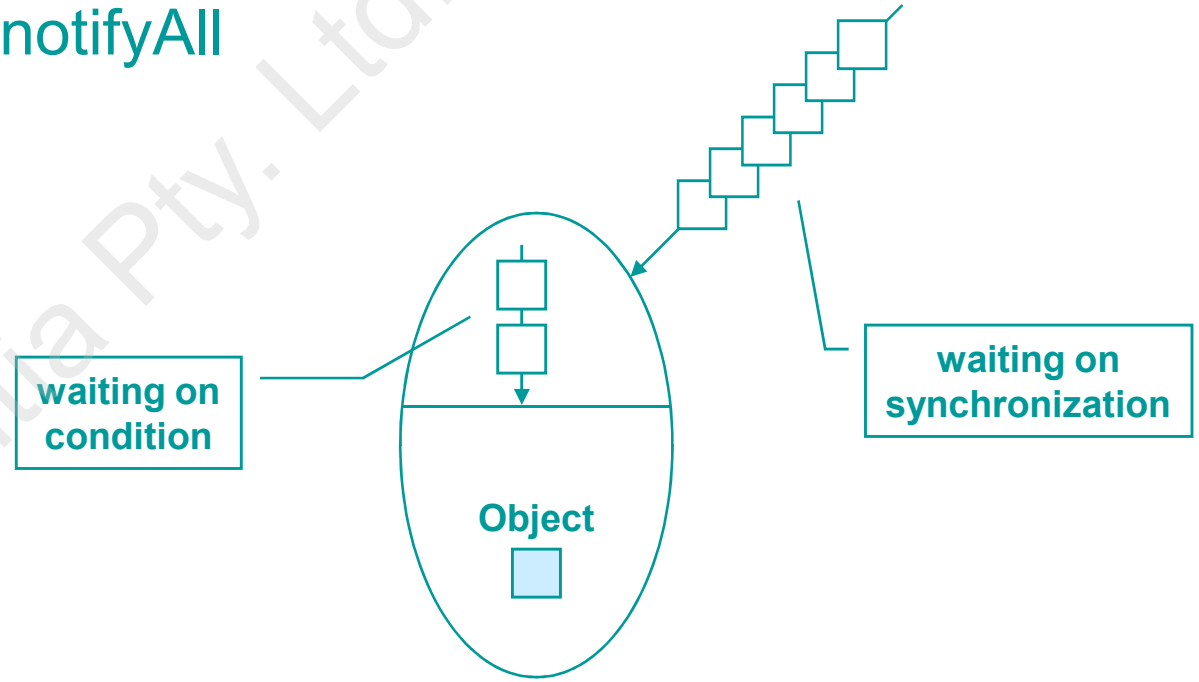
```
int myMethod ()
  {
  …                  // pre (or post) non-critical sections
  synchronized (anObjectOrThis)
    { … }
  }
```

Exceptions & Threads

- can also synchronize to the *class*
  - (actually, to the associated java.lang.Class object…)
  - static synchronized method ()
    - serialize shared access to static data

Exceptions & Threads

- Synchronized not enough
  - just provides *mutual exclusion* to a *critical section*
  - also need to account for changing conditions
    - wait
      - (releases lock; waits; recovers lock; continues…)
    - notify, notifyAll



Exceptions & Threads

```
class Monitor extends SimpleBoundedBuffer
 {
 public synchronized void put (int x) throws InterruptedException
   {
   while (numberContained == MAX) wait ();

   add_it (x);

   if (numberContained++ == 0) notifyAll ();
   }
 public synchronized int get () throws InterruptedException
   {
  while (numberContained == 0) wait ();

   int it = remove_it ();

   if (numberContained-- == MAX) notifyAll ();

   return (it);
   }
...
```

Exceptions & Threads

- Notify versus notifyAll
  - 1 vs. all
  - consider notify an optimised form of notifyAll
    - e.g. notify is not fair: doesn't account for priority or length of time a thread has been waiting