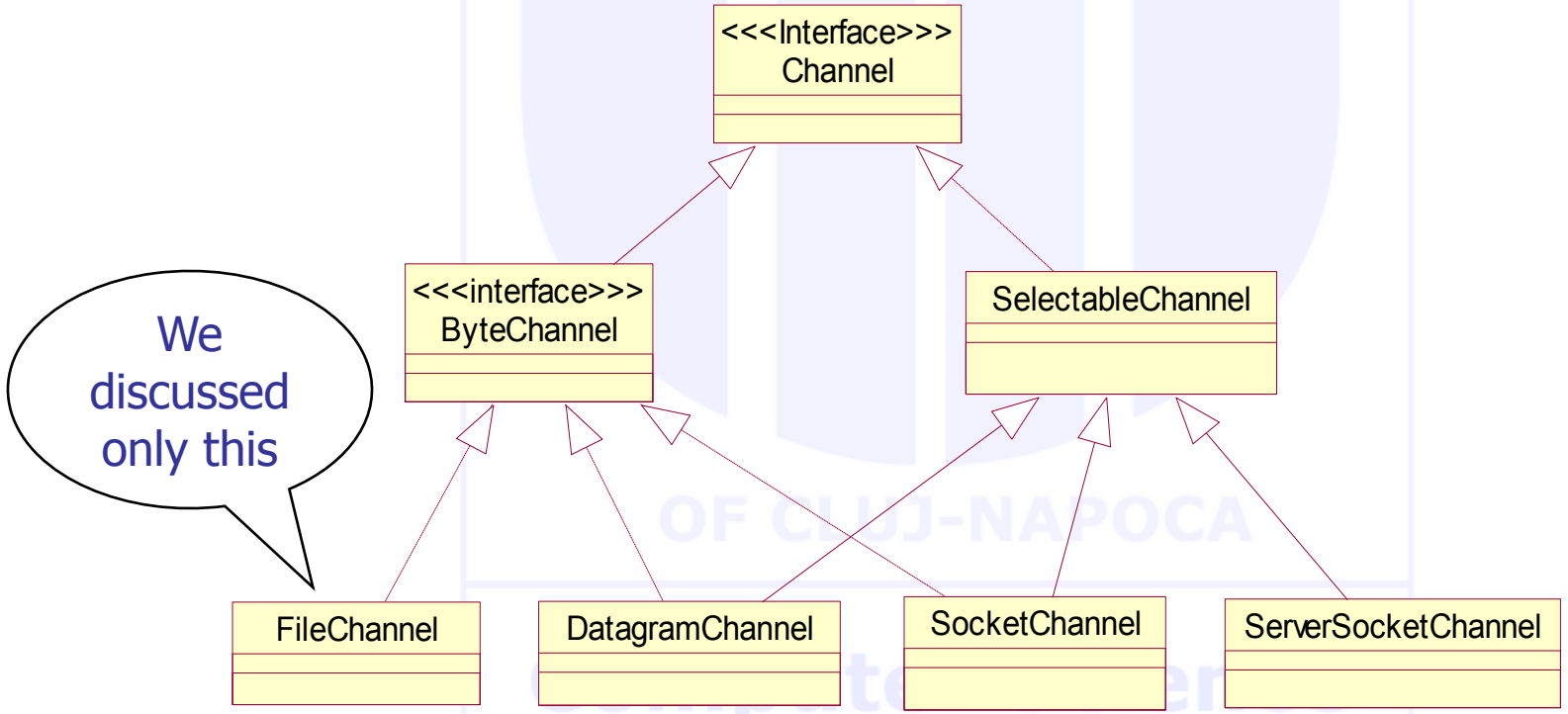
Object Oriented Programming

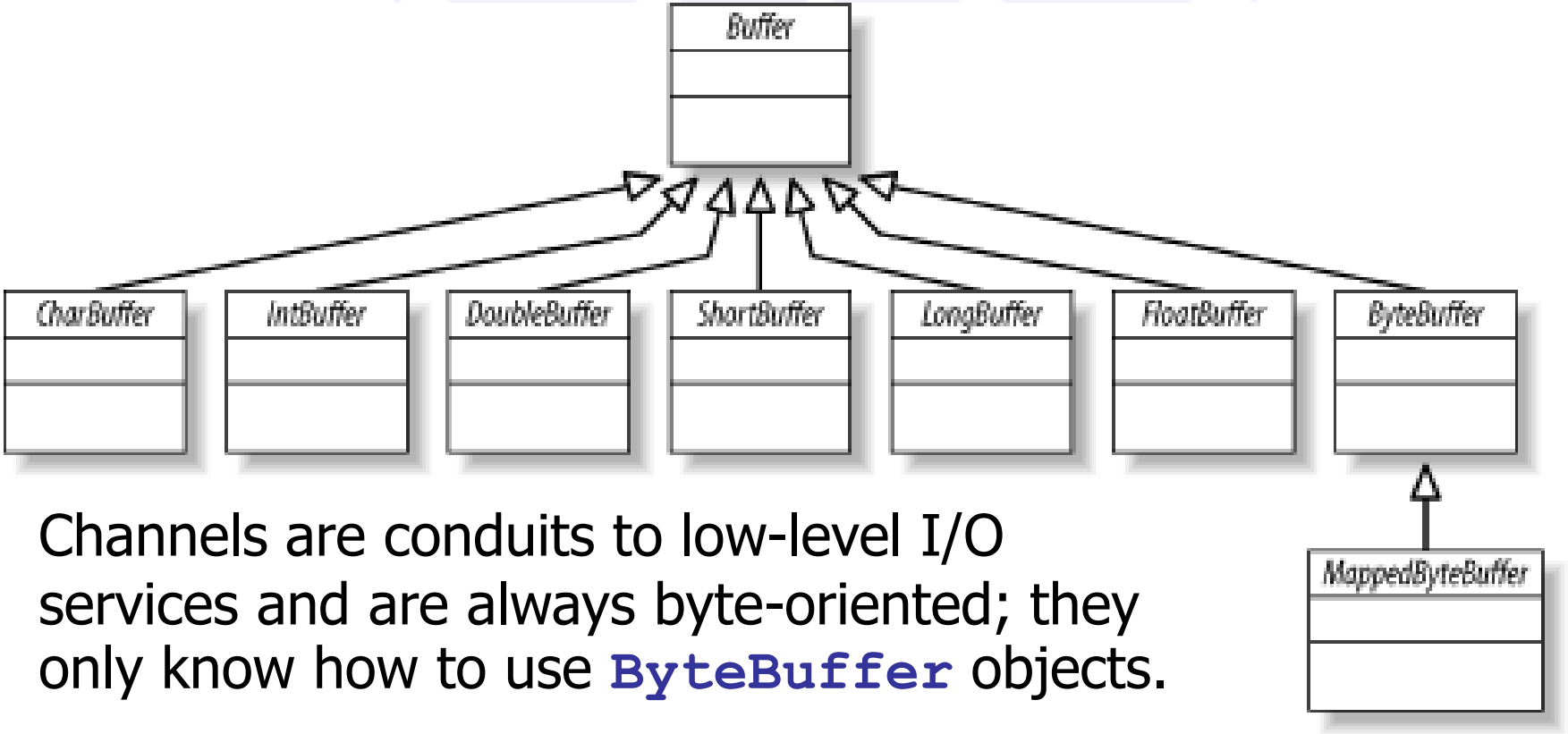
1. More Java new I/O
2. Introduction to Threads

Simplified `Channel` Hierarchy



Buffers

- Buffers were created primarily to act as containers for data being sent to or received from channels.



- Channels are conduits to low-level I/O services and are always byte-oriented; they only know how to use **`ByteBuffer`** objects.

Buffer Views

- Assume we have a file containing Unicode characters stored as 16-bit values (UTF-16 not UTF-8 encoding. UTF = Unicode Transformation Format)
- To read a chunk of this file into the byte buffer, we could then create a `CharBuffer` view of those bytes:

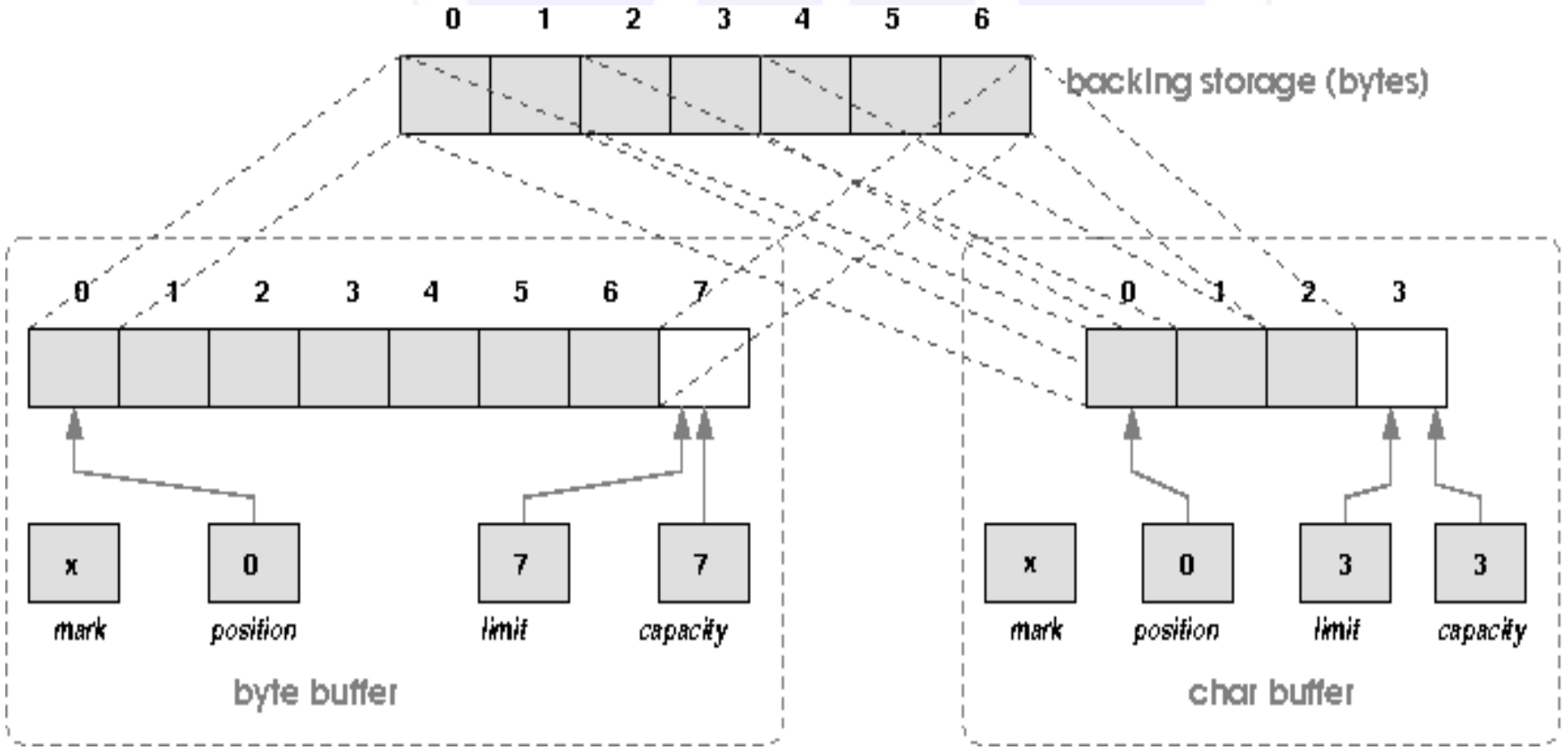
```
CharBuffer charBuffer = byteBuffer.asCharBuffer();
```

- This creates a view of the original `ByteBuffer`, which behaves like a `CharBuffer` (combines each pair of bytes in the buffer into a 16-bit char value)
- The `ByteBuffer` class also has methods to do ad hoc accesses of individual primitive values. For example, to access four bytes of a buffer as an `int`, you could do the following:

```
int fileSize = byteBuffer.getInt();
```

Buffer Views

- **`CharBuffer`** view of a **`ByteBuffer`**



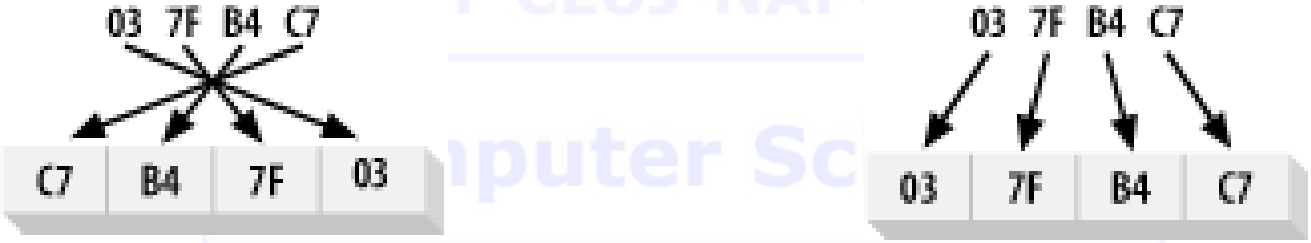
Buffer Views Example

```
import java.nio.*;
public class Buffers {
  public static void main(String[] args)  {
    try {
      float[] floats = {
         6.612297E-39F, 9.918385E-39F,
         1.1093785E-38F, 1.092858E-38F,
         1.0469398E-38F, 9.183596E-39F
      }
      ByteBuffer bb =
        ByteBuffer.allocate(floats.length * 4 );
      FloatBuffer fb = bb.asFloatBuffer();
      fb.put(floats);
      CharBuffer cb = bb.asCharBuffer();
      System.out.println(cb.toString());
    } catch (Exception e) {
      System.out.println(e.getMessage();
      e.printStackTrace();
    }
  }
}
```

FloatBuffer	6.612297E-39		9.918385E-39		1.0193785E-38		1.092858E-38		1.0469398E-38		9.183596E-39	
CharBuffer	H	e	l	l	o		w	o	r	l	d	!
ByteBuffer	0048	0065	006C	006C	006F	0020	0077	006F	0072	006C	0064	0021

Byte Swabbing

- *Endian-ness* : the order in which bytes are combined to form larger numeric values.
  - When the numerically-most-significant byte is stored first in memory (at the lower address), this is *big-endian* byte order
  - The opposite, where the least significant byte occurs first, is *little-endian*



little endian

big endian

Buffer Views and Endian-ness

- Every buffer object has a *byte order* setting.
  - For all but `ByteBuffer`, this is a read-only property and cannot be changed.
- The byte order setting of `ByteBuffer` objects can be changed at any time.
  - This affects the resulting byte order of any views created of that `ByteBuffer` object.
  - If the Unicode data in our file was encoded as UTF-16LE (little-endian), we'd set the `ByteBuffer`'s byte order prior to creating the view `CharBuffer`:

```
byteBuffer.order (ByteOrder.LITTLE_ENDIAN);
CharBuffer charBuffer = byteBuffer.asCharBuffer();
```

- The new view buffer inherits the byte order setting of the `ByteBuffer`
- The buffer's byte order setting at the time of the call *affects* how bytes are combined to form the return value or broken out for storage in the buffer.

Direct Buffers

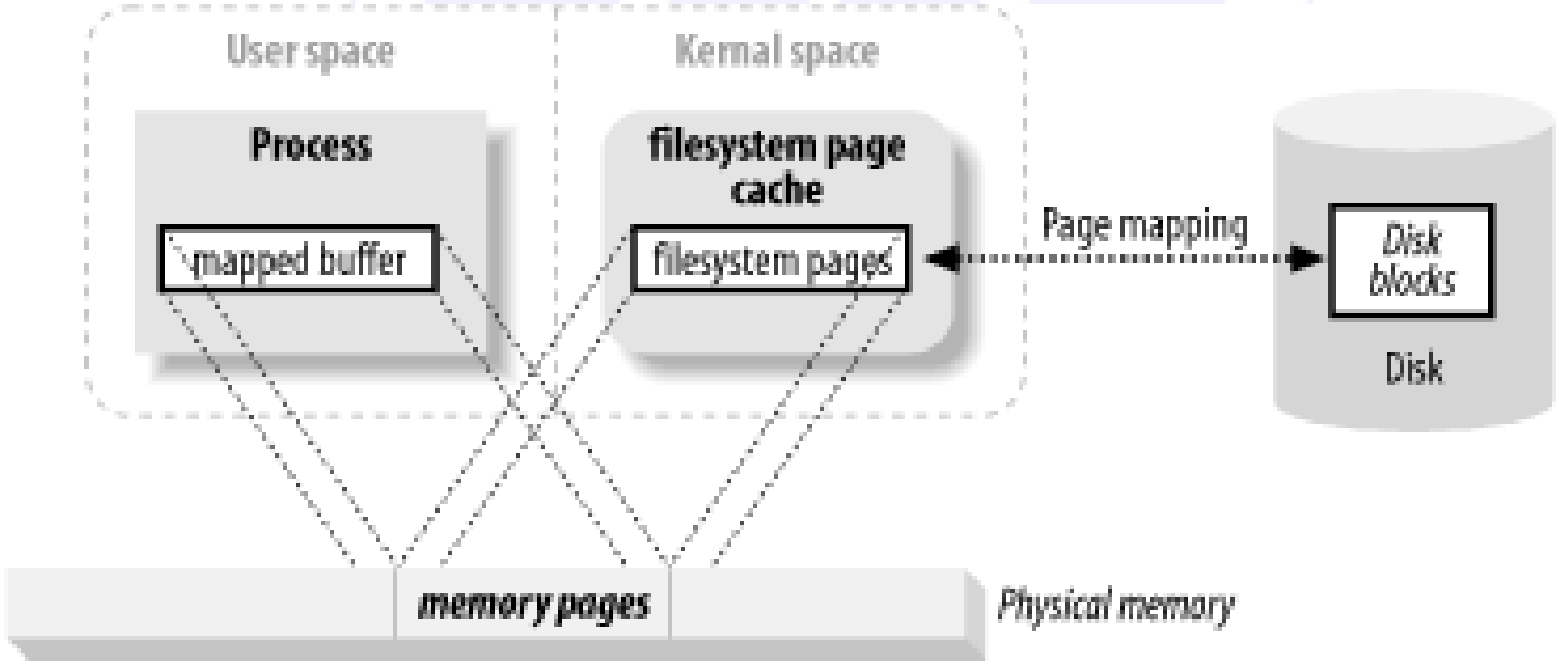
- The data elements encapsulated by a buffer can be stored in one of several different ways:
  - in a private array created by the buffer object (allocation),
  - in an array you provide (wrapping), or,
  - in the case of *direct* buffers, in *native memory* space outside of the JVM's memory heap.
- When we create a direct buffer (by invoking `ByteBuffer.allocateDirect()`), native system memory is allocated and a buffer object is wrapped around it.
- The primary purpose of direct buffers is for doing I/O on channels.
- Channel implementations can set up OS-level I/O operations to act directly upon a direct buffer's native memory space.

Memory Mapped Files

- **`MappedByteBuffer`** is a specialized form of **`ByteBuffer`**.
  - On most operating systems, it's possible to *memory map* a file using the **`mmap()`** *system call* (or something similar – note this does not belong to Java) on an open file descriptor.
    - Calling mmap() returns a pointer to a memory segment, which actually represents the content of the file.
  - Fetches from memory locations within that memory area will return data from the file at the corresponding offset.
  - Modifications made to the memory space are written to the file on disk.

Memory Mapped Files

- Additional processes running in user space would map to that same physical memory space, through the same filesystem cache and thence to the same file data on disk.
- Each of those processes would see changes made by any other.
- This can be exploited as a form of persistent, shared memory.



`MappedByteBuffer` Example: Reversing Bytes in a File

```
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
public class MappedBufferDemo {
   public static void main(String
    args[])  throws IOException {
      // check command-line argument
      if (args.length != 1) {
         System.err.println(
            "missing file argument");
         System.exit(1);
      }
      // get channel
      RandomAccessFile raf =
            new RandomAccessFile(
                  args[0], "rw");
      FileChannel fc = raf.getChannel();
```

```
// map file to buffer
      MappedByteBuffer mbb =
     fc.map(FileChannel.MapMode.READ_W
     RITE, 0,  fc.size());
// reverse bytes of file
      int len = (int)fc.size();
      for ( int i = 0, j = len - 1;
                  i < j;
                   i++, j--)
     {
         byte b = mbb.get(i);
         mbb.put(i, mbb.get(j));
         mbb.put(j, b);
      }
      // finish up
      fc.close();
      raf.close();
   }
}
```

Scattering Reads and Gathering Writes

- E.g. With a single read request to the channel we can place the first 32 bytes into the **`header`** buffer, the next 768 bytes into the **`colorMap`** buffer, and the remainder into **`imageBody`**
- The channel fills each buffer in turn until all are full or there're no more data to read

```
. . .
ByteBuffer header = ByteBuffer.allocate (32);
ByteBuffer colorMap = ByteBuffer (256 * 3);
ByteBuffer imageBody = ByteBuffer (640 * 480);
ByteBuffer [] scatterBuffers = { header, colorMap,
  imageBody };
fileChannel.read (scatterBuffers);
```

Direct Channel Transfers

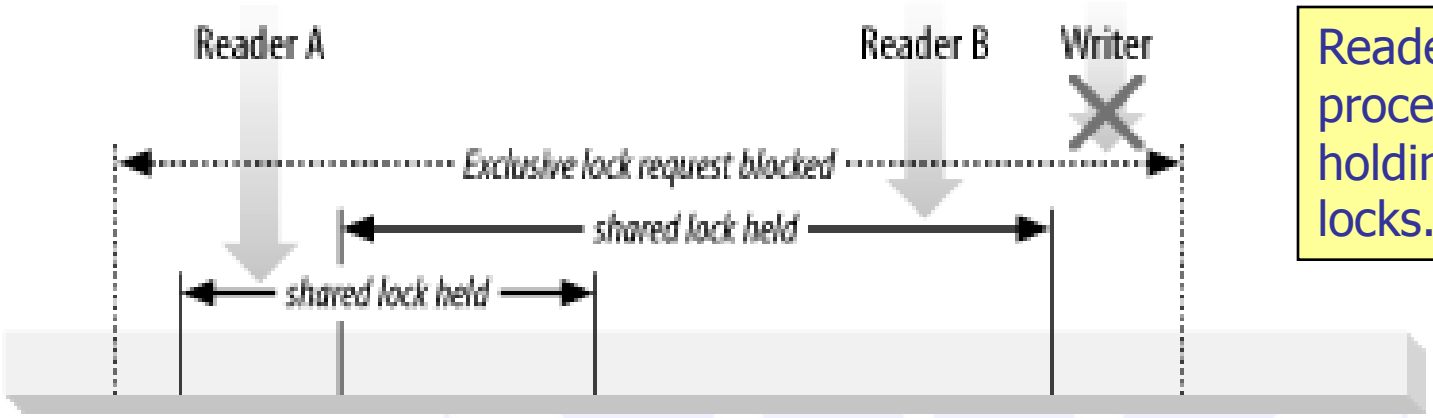
- A channel transfer lets you cross-connect two channels so that data is transferred directly from one to the other without any further intervention on your part.
- Because the `transferTo()` and `transferFrom()` methods belong to the `FileChannel` class, a `FileChannel` object must be the source or destination of a channel transfer (you can't transfer from one socket to another, for example).
- The other end may be any `ReadableByteChannel` or `WritableByteChannel`, as appropriate.
- Note: demos in nio subdirectory

Direct Channel Transfers Example

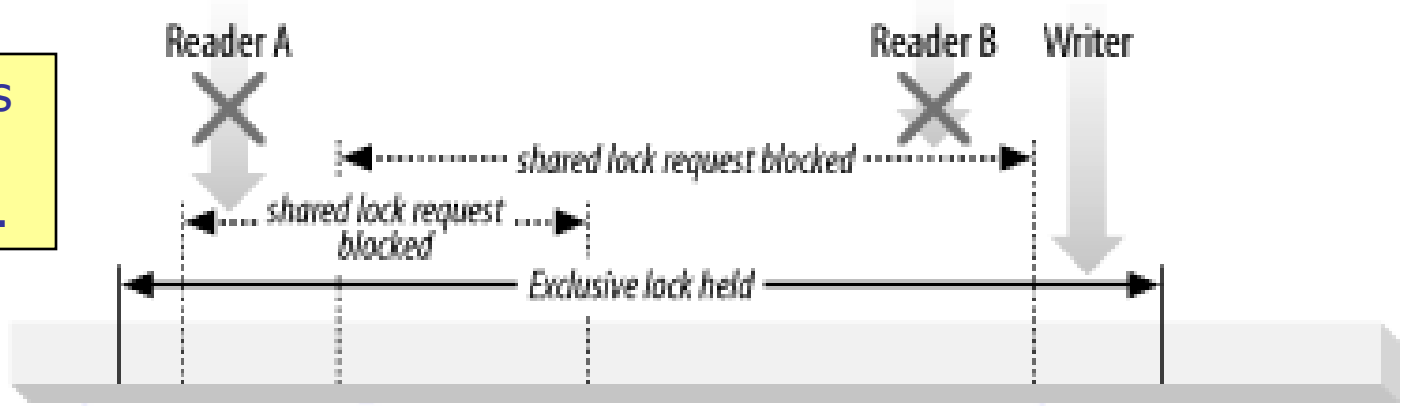
```
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
public class ChannelDemo {
  public static void main(String args[])  throws IOException {
    // check command-line arguments
    if (args.length != 2) {
      System.err.println("missing filenames");
      System.exit(1);
    }
    // get channels
    FileInputStream fis =  new FileInputStream(args[0]);
    FileOutputStream fos =  new FileOutputStream(args[1]);
    FileChannel fcin = fis.getChannel();
    FileChannel fcout = fos.getChannel();
    // do the file copy
     fcin.transferTo(0, fcin.size(), fcout);
    // finish up
    fcin.close();
    fcout.close();
    fis.close();
    fos.close();
  }
}
```

transferTo method transfers bytes from the source channel (**fcin**) to the specified target channel (**fcout**). The transfer is typically done *without explicit user-level reads and writes* of the channel.

File Locking



Reader processes holding shared locks.



Writer process holding exclusive lock.

- File locks are generally needed when integrating with non-Java applications, to mediate access to shared data files

Regular Expressions

- Regular expressions (`java.util.regex`) are part of NIO
- `String` class is regex-aware by adding the following methods:

```
package java.lang;
public final class String implements java.io.Serializable,
  Comparable, CharSequence
{
  // This is a partial API listing
  public boolean matches (String regex)
  public String [] split (String regex)
  public String [] split (String regex, int limit)
  public String replaceFirst (String regex, String
  replacement)
  public String replaceAll (String regex, String
  replacement)
}
```

Regex Examples

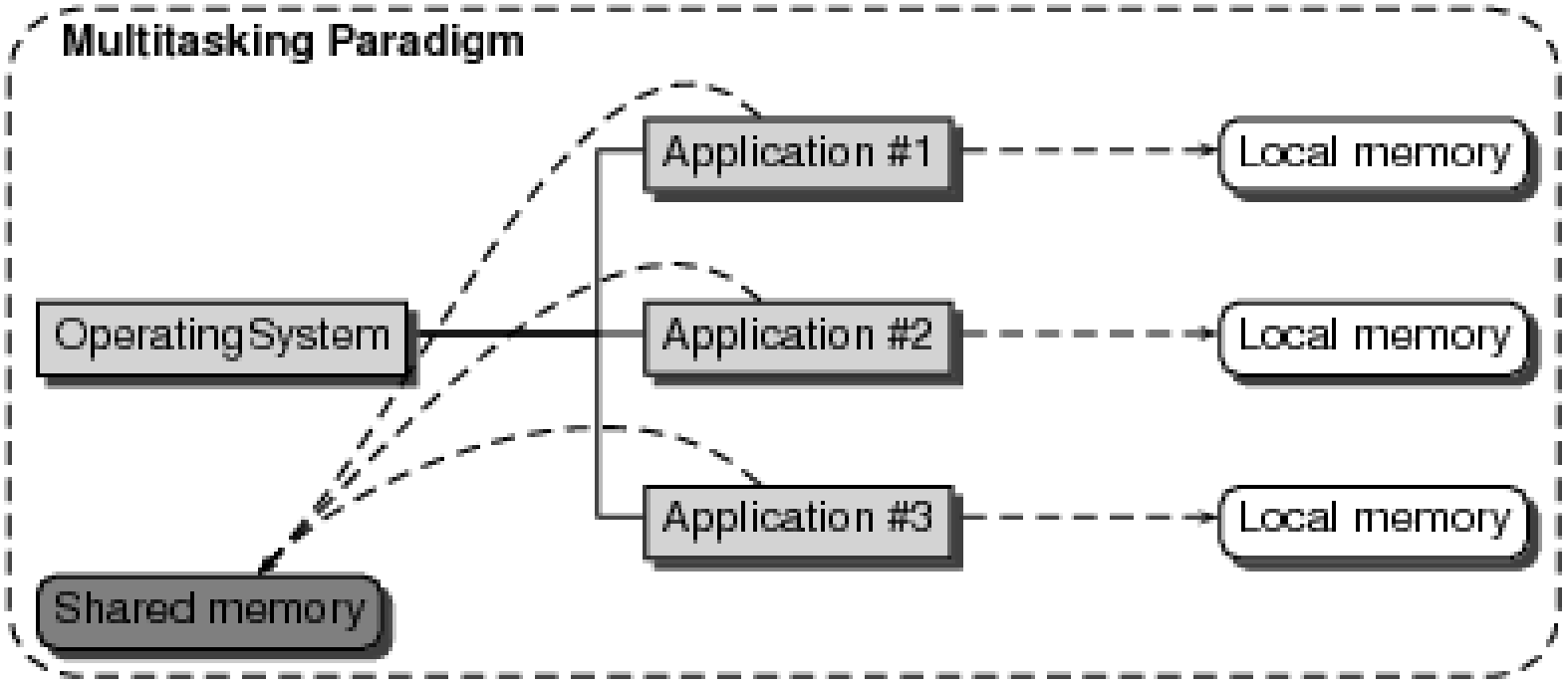
```
public static final String VALID_EMAIL_PATTERN = "([a-zA-Z0-9_\\-
    \\.]+)@((\\[[0-9]{1,3}\\.[0-9]" + "{1,3}\\.[0-9]{1,3}\\.)|(([a-zA-
    Z0-9\\-]+\\.)+))" + "([a-zA-Z]{2,4}|[0-9]{1,3})(\\]?)";
...
if (emailAddress.matches (VALID_EMAIL_PATTERN))
{
  addEmailAddress (emailAddress);
}
else
{
  throw new IllegalArgumentException (emailAddress);
}

// splits the string lineBuffer (which contains a series of comma-separated
    values) into substrings and returns those strings in a type-safe array
String [] tokens = lineBuffer.split ("\\s*,\\s*");
```

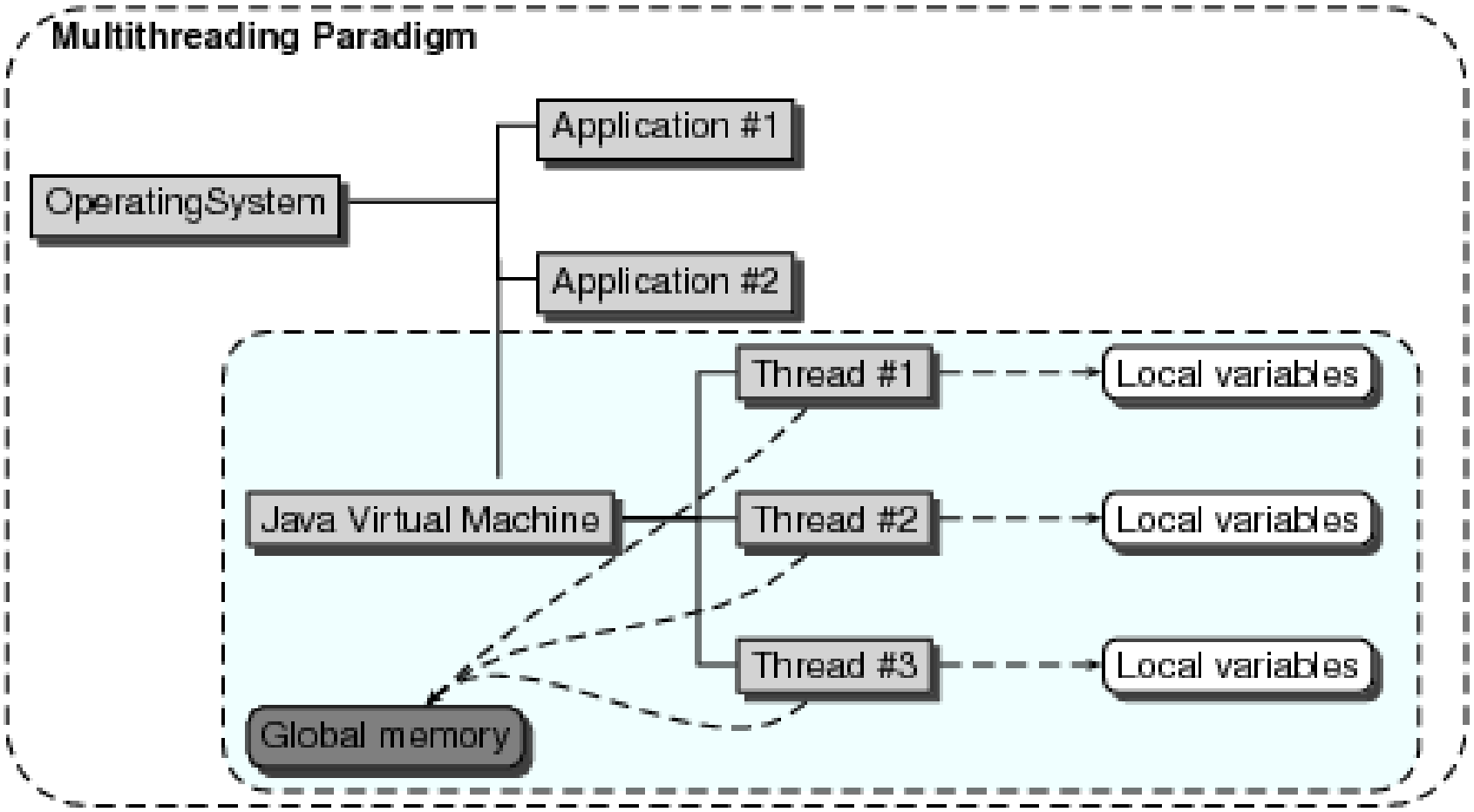
Introduction to Java Threads

Multitasking

- At SO level, a number of processes appear to execute simultaneously



Multitasking vs Multithreading



Thread Tasks

- Threads are useful in many ways:
  - Where several things must happen at once multi-threading has great appeal.
    - For example, a multimedia application can require audio, video, and control processes to run in parallel. There are often periods of waiting on slow IO systems to respond when the processor could be doing other tasks.
  - Programs such as server/client systems are much easier to design and write with threads.
  - Mathematical algorithms such as sorting, prime searching, etc. are suitable for parallel processing.
  - On multiprocessor systems, the JVMs can put threads on different processors and thus obtain true parallel processing and get significant speedups in performance over single processor platforms.

Multithreading in Java

- Properties of multiple threads within Java programs:
  - Each thread begin execution at a well-known, predefined location
  - Each thread executes its code from its starting location in an ordered, predefined (for a given set of inputs) sequence
  - Each thread executes its code independent of the other threads in the program
  - Threads appear to have a certain amount of simultaneous execution
  - Threads have access to various types of data

Multithreading in Java

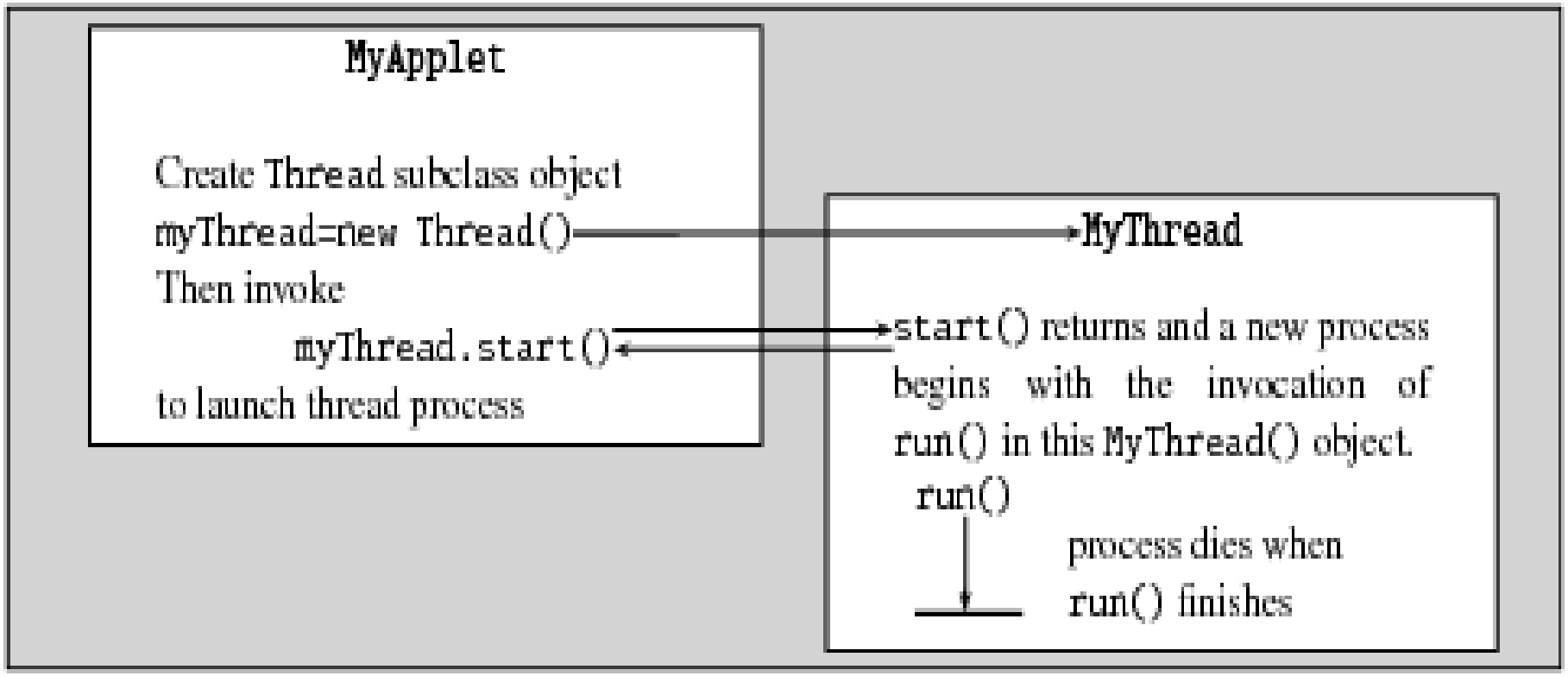
- All Java programs other than simple console-based applications are multithreaded, whether you like it or not.
- *Heavyweight* processes run in the local machine system.
- Thread: a single sequential flow of control within a program (also called *lightweight process*).
  - parallel processes running *inside* of a program
  - in Java you can create one or more threads within your program just as you can run one or more programs in an operating system
- Java: create threads in two ways:
  - A class extends the **`Thread`** class and overrides it's **`run()`** method.
  - A class implements the **`Runnable`** interface, which has one method: **`run()`**.
    - The class passes a reference to itself when it creates a thread.
    - The thread then calls back to the **`run()`** method in the class.

`Thread` Subclass

- The `run()` method in the thread corresponds to the `main()` method for an application.
- When the thread starts, it invokes the method `run()` and when the process returns from the `run()` method, the thread *dies*.
  - You cannot resurrect a dead thread. You must instead create a new thread instance.
- The subclass must override the `run()` method
- In `run()` you put the code you wish to process in parallel to the main program

Thread Subclass Example

- Demo: subclass

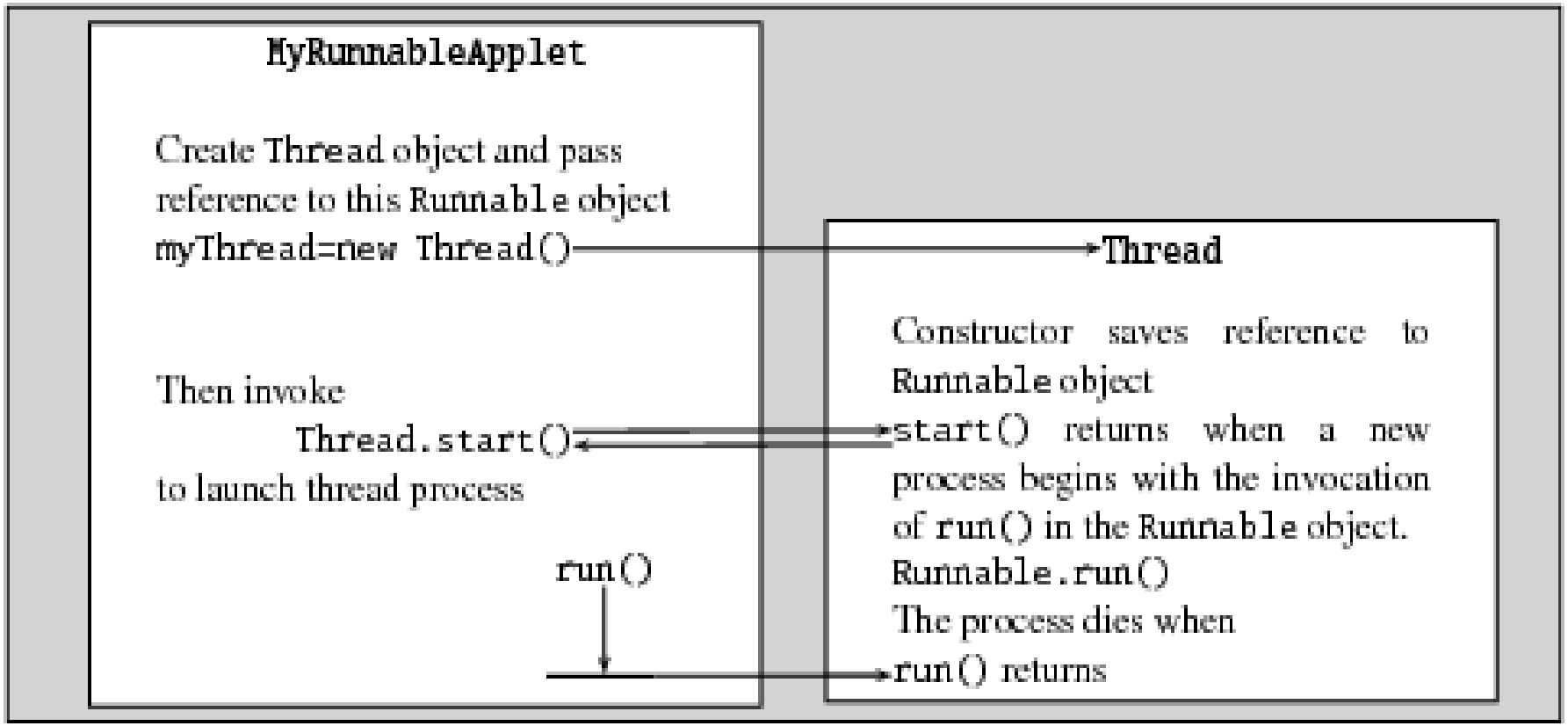


`Runnable` Implementation

- Implement the `Runnable` interface and override its `run()` method
- Pass a reference to an instance of that `Runnable` implementation to a thread instance and the thread *calls back* to the `run()` method in the `Runnable` object.
- The thread dies as in the previous method when the process returns from `run()`.
- Convenient for cases where you want to create a single type of thread, such as an *animation in an applet*.

`Runnable` Implementation Example

- Demo: runnable



The `Runnable` Interface: Suggested Implementation Outline

```
public class ClassToRun extends SomeClass implements
   Runnable
{ . . .
  public void run()
  {
    // Fill this as if ClassToRun
    // were derived from Thread
  }
  . . .
  public void startThread()
  {
    Thread theThread = new Thread(this);
    theThread.run();
  }
  . . .
}
```

Subclass vs. `Runnable`

- **`Runnable`** technique is particularly convenient when you want to create just a single thread to carry out a specific task.
  - The **`run()`** method will have access to the instance variables in the **`Runnable`** object.
  - E.g., for applet animation make the applet **`Runnable`**.
    - The **`run()`** method then has access to the applet's variables, which could be parameters passed in the applet tags or set by the user via the graphical interface
- If you want to create multiple threads, then it usually makes more sense to use a **`Thread`** subclass.
  - Helps to better conceptualize the threads as independent objects
  - You can set the values of whatever parameters they need via their constructors or "setter" methods

Stopping/Pausing a Thread

- A thread stops in three ways:
  - It returns smoothly from `run()`. [Best way]
  - The thread's `stop()` method is called. (Now deprecated. Don't use this.)
  - Interrupted by an uncaught exception.
- If the `run()` method contains a long or endlessly running loop – stop the looping when a variable that can be changed by the main process changes. E.g. `boolean` value set to `false` or a reference variable set to `null` to stop
- Always explicitly stop your threads in applets when the applet `stop()` is called.
  - Otherwise, the threads may continue running even when the browser loads a new web page

Stopping/Pausing a Thread

- For the **`Runnable`** case, you can start a new thread with the variable settings still at the values they had at the point the previous thread stopped.
  - In this case, stopping a thread and starting a new one acts just like *pause/start* actions.
- The **`Thread`** class *suspend()* and *resume()* methods were deprecated to avoid deadlock problems

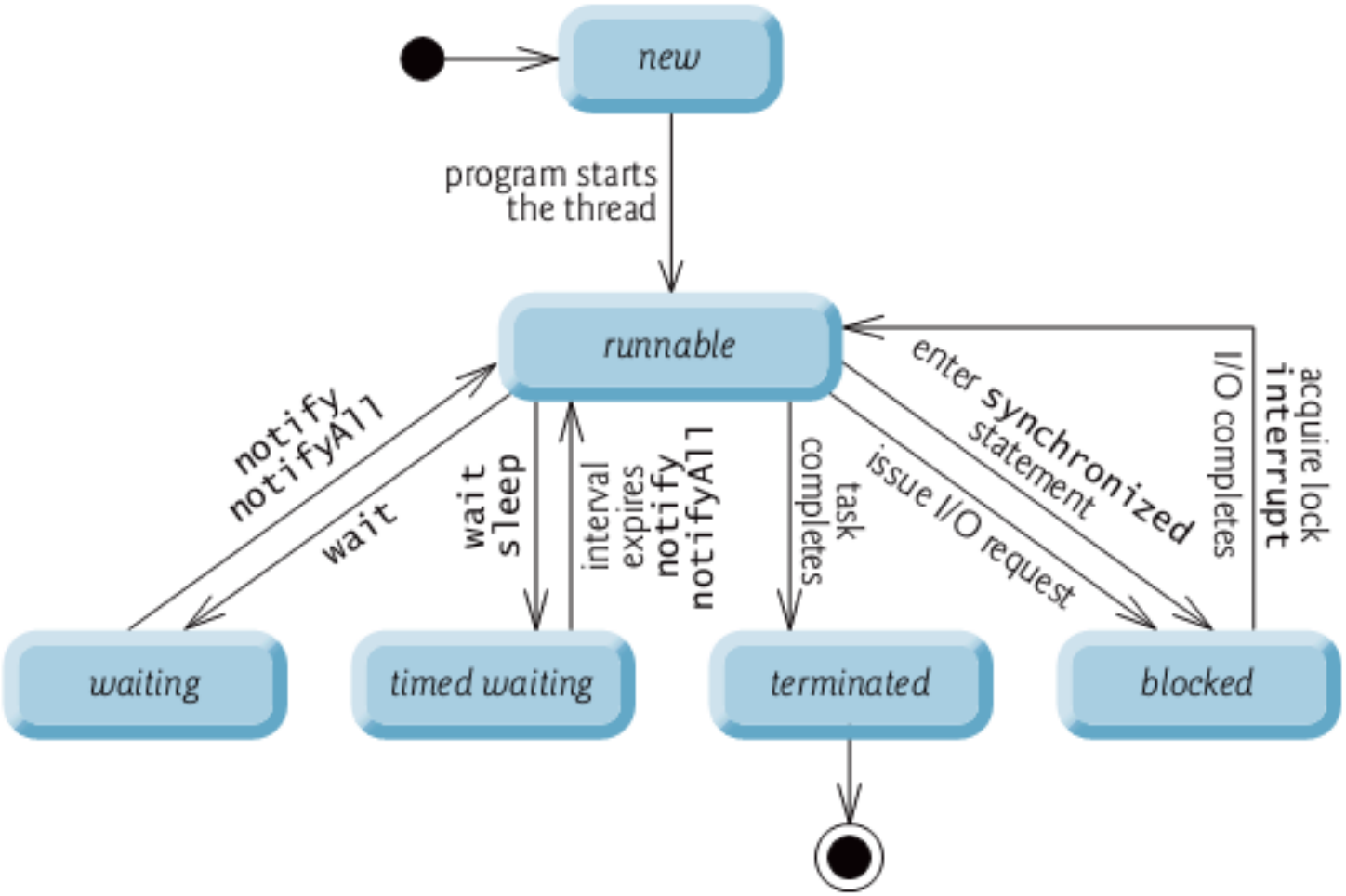
Thread.sleep

- `Thread.sleep` is a static method in the class `Thread` that pauses the thread that includes the invocation
  - It pauses for the number of milliseconds given as an argument
  - Note that it may be invoked in an ordinary program to insert a pause in the single thread of that program
- It may throw a checked exception, `InterruptedException`, which must be caught or declared
  - Both the `Thread` and `InterruptedException` classes are in the package `java.lang`

Thread States

- **`getState()`:** (Java 5) method returns an **`Enum`** of **`Thread.States`**. Thread states:
  - NEW – a fresh thread that has not yet started to execute.
  - RUNNABLE – a thread that is executing in the Java virtual machine.
  - BLOCKED – a thread that is blocked waiting for a monitor lock.
  - WAITING – a thread that is waiting to be notified by another thread.
  - TIMED_WAITING – a thread that is waiting to be notified by another thread for a specific amount of time.
  - TERMINATED – a thread whose run method has ended.

Thread Life Cycle



Thread Scheduler

- The thread scheduler runs each thread for a short amount of time (a *time slice*)
- Then the scheduler activates another thread
- There will always be slight variations in running times especially when calling operating system services (e.g. input and output)
- There is no guarantee about the order in which threads are executed

Terminating Threads

- A thread terminates when its `run` method terminates
- Do *not* terminate a thread using the deprecated `stop` method
- Instead, notify a thread that it should terminate

```
t.interrupt();
```

- `interrupt` does not cause the thread to terminate – it sets a boolean field in the thread data structure

Terminating Threads

- The **`run()`** method should check occasionally whether it has been interrupted
  - Use the **`interrupted`** method
  - An interrupted thread should release resources, clean up, and exit

```
public void run()
{
    for (int i = 1;
          i <= REPETITIONS && !Thread.interrupted(); i++)
    {
        Do work
    }
    Clean up
}
```

Terminating Threads

- The `sleep` method throws an `InterruptedException` when a sleeping thread is interrupted
  - Catch the exception
  - Terminate the thread

```
public void run() {
    try {
        for (int i = 1; i <= REPETITIONS; i++)
        {
            Do work
        }
    }
    catch (InterruptedException exception) {
    }
    Clean up
}
```

Terminating Threads

- Java does not force a thread to terminate when it is interrupted
- It is entirely up to the thread what it does when it is interrupted
- Interrupting is a general mechanism for getting the thread's attention

Animations

- Popular task for a thread in Java: control an animation
  - A thread process can direct the drawing of each frame while other aspects of the interface, such as
  - responding to user input, can continue in parallel
- Demos: clock, drop2d, sunsort

Reading

- Eckel: chapter 19 & 22
- Deitel: chapter 26
- Java API documentation

Summary

- More Java New I/O
  - Buffer
    - Views
    - Endian-ness
    - Direct ~
    - Memory mapped ~
  - File channels
    - Memory mapped files
    - Direct transfers
    - Locking
  - `String` class – regular expressions
- Intro to Java Threads
  - Multitasking vs. multithreading
  - Thread creation:
    - Subclassing `Thread`
    - `Runnable` implementation
  - Stopping/pausing ~
  - Terminating ~
  - Simple animation with threads