CSC 7322 : Object Oriented Development

J **Paul** Gibson, A207

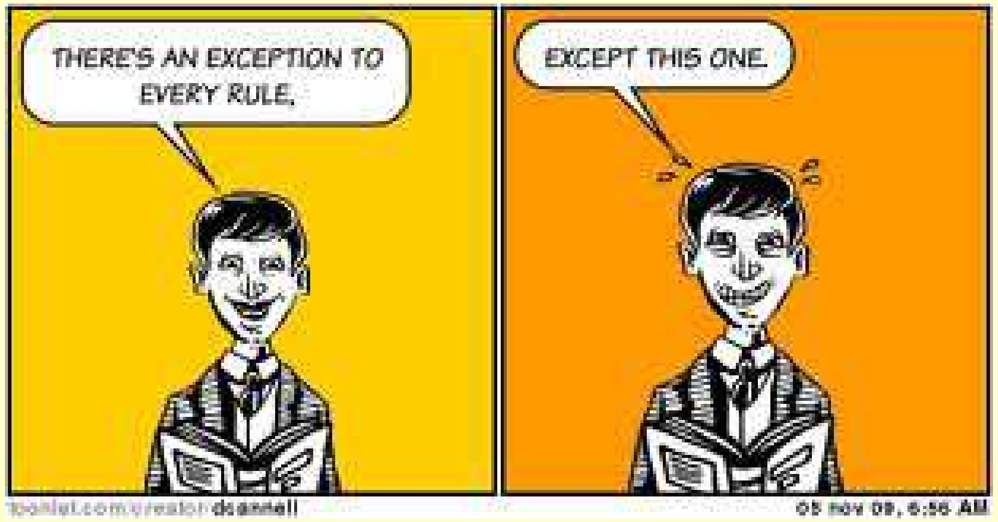
paul.gibson@int-edu.eu

http://www-public.it-sudparis.eu/~gibson/Teaching/CSC7322/

Exceptions and Threads(in Java)

…/~gibson/Teaching/CSC7322/L8-ExceptionsAndThreads.pdf

Exceptions


THERE'S AN EXCEPTION TO EVERY RULE.
EXCEPT THIS ONE.
toonlet.com/creator/dcannell
05 nov 09, 6:56 AM

"When certain concepts of TeX are introduced informally, general rules will be stated; afterwards you will find that the rules aren't strictly true. In general, the later chapters contain more reliable information than the earlier ones do. The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions.", Donald Knuth

Exceptions – General History

The control of exceptional conditions in PL/I object programs
JM Noble - Proc. IFIP Congress, 1968.

Exception Handling: Issues and a Proposed Notation
John B. Goodenough Commun. ACM, 1975

Software reliability: The role of programmed exception handling, **Melliar-Smith, P. M. and Randell, B, SIGSOFT Softw. Eng. Notes, 1977**

Exception Handling in CLU, **Liskov, B.H.; Snyder, A.**; **Software Engineering, IEEE Transactions, 1979**

A modular verifiable exception handling mechanism, **Shaula Yemini and Daniel M. Berry. 1985.. ACM Trans. Program. Lang. Syst**

2. Exceptions – Further Reading (C++ and Java)

Exception Handling for C++, **A. R. Koenig and B. Stroustrup: Journal of Object Oriented Programming, 1990**

Analyzing exception flow in Java programs. **Martin P. Robillard and Gail C. Murphy. In Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering. 1999**

Analysis and testing of programs with exception handling constructs , **Sinha, S.; Harrold, M.J.; Software Engineering, IEEE Transactions Sep 2000**

A comparative study of exception handling mechanisms for building dependable object-oriented software, **Alessandro F. Garcia, Cecilia M. F. Rubira, Alexander Romanovsky, Jie Xu, Journal of Systems and Software, Volume 59, Issue 2, 15 November 2001**

Exceptions in Java

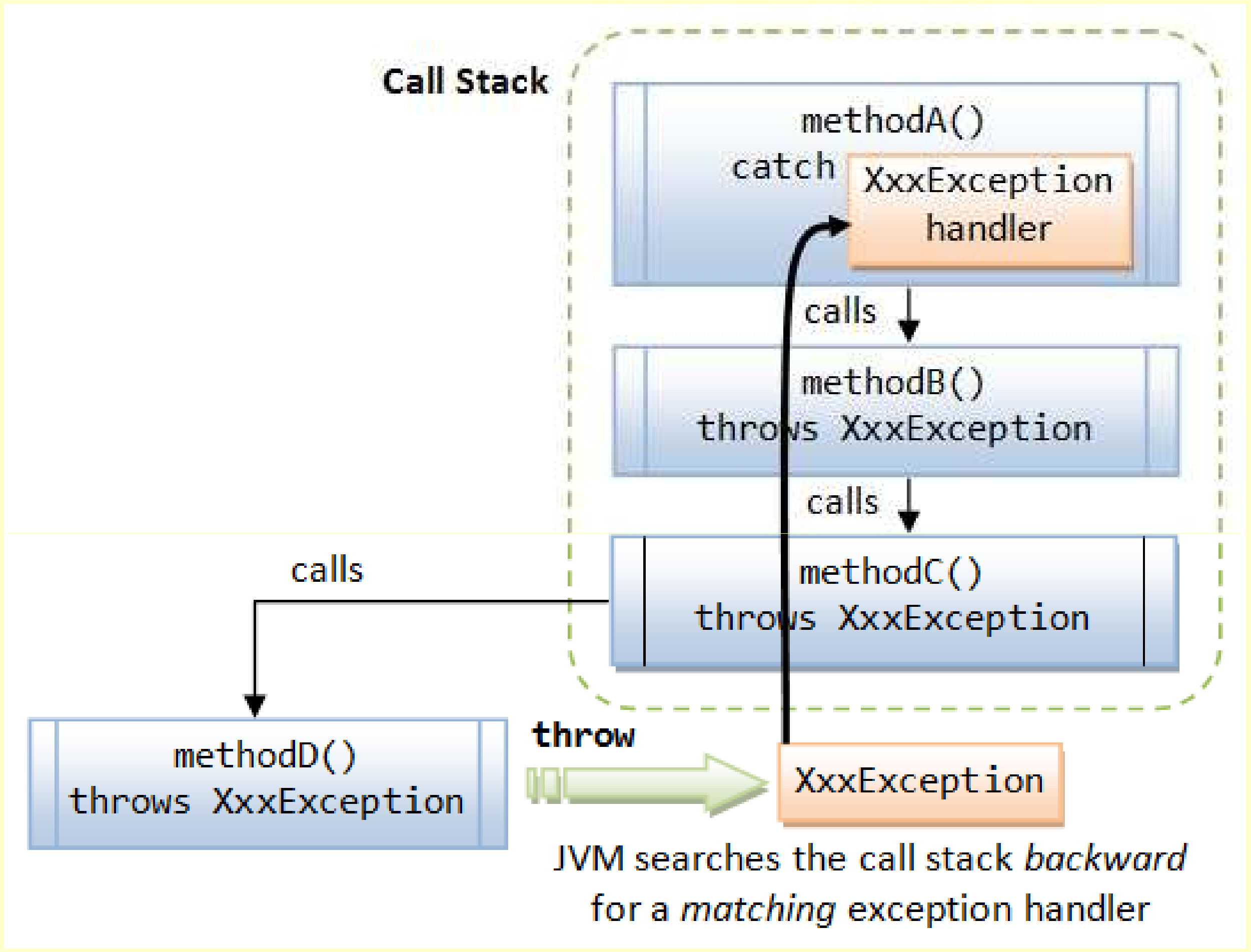
When a method encounters an abnormal condition (an *exception condition*) that it can't handle itself, it may *throw* an exception.

Exceptions are *caught* by handlers positioned along the thread's method invocation stack. If the calling method isn't prepared to catch the exception, it throws the exception up to *its* calling method, and so on.

When you program in Java, you must position catchers (the exception handlers) strategically, so your program will catch and handle all exceptions from which you want your program to recover.

NOTE: If one of the threads of your program throws an exception that isn't caught by any method along the method invocation stack, that thread will expire. (We will come back to this when we look at threads)

Exceptions in Java



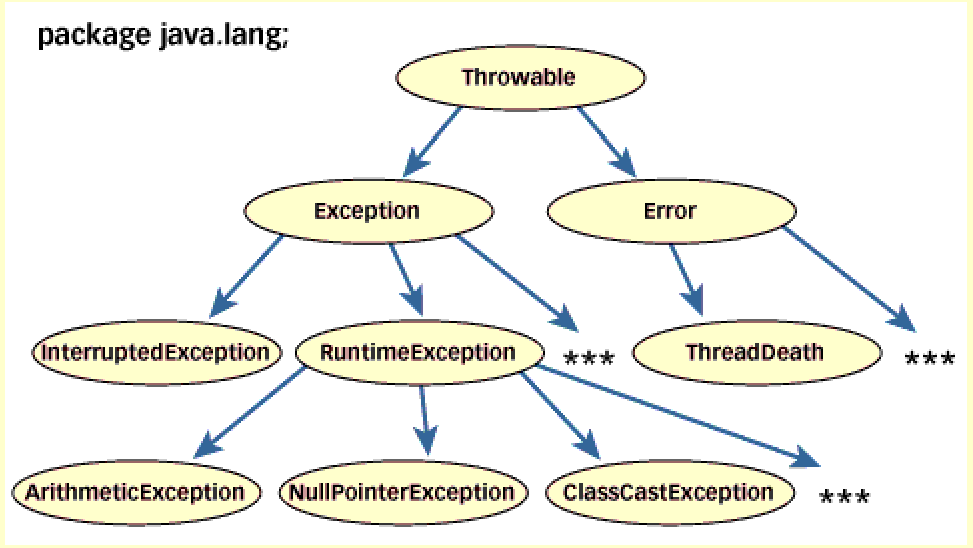
http://www.ntu.edu.sg/home/ehchua/programming/java/J5a_ExceptionAssert.html

Exceptions in Java

In Java, exceptions are objects. When you throw an exception, you throw an object.

You can't throw just any object as an exception, however -- only those objects whose classes descend from Throwable.

Throwable serves as the base class for an entire family of classes, declared in java.lang, that your program can instantiate and throw.



QUESTION: Have you seen Errors?

Exceptions (and errors) in Java

Exceptions (members of the Exception family) are thrown to signal abnormal conditions that can often be handled by some catcher, though it's possible they may not be caught and therefore could result in a dead thread.

Errors (members of the Error family) are usually thrown for more serious problems, such as OutOfMemoryError, that may not be so easy to handle.

In general, code you write should throw only exceptions, not errors. Errors are usually thrown by the methods of the Java API, or by the Java virtual machine itself.

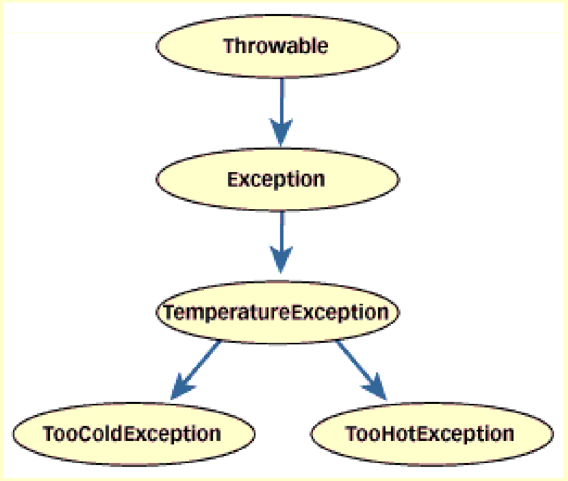
In addition to throwing objects whose classes are declared in java.lang, you can throw objects of your own design. To create your own class of throwable objects, you need only declare it as a subclass of some member of the Throwable family. In general, however, the throwable classes you define should extend class Exception.

Exceptions in Java

Whether you use an existing exception class from java.lang or create one of your own depends upon the situation. In some cases, a class from java.lang will do just fine.

For example, if one of your methods is invoked with an invalid argument, you could throw IllegalArgumentException, a subclass of RuntimeException in java.lang.

Sometimes you will want to indicate that a method encountered an abnormal condition that isn't represented by a class in the Throwable family of java.lang.
For example, in a coffee machine:



NOTE: Exceptional conditions are not necessarily rare, just outside the normal flow of events.

Exceptions in Java: Example Coffee Cup

```
public void drinkCoffee(CoffeeCup cup) throws TooColdException,
TooHotException {

 int temperature = cup.getTemperature();

 if (temperature <= TOOCOLD) throw new TooColdException();
 else if (temperature >= TOOHOT) throw new TooHotException();
 else cup.sip();

}
```

```
try {
cust.drinkCoffee(cup); System.out.println("Coffee is just right.");
}
catch (TooColdException e) { System.out.println("Coffee is too cold."); }
catch (TooHotException e) { System.out.println("Coffee is too hot."); }
 }
```

Exceptions in Java: Example Coffee Cup

You can also group catches:

```
try {
cust.drinkCoffee(cup);
System.out.println("Coffee is just right.");
}
catch (TemperatureException e) {
System.out.println("Coffee is too cold or too hot.");
}
```

QUESTION: What about throwing exceptions inside the catch?
Does Java allow this? What are the semantics/rules?

TO DO: Write some experimental code to find the answers to these questions.

Exceptions in Java: Embedding information in an exception object

When you throw an exception, you are performing a kind of structured go-to from the place in your program where an abnormal condition was detected to a place where it can be handled.

The Java virtual machine uses the class of the exception object you throw to decide which catch clause, if any, should be allowed to handle the exception.

But an exception doesn't just transfer control from one part of your program to another, it also transmits information. Because the exception is a full-fledged object that you can define yourself, you can embed information about the abnormal condition in the object before you throw it. The catch clause can then get the information by querying the exception object directly.

Exceptions in Java: Example Coffee Cup

```
class UnusualTasteException extends Exception {
UnusualTasteException() { }
UnusualTasteException(String msg) { super(msg);}
}

new UnusualTasteException("This coffee tastes like tea.")


try {
      cust.drinkCoffee(cup);
      System.out.println("Coffee ok.");
}
catch (UnusualTasteException e) {
System.out.println( "Customer is complaining of unusual taste.");
String s = e.getMessage();
if (s != null) System.out.println(s);
}
```

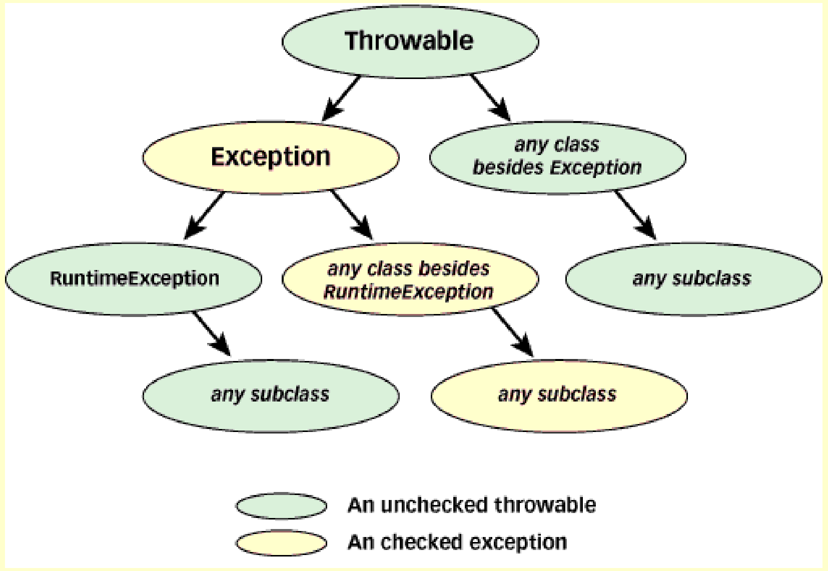
NOTE: here the info passed is a String explaining the strange taste, for TOOHOT or TOOCOLD we could pass the temperature value

Exceptions in Java: Checked vs. unchecked exceptions

There are two kinds of exceptions in Java, *checked* and *unchecked*, and only checked exceptions need appear in throws clauses.

The general rule is: Any checked exceptions that may be thrown in a method must either be caught or declared in the method's throws clause.

Checked exceptions are so called because both the Java compiler and the Java virtual machine check to make sure this rule is obeyed.



Exceptions in Java: finally block

Once a Java virtual machine has begun to execute a block of code -- the statements between two matching curly braces -- it can exit that block in any of several ways.

It could, for example, simply execute past the closing curly brace. It could encounter a break, continue, or return statement that causes it to jump out of the block from somewhere in the middle. Or, if an exception is thrown that isn't caught inside the block, it could exit the block while searching for a catch clause.

Given that a block can be exited in many ways, it is important to be able to ensure that something happens upon exiting a block, no matter how the block is exited. For example, if you open a file in a method, you may want to ensure the file gets closed no matter how the method completes. In Java, you express such a desire with a finally clause.

```
try { // Block of code with multiple exit points }
 finally {
 /* Block of code that must always be executed when the try
block exited, no matter how the try block is exited */
}
```

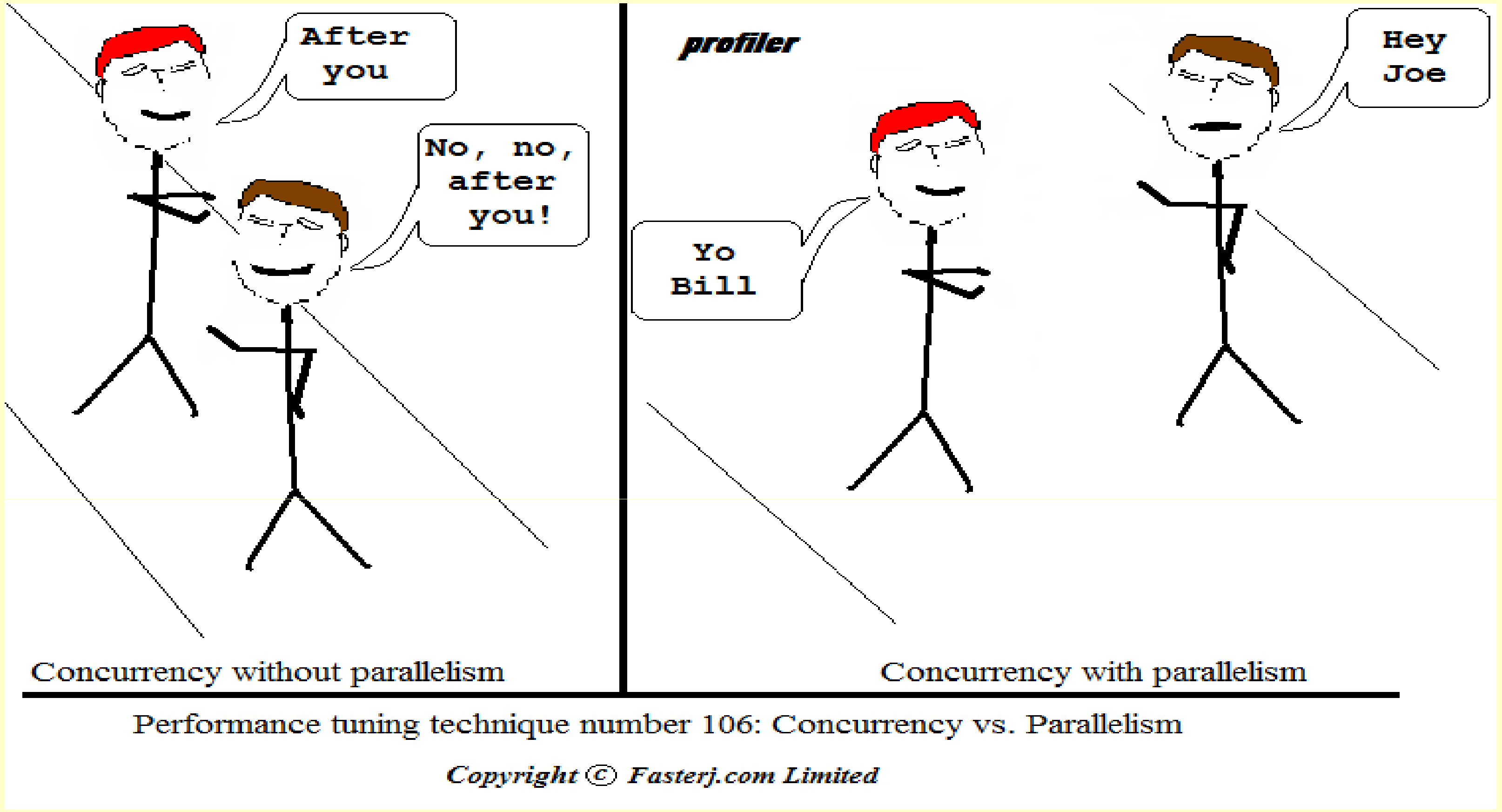
Java: Overriding methods that throw exceptions

TO DO: Write some experimental code to see if you can identify the rules for overiding methods that throw exceptions.

QUESTION: Can the new method (in the subclass)

- Add a new exception?
- Specialise/Generalise an exception thrown by the base class
- Ignore an exception thrown by the base class?

Processes and Threads


After you
No, no, after you!
profiler
Hey Joe
Yo Bill
Concurrency without parallelism
Concurrency with parallelism
Performance tuning technique number 106: Concurrency vs. Parallelism
Copyright © Fasterj.com Limited

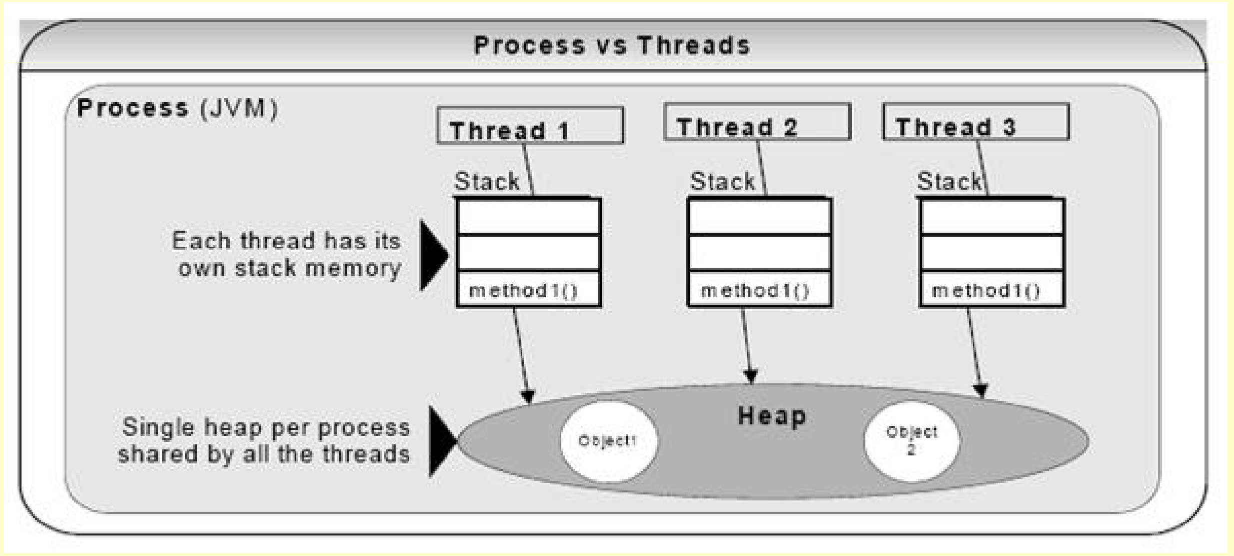
Processes and **Threads** are the two fundamental units of execution in a concurrent program.

Processes and Threads

•In Java, concurrent programming is mostly thread-based.

•Processing time for each core in a system is shared among processes and threads through an OS feature called time slicing.

•Concurrency is possible even on simple systems, without multiple processors or execution cores.



http://www.java-forums.org/blogs/thread/

Processes

Self-contained execution environment.

Independent set of basic run-time resources, such as memory space.

A single application may be implemented by a set of cooperating processes.

Most operating systems support *Inter Process Communication* (IPC) resources.

IPC can also used for communication between processes on different systems.

Most implementations of the JVM run as a single process.

Threads

Also known as *lightweight processes*.

Creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one.

Threads share the process's resources, including memory and open files.

This has advantages and disadvantages … can you think of them?

Multithreaded execution is essential in Java:

- every application has at least one thread
- "system" threads that do memory management, event/signal handling, etc.

In programming, we start with just one thread, called the *main thread*.

Any thread (including the main thread) can create new threads.

Threads in Java: some additional reading

Fixing The Java Memory Model, William Pugh, 1999.

The Problem with Threads, Edward Lee, 2006.

Java Thread Programming, by Paul Hyde ISBN: 0672315858
Sams 1999

Concurrent Programming in Java™: Design, Principles and Patterns, Second Edition, By Doug Lea, ISBN: 0-201-31009-0
Addison Wesley, 2001

Thread Example

```
public class ThreadExample {

    public static void main (String[] args) {
        System.out.println("Starting Thread main");
        new SimpleThread("Add1", '1').start();
        new SimpleThread("Add2", '2').start();
        System.out.println("Finishing Thread main");
    }
}
```

Thread Example - typical output

```
Starting Thread main
Finishing Thread main
String Add2 extended to 2
String Add2 extended to 22
String Add2 extended to 222
String Add1 extended to 1
String Add1 extended to 11
String Add2 extended to 2222
String Add2 extended to 22222
No more increments left for threadAdd2
String Add1 extended to 111
String Add1 extended to 1111
String Add1 extended to 11111
No more increments left for threadAdd1
```

Thread Example - SimpleThread Code

```java
class SimpleThread extends Thread {

// see http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Thread.html

String stringofchars;
char increment;

    public SimpleThread(String str,  char inc) {
        super(str);
        stringofchars = "";
        increment = inc;
    }
    public void run() {
        for (int i = 0; i < 5; i++) {

            try {
                sleep((int)(Math.random() * 3000));
            } catch (InterruptedException e) {}
        stringofchars = stringofchars + increment;
        System.out.println("String " + getName()+
                           " extended to "+ stringofchars );
        }
        System.out.println("No more increments left for
                              thread" + getName());
    }
}
```

`IllegalThreadStateException`

The runtime system throws an `IllegalThreadStateException` when you call a method on a thread and that thread's state does not allow for that method call. (See the state machine diagram in later slides)

So, when you call a thread method that can throw an exception, you must either catch and handle the exception, or specify that the calling method throws the uncaught exception.

The sleep method can also throw an `InterruptedException`, and so we needed a try/catch in the previous code:

```
try {
        sleep((int)(Math.random() * 3000));
      } catch (InterruptedException e) {}
```

Sharing Thread Problem

The previous example showed how two independent threads execute concurrently.

Threads can also share data/objects and so their concurrent behaviours are inter-dependent.

We wish to change the previous code so that the 2 threads update the same string of characters.

We will do this using a SharedString class

Sharing Thread Problem

```
 class SharedString {

public SharedString(){str ="";}

public String str;

public  void add (char c){str = str + c;}

public String toString () {return str;}
}


public class SharingThreads {

    public static void main (String[] args) {

    SharedString soc = new SharedString();
        new SharingThread("SharingAdda", soc, 'a').start();
        new SharingThread("SharingAddb", soc, 'b').start();
    }

}
```

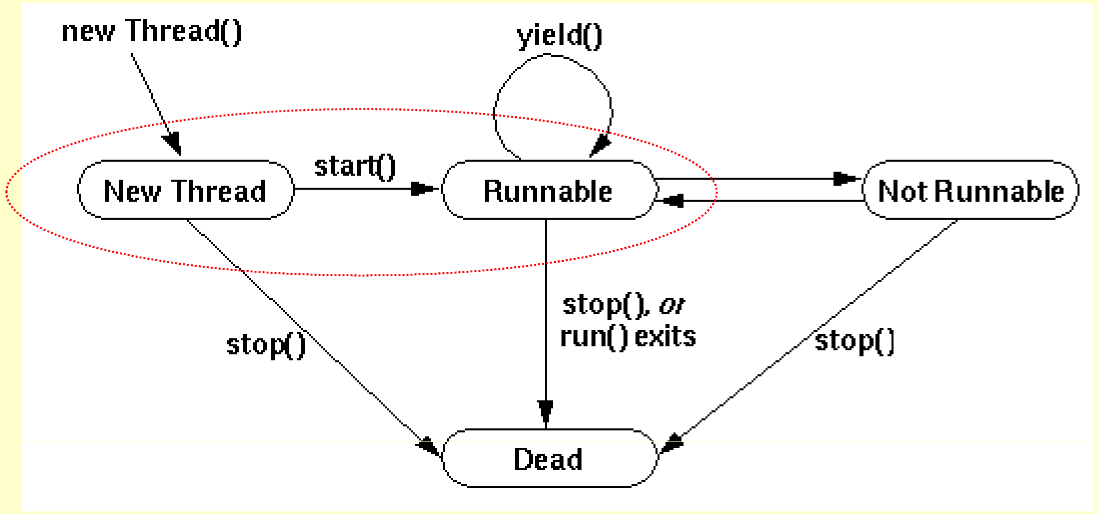
Sharing Thread Problem

We want the output from this code to produce, typically:

```
Shared String extended by SharingAddb to b
Shared String extended by SharingAddb to bb
Shared String extended by SharingAdda to bba
Shared String extended by SharingAddb to bbab
Shared String extended by SharingAddb to bbabb
Shared String extended by SharingAdda to bbabba
Shared String extended by SharingAddb to bbabbab
No more increments left SharingAddb
Shared String extended by SharingAdda to bbabbaba
Shared String extended by SharingAdda to bbabbabaa
Shared String extended by SharingAdda to bbabbabaaa
No more increments left SharingAdda
```

TO DO: Your task is to code the **`class`** `SharingThread` **`extends`** `Thread {}` to provide this behaviour

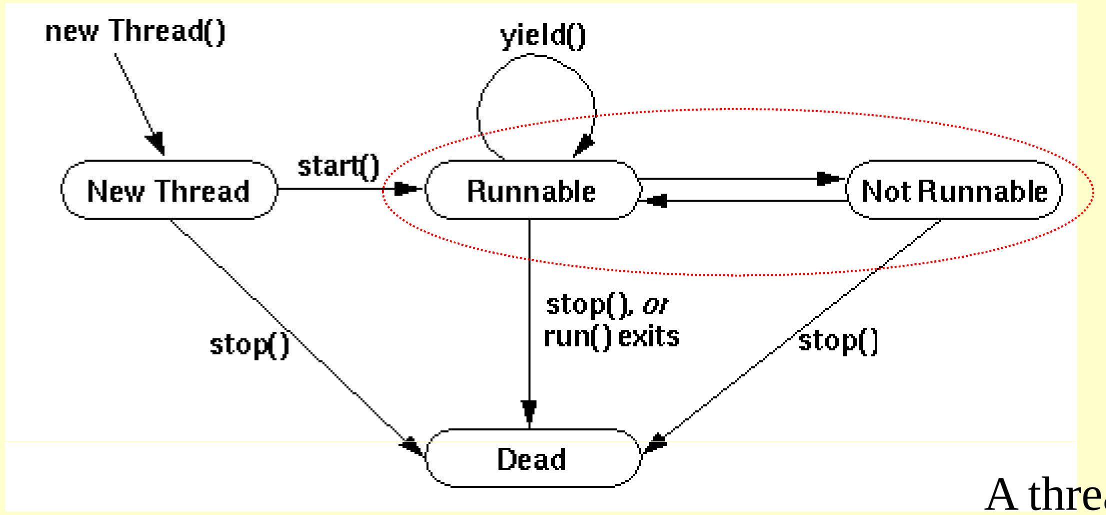
Thread State Machine: an abstraction of the complete diagram



The `start()` method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's `run()` method.

The next state state is "`Runnable`" rather than "`Running`" because the thread might not actually be running when it is in this state.

Thread State Machine: an abstraction of the complete diagram



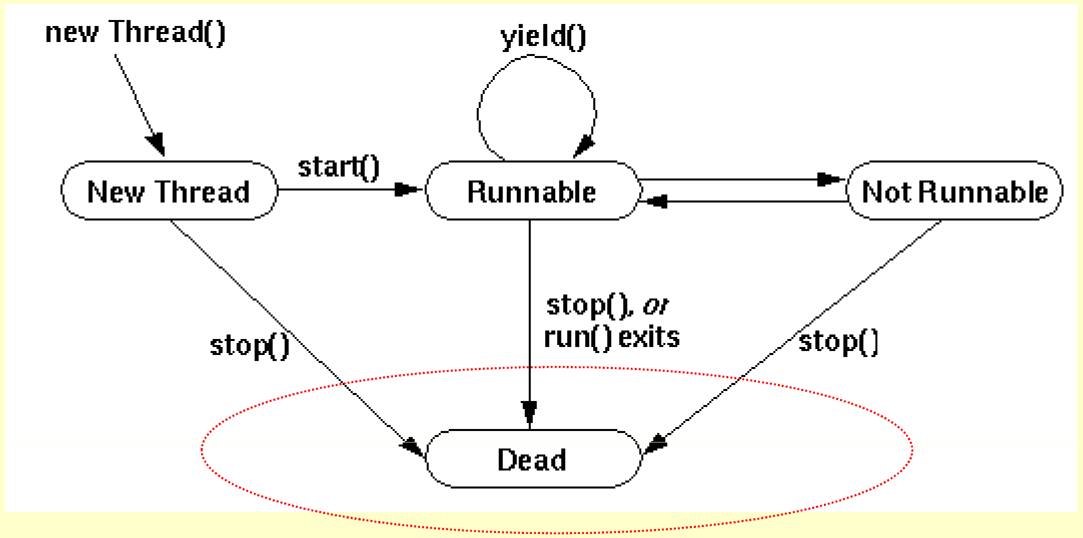
A thread enters the "`Not Runnable`" state when:

- `sleep()` is called.
- `suspend()` is called.
- The thread uses its `wait()` method to wait on a condition variable.
- The thread is blocking on I/O.

A thread leaves the "`Not Runnable`" state when a matching condition is met:

- `sleep()` is completed.
- `resume()` is called
- object owning the variable calls `notify()` or `notifyAll()`
- I/O completes

Thread State Machine: an abstraction of the complete diagram



A thread dies naturally when its `run()` method exits normally

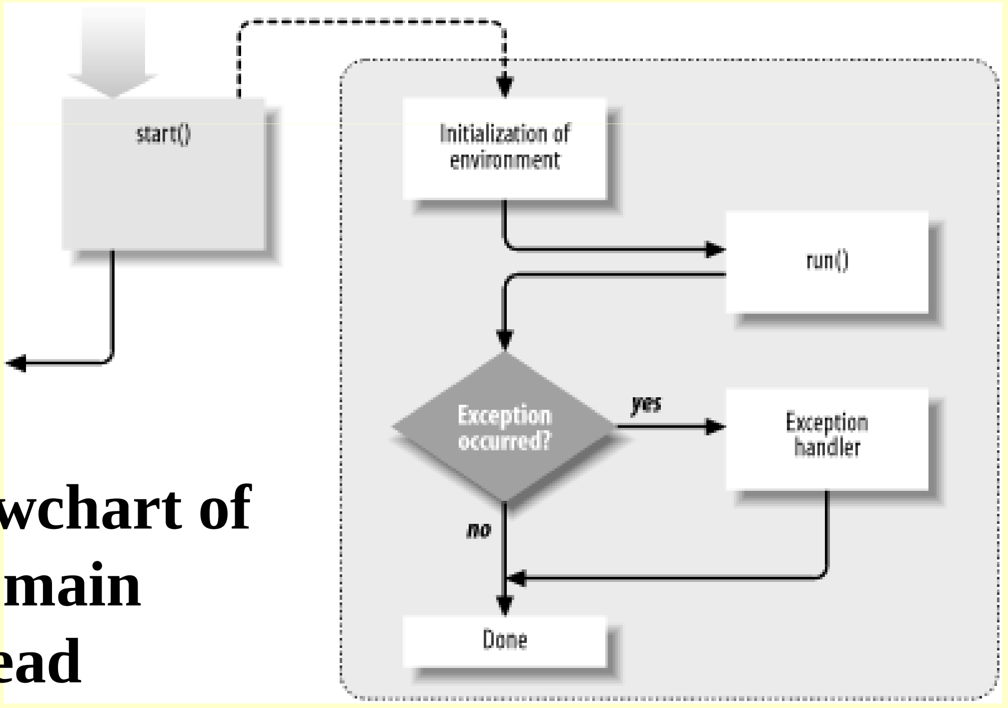
You can also kill a thread at any time by calling its `stop()` method

QUESTION: What should happen if an exception occurs inside a thread?

Threads and exceptions: an aside

The start() method does start another thread of control, but the run() method is not really the "main" method of the new thread.

The run() method is executed inside a context that allows the virtual machine to handle runtime exceptions thrown from the run() method.



Flowchart of the main thread

Threads and exceptions: an aside

All uncaught exceptions are handled by code outside of the run() method before the thread terminates. The default exception handler is a Java method; it can be overridden. This means that it is possible for a program to write a new default exception handler.

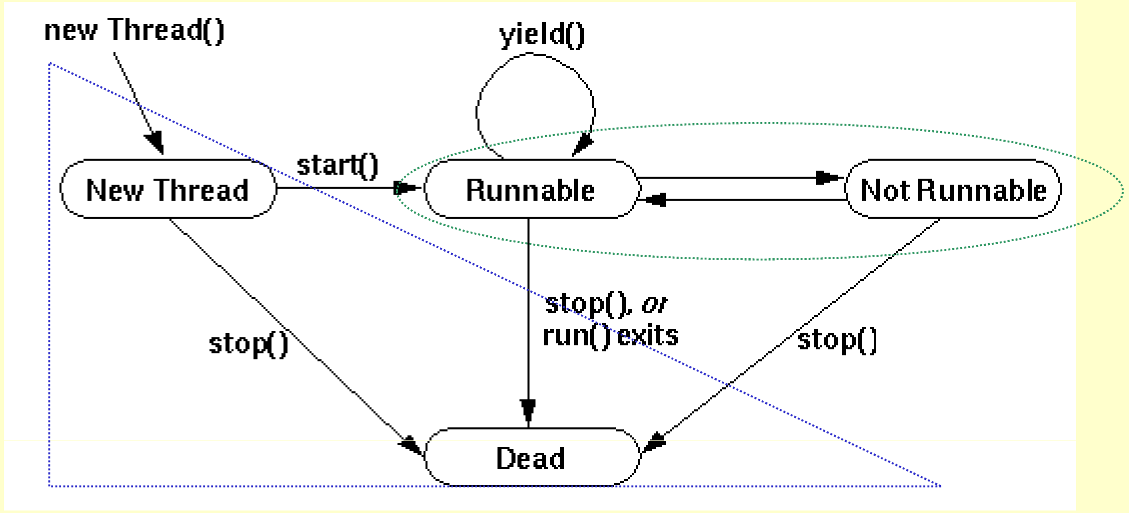
The default exception handler is the uncaughtException() method of the ThreadGroup class. It is called only when an exception is thrown from the run() method. The thread is technically completed when the run() method returns, even though the exception handler is still running the thread.

The default implementation of the uncaughtException() method is to print out the stack trace of the Throwable object thrown by the run() method

In most cases, this is sufficient: the only exceptions that the run() method can throw are runtime exceptions or errors. By the time the run() method has returned, it's too late to recover from these errors.

One case in which it's useful to override the uncaughtException() method is to send a priority notification to an administrator that an unusual, fatal error has occurred. Here's an example that does that when its thread eventually encounters an out-of-memory error:

Thread State Machine: an abstraction of the complete diagram



The `isAlive()` method returns true if the thread has been started and not stopped.

`thread.isAlive() = false` => `thread` is either a "`New Thread`" or "`Dead`".

`thread.isAlive() = true` => `thread` is either a "`Runnable`" or "`Not Runnable`".

Threads and Synchronization Issues

Threads can share state (objects)

This is very powerful, and makes for very efficient inter-thread communication

However, it makes two kinds of errors possible:

- *thread interference*, and
- *memory inconsistency*.

Java provides a *synchronization "tool"* in order to avoid these types of errors.

Thread Interference

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap. Because they are unpredictable, thread interference bugs can be difficult to detect and fix.

Consider a simple class called Counter

```
class Counter {
    private int c = 0;
            public void increment() {c++;}

   /*             Multiple steps of c++
             1.  Retrieve the current value of c.
            2.  Increment the retrieved value by 1.
           3.  Store the incremented value back in c.
   */

    public void decrement() {c--;}
    public int value() {return c;}
}
```

If a Counter object is referenced from multiple threads, interference between threads may give rise to unexpected behaviour.

Memory inconsistency

Consider the following example.

```
        int counter = 0;
```

The counter field is shared between two threads, A and B.

Suppose thread A increments counter:

```
            counter++;
```

Then, shortly afterwards, thread B prints out counter:

```
            System.out.println(counter);
```

If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "`1`".

But, in this example, the value printed out might well be "0", because there's no guarantee that thread A's change to counter will be visible to thread B — unless the programmer has established a ***happens-before*** relationship between these two statements.

There are several actions that create ***happens-before*** relationships.

The simplest technique/tool is to use ***`synchronization`***

Synchronized methods, example:

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {c++;}
    public synchronized void decrement() {c--;}
    public synchronized int value() {return c;}
}
```

Two invocations of synchronized methods on the same object cannot interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

When a synchronized method exits, it automatically establishes a *happens-before* relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Constructors cannot be synchronized

Sychronization is effective for keeping systems *safe*, but can present problems with *liveness*