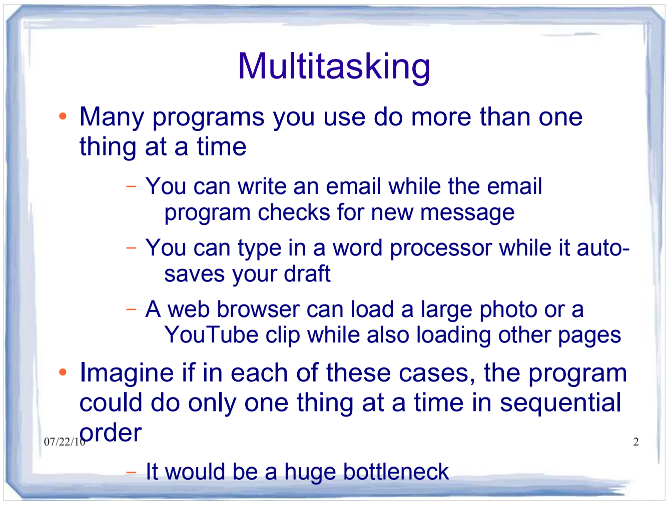
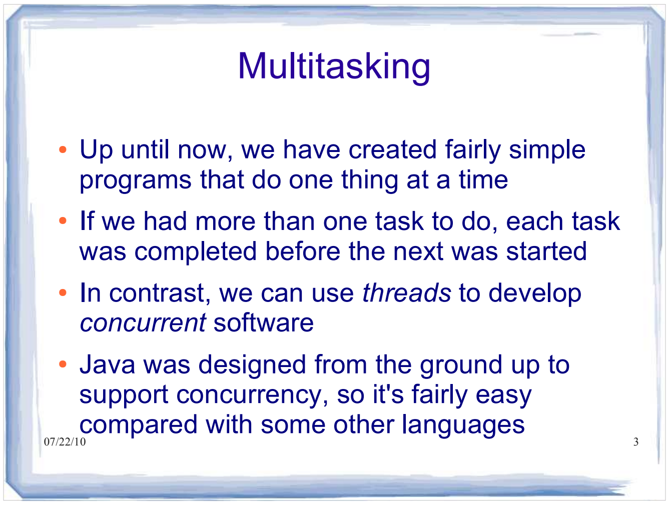
Review



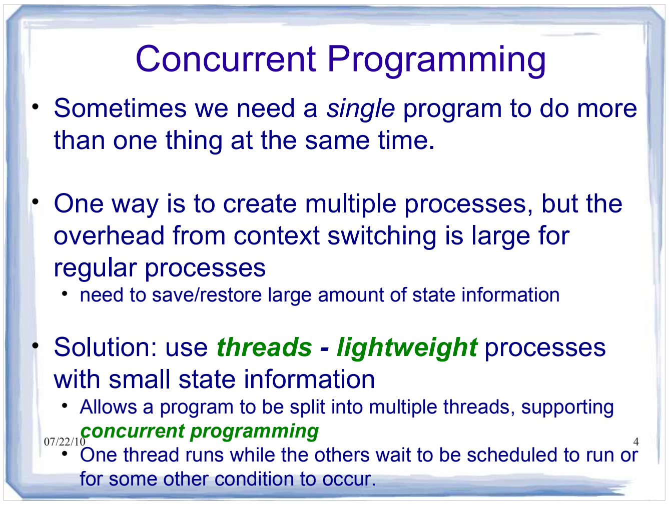
Multitasking

- Many programs you use do more than one thing at a time
  - You can write an email while the email program checks for new message
  - You can type in a word processor while it auto-saves your draft
  - A web browser can load a large photo or a YouTube clip while also loading other pages
- Imagine if in each of these cases, the program could do only one thing at a time in sequential order
  - It would be a huge bottleneck



Multitasking

- Up until now, we have created fairly simple programs that do one thing at a time
- If we had more than one task to do, each task was completed before the next was started
- In contrast, we can use *threads* to develop *concurrent* software
- Java was designed from the ground up to support concurrency, so it's fairly easy compared with some other languages



Concurrent Programming

- Sometimes we need a *single* program to do more than one thing at the same time.
- One way is to create multiple processes, but the overhead from context switching is large for regular processes
  - need to save/restore large amount of state information
- Solution: use ***threads - lightweight*** processes with small state information
  - Allows a program to be split into multiple threads, supporting ***concurrent programming***
  - One thread runs while the others wait to be scheduled to run or for some other condition to occur.

Java Threads

- The JVM supports threads and concurrent programming.
- If a program has more than one thread, a (JVM) scheduler will determine when each thread gets to run.
- There are two types of schedulers:
  - ***pre-emptive***: each thread is allowed to run for a maximum amount of time (a time slice) before it is suspended and another thread is allowed to run
  - ***non pre-emptive***: once a thread is allowed to run it continues to run until it has completed its task or until it explicitly yields to another thread
- The scheduler in *most* JVMs is pre-emptive.
  - As this is not guaranteed, we must not write code that assumes a pre-emptive scheduler.

Java Threads

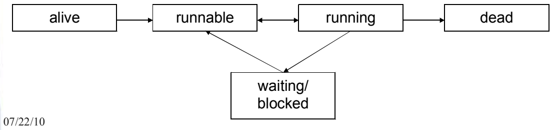
- A thread in Java
  - is an instance of the `Thread` class
  - has a priority and an optional name
  - has a `start()` method
    - this method puts the thread into the *runnable* state so that it can be selected for execution by the thread scheduler
  - has a `run()` method
    - the code in this method is executed when the thread runs
    - it is overridden to specify the particular behaviour of the thread

Thread States

- A thread can be in one of five states:
  - ***alive***
    - the thread has been constructed but start() has not been called
  - ***runnable***
    - the thread is ready to be scheduled to run
  - ***running***
    - the run() method is executing
  - ***waiting/blocking***
    - the thread can't run until some event has occurred (e.g., the passing of a certain amount of time)
  - ***dead***
    - the run() method has run to completion

Thread States

- The diagram below shows the transitions that can occur between these five states.
- For example, a thread can transition from *runnable* to *running,* or vice versa.
- However, a thread cannot transition directly from *runnable* to *dead*.



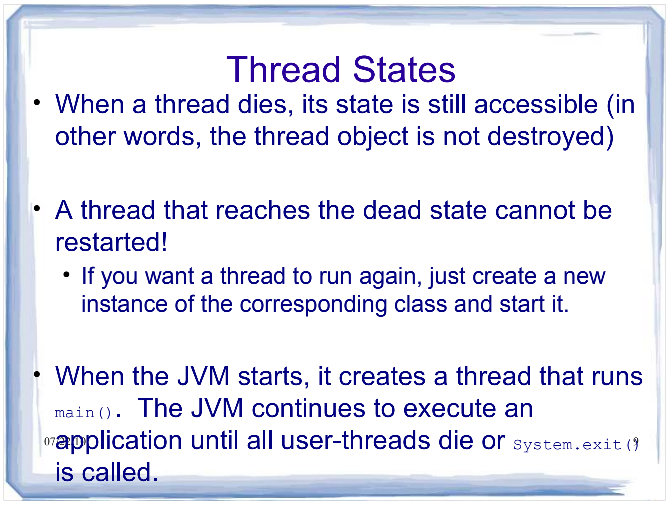
alive

runnable

running

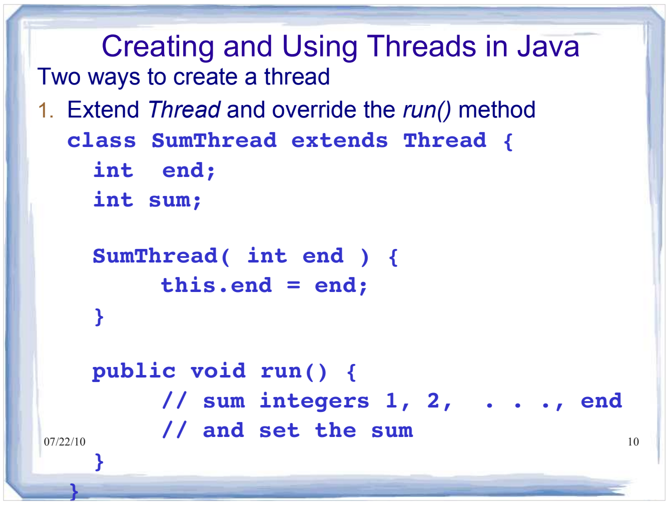
dead

waiting/ blocked



Thread States

- When a thread dies, its state is still accessible (in other words, the thread object is not destroyed)
- A thread that reaches the dead state cannot be restarted!
  - If you want a thread to run again, just create a new instance of the corresponding class and start it.
- When the JVM starts, it creates a thread that runs `main()`. The JVM continues to execute an application until all user-threads die or `System.exit()` is called.

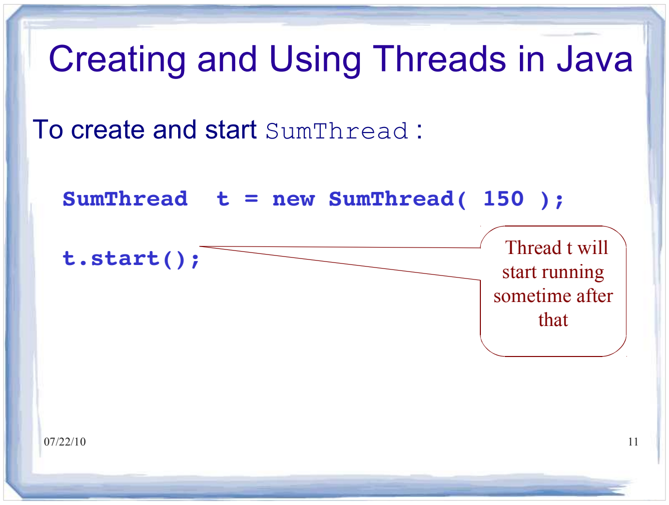


Creating and Using Threads in Java

Two ways to create a thread

1. Extend *Thread* and override the *run()* method

```
class SumThread extends Thread {
  int  end;
  int sum;

  SumThread( int end ) {
       this.end = end;
  }

  public void run() {
       // sum integers 1, 2,  . . ., end
       // and set the sum
  }
}
```

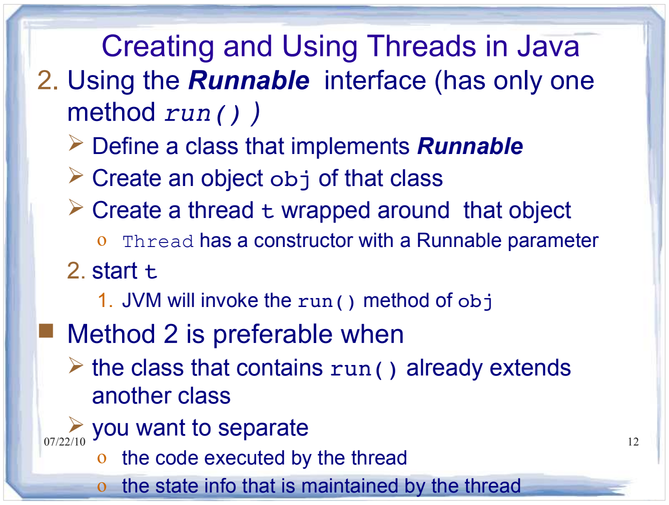


Creating and Using Threads in Java

To create and start `SumThread`:

```
SumThread  t = new SumThread( 150 );

t.start();
```

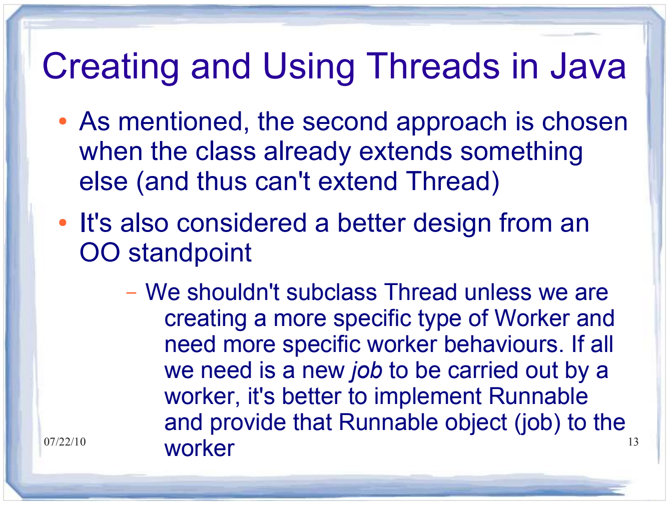
Thread t will start running sometime after that



Creating and Using Threads in Java

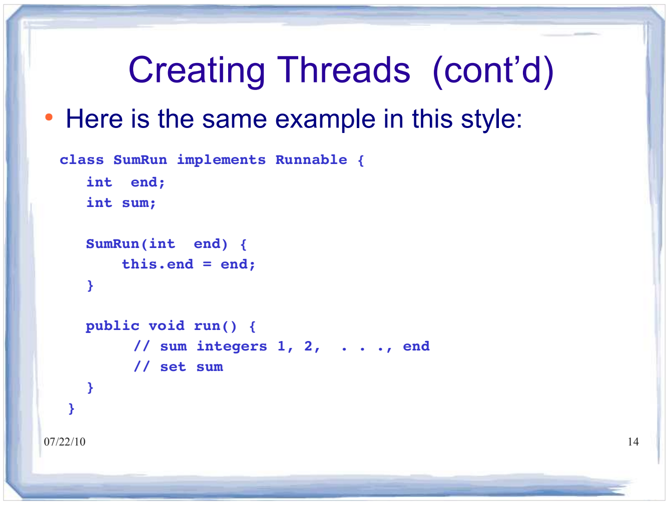
2. Using the ***Runnable*** interface (has only one method *`run()`*)
   - Define a class that implements ***Runnable***
   - Create an object `obj` of that class
   - Create a thread `t` wrapped around that object
     - `Thread` has a constructor with a Runnable parameter
   2. start `t`
      1. JVM will invoke the `run()` method of `obj`

- Method 2 is preferable when
  - the class that contains `run()` already extends another class
  - you want to separate
    - the code executed by the thread
    - the state info that is maintained by the thread



Creating and Using Threads in Java

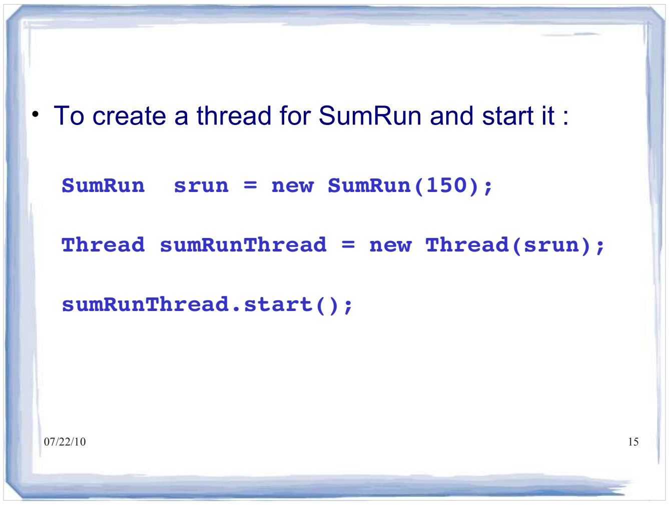
- As mentioned, the second approach is chosen when the class already extends something else (and thus can't extend Thread)
- It's also considered a better design from an OO standpoint
  - We shouldn't subclass Thread unless we are creating a more specific type of Worker and need more specific worker behaviours. If all we need is a new *job* to be carried out by a worker, it's better to implement Runnable and provide that Runnable object (job) to the worker



Creating Threads (cont’d)

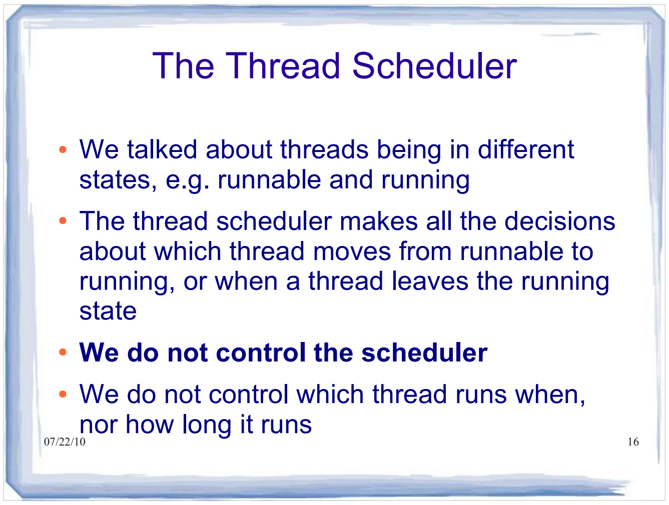
- Here is the same example in this style:

```
class SumRun implements Runnable {
   int  end;
   int sum;

   SumRun(int  end) {
       this.end = end;
   }

   public void run() {
        // sum integers 1, 2,  . . ., end
        // set sum
   }
 }
```



- To create a thread for SumRun and start it :

```
SumRun  srun = new SumRun(150);

Thread sumRunThread = new Thread(srun);

sumRunThread.start();
```



The Thread Scheduler

- We talked about threads being in different states, e.g. runnable and running
- The thread scheduler makes all the decisions about which thread moves from runnable to running, or when a thread leaves the running state
- **We do not control the scheduler**
- We do not control which thread runs when, nor how long it runs

The Thread Scheduler

- Because of this, we should never write code that depends on the scheduler working in a particular way
- We cannot assume that threadA will run to completion and then threadB will run to completion and so on
- There are some things we can do to affect which threads are run
  - e.g. putting a thread to sleep for a few milliseconds gives other threads a chance to run
  - More on that in a moment

So, what gets printed if we run this?

```
public class ThreadTester {

public static void main(String[] args)
{
Runnable threadJob = new MyRunnable();
Thread myThread = new Thread(threadJob);
myThread.start();
System.out.println("back in main");
}
}
```

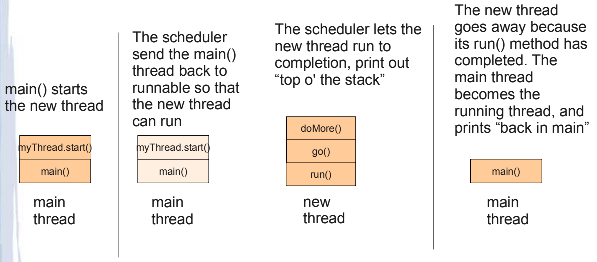
Answer: it depends

- It depends on the scheduler and how the scheduler decides what to run and when
- If you run this program multiple times, you are liable to get different results:

```
% java ThreadTester
   back in main
   top o' the stack
% java ThreadTester
   top o' the stack
   back in main
```

Why does it vary?

- Sometimes it runs like this:



main() starts the new thread

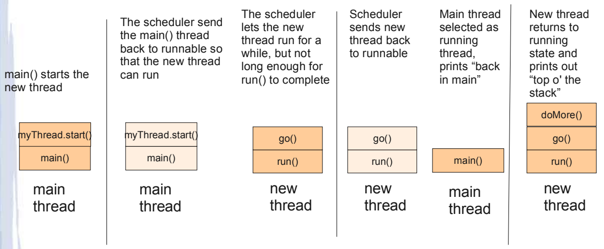
The scheduler send the main() thread back to runnable so that the new thread can run

The scheduler lets the new thread run to completion, print out “top o' the stack”

The new thread goes away because its run() method has completed. The main thread becomes the running thread, and prints “back in main”

Why does it vary?

- And sometimes it runs like this:



main() starts the new thread

The scheduler send the main() thread back to runnable so that the new thread can run

The scheduler lets the new thread run for a while, but not long enough for run() to complete

Scheduler sends new thread back to runnable

Main thread selected as running thread, prints "back in main"

New thread returns to running state and prints out "top o' the stack"

Thread Scheduler

- As mentioned, we can't assume that the new thread will be allowed to run to completion

sleep(), yield() and join()

- We can have an effect on which thread gets run by calling one of these methods
  - sleep() puts the current running thread to sleep for some amount of time (in waiting state, then becomes runnable)
  - yield() puts the current running thread into runnable state)
  - If running thread t calls join(r) then t will be in waiting/blocked state until r finishes
- Allow other threads to compete for running state
- We can also assign priorities to threads (0-10)

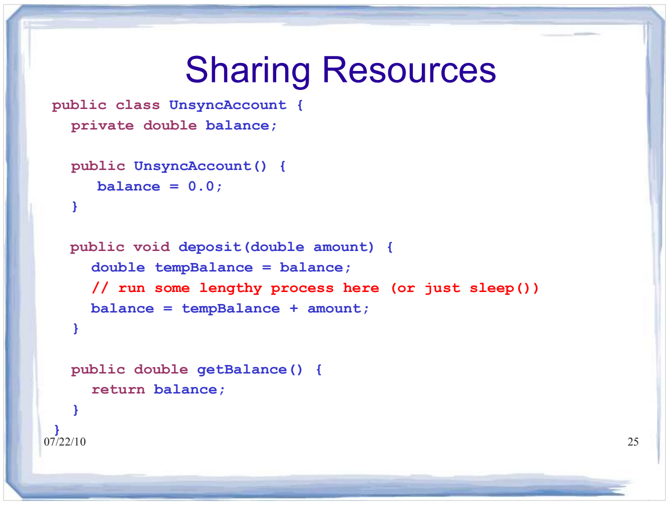
Sharing Resources

- Many threads may need to access the same resource (object, file, memory, etc.). Such cases must be handled carefully.

- In the following (very contrived) example, we'll create a single bank account with an initial balance of $0 that will be shared by three threads.

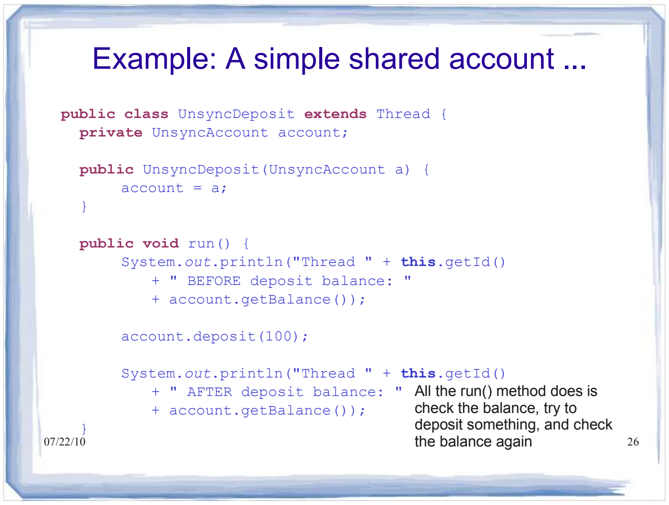
- Each thread will deposit $100 to the account.

- We'll see that, unless we're careful, the account will not have a deposit of $300 by the time the three threads have finished running…



Sharing Resources

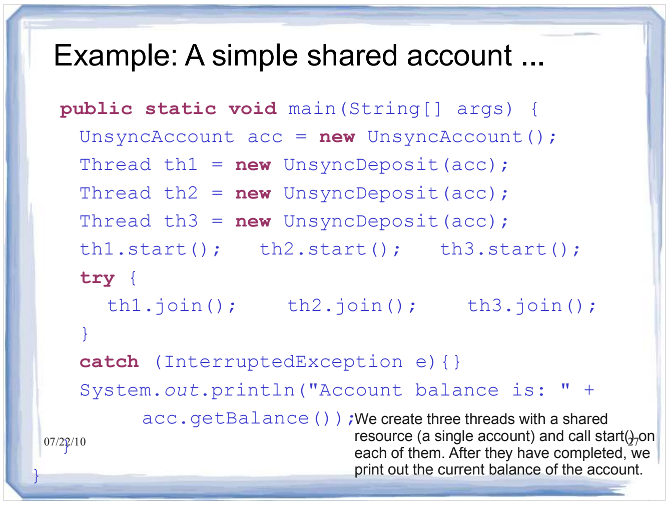
```
public class UnsyncAccount {
  private double balance;

  public UnsyncAccount() {
     balance = 0.0;
  }

  public void deposit(double amount) {
    double tempBalance = balance;
    // run some lengthy process here (or just sleep())
    balance = tempBalance + amount;
  }

  public double getBalance() {
    return balance;
  }
}
```



Example: A simple shared account ...

```
public class UnsyncDeposit extends Thread {
  private UnsyncAccount account;

  public UnsyncDeposit(UnsyncAccount a) {
      account = a;
  }

  public void run() {
      System.out.println("Thread " + this.getId()
         + " BEFORE deposit balance: "
         + account.getBalance());

      account.deposit(100);

      System.out.println("Thread " + this.getId()
         + " AFTER deposit balance: "
         + account.getBalance());
  }
```

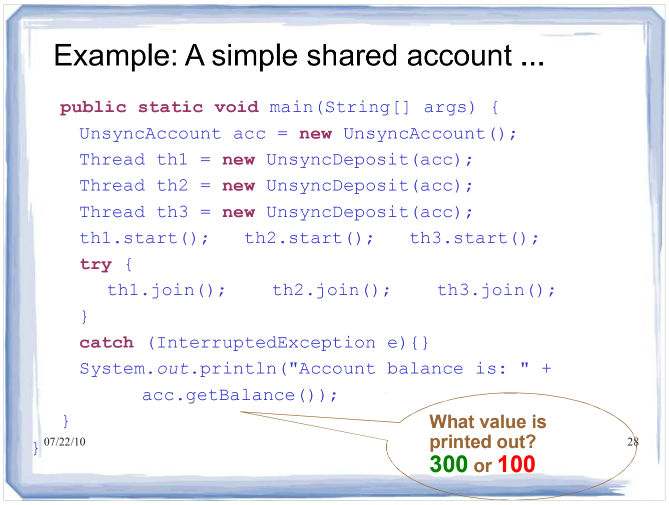
All the run() method does is check the balance, try to deposit something, and check the balance again



Example: A simple shared account ...

```
public static void main(String[] args) {
  UnsyncAccount acc = new UnsyncAccount();
  Thread th1 = new UnsyncDeposit(acc);
  Thread th2 = new UnsyncDeposit(acc);
  Thread th3 = new UnsyncDeposit(acc);
  th1.start();   th2.start();   th3.start();
  try {
    th1.join();    th2.join();    th3.join();
  }
  catch (InterruptedException e){}
  System.out.println("Account balance is: " +
        acc.getBalance());
  }
}
```

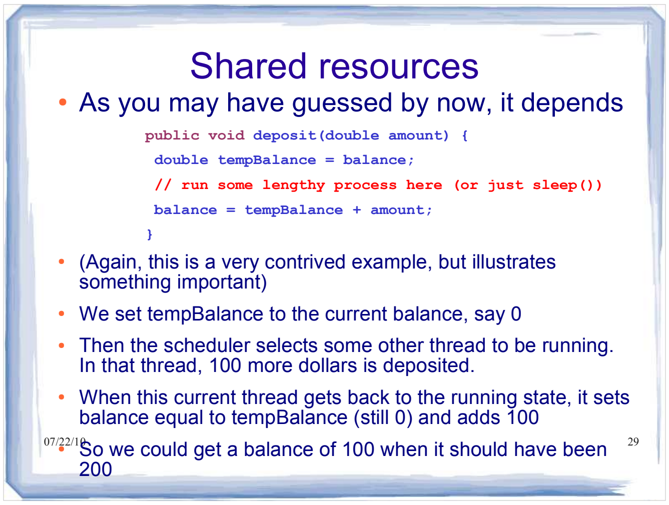
We create three threads with a shared resource (a single account) and call start() on each of them. After they have completed, we print out the current balance of the account.



Example: A simple shared account ...

```
public static void main(String[] args) {
  UnsyncAccount acc = new UnsyncAccount();
  Thread th1 = new UnsyncDeposit(acc);
  Thread th2 = new UnsyncDeposit(acc);
  Thread th3 = new UnsyncDeposit(acc);
  th1.start();   th2.start();   th3.start();
  try {
    th1.join();    th2.join();    th3.join();
  }
  catch (InterruptedException e){}
  System.out.println("Account balance is: " +
        acc.getBalance());
 }
}
```

What value is printed out?
300 or **100**

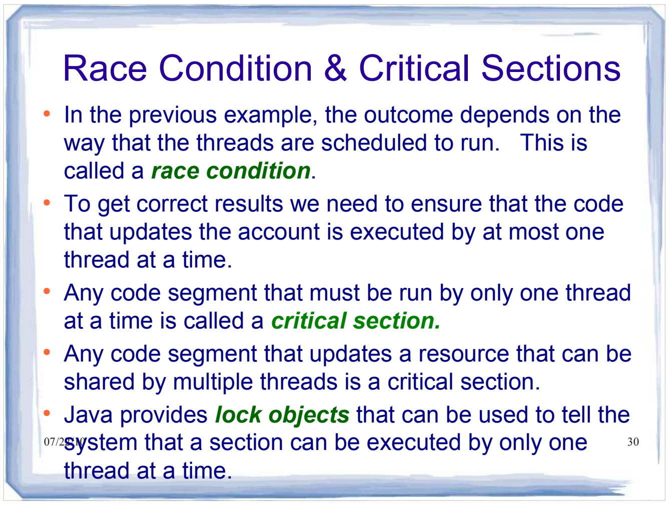


Shared resources

- As you may have guessed by now, it depends

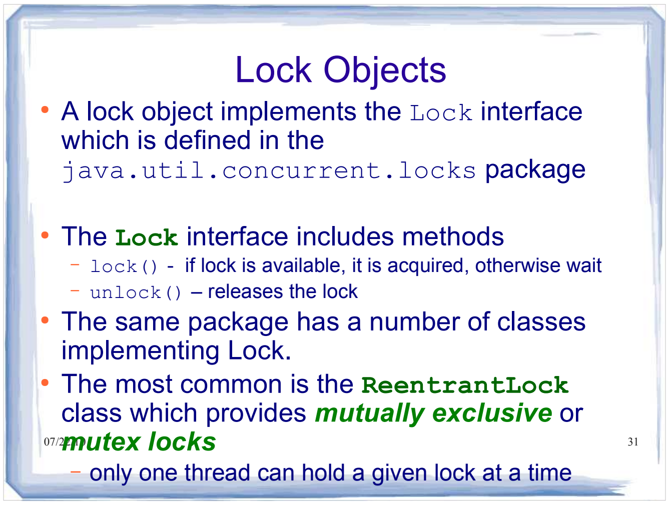
```
public void deposit(double amount) {
 double tempBalance = balance;
 // run some lengthy process here (or just sleep())
 balance = tempBalance + amount;
}
```

- (Again, this is a very contrived example, but illustrates something important)
- We set tempBalance to the current balance, say 0
- Then the scheduler selects some other thread to be running. In that thread, 100 more dollars is deposited.
- When this current thread gets back to the running state, it sets balance equal to tempBalance (still 0) and adds 100
- So we could get a balance of 100 when it should have been 200



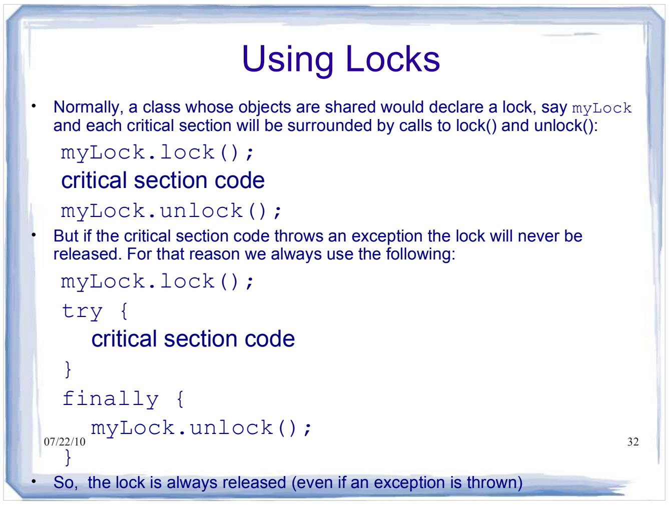
Race Condition & Critical Sections

- In the previous example, the outcome depends on the way that the threads are scheduled to run. This is called a ***race condition***.
- To get correct results we need to ensure that the code that updates the account is executed by at most one thread at a time.
- Any code segment that must be run by only one thread at a time is called a ***critical section.***
- Any code segment that updates a resource that can be shared by multiple threads is a critical section.
- Java provides ***lock objects*** that can be used to tell the system that a section can be executed by only one thread at a time.



Lock Objects

- A lock object implements the `Lock` interface which is defined in the `java.util.concurrent.locks` package
- The **`Lock`** interface includes methods
  - `lock()` - if lock is available, it is acquired, otherwise wait
  - `unlock()` – releases the lock
- The same package has a number of classes implementing Lock.
- The most common is the **`ReentrantLock`** class which provides ***mutually exclusive*** or ***mutex locks***
  - only one thread can hold a given lock at a time



Using Locks

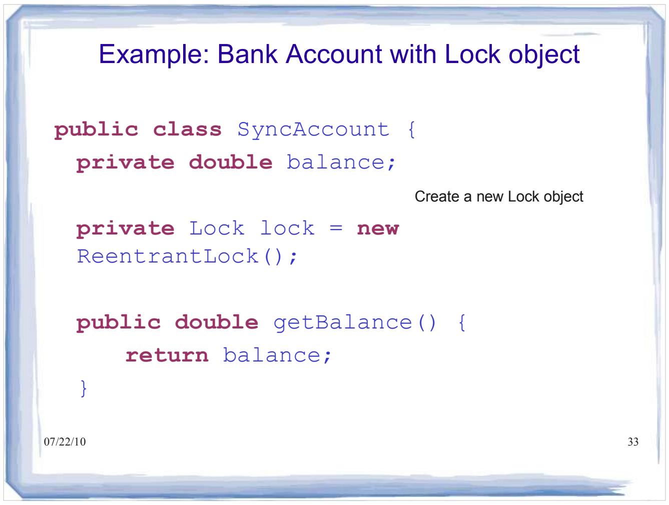
- Normally, a class whose objects are shared would declare a lock, say `myLock` and each critical section will be surrounded by calls to lock() and unlock():

```
myLock.lock();
critical section code
myLock.unlock();
```

- But if the critical section code throws an exception the lock will never be released. For that reason we always use the following:

```
myLock.lock();
try {
   critical section code
}
finally {
   myLock.unlock();
}
```

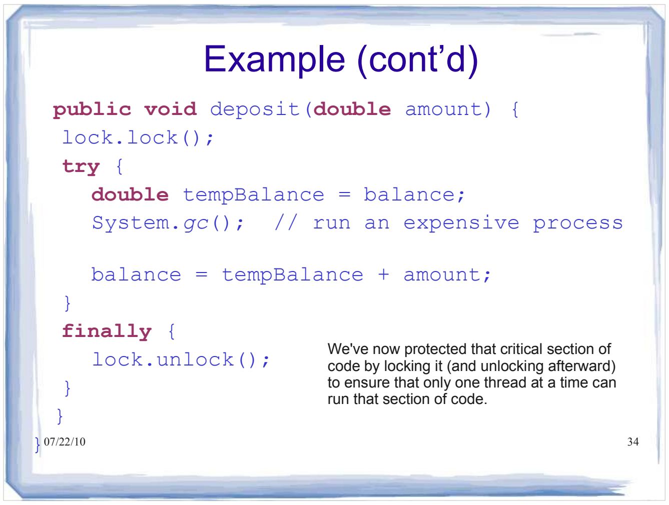
- So, the lock is always released (even if an exception is thrown)



Example: Bank Account with Lock object

```
public class SyncAccount {
  private double balance;

  private Lock lock = new
  ReentrantLock();

  public double getBalance() {
      return balance;
  }
```

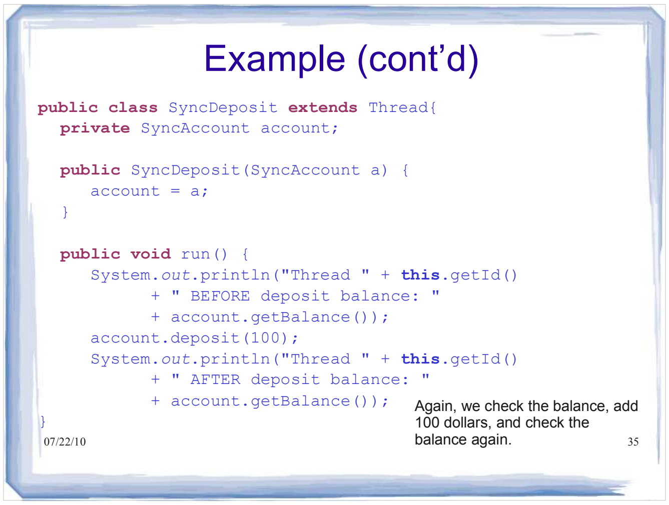
Create a new Lock object



Example (cont'd)

```
public void deposit(double amount) {
 lock.lock();
 try {
   double tempBalance = balance;
   System.gc();  // run an expensive process

   balance = tempBalance + amount;
 }
 finally {
   lock.unlock();
 }
}
}
```

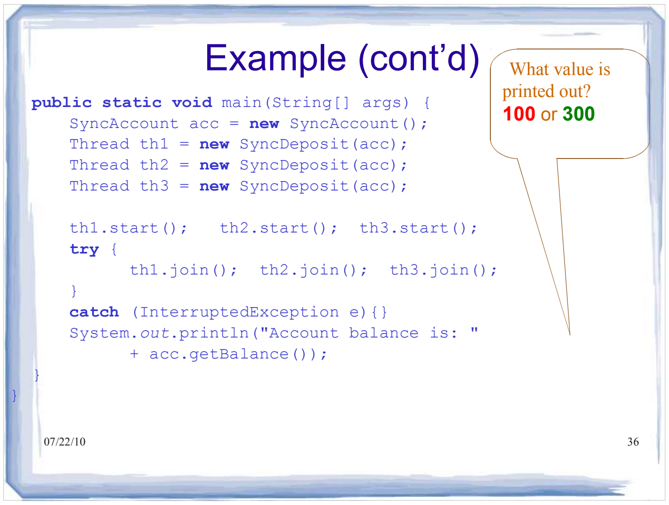
We've now protected that critical section of code by locking it (and unlocking afterward) to ensure that only one thread at a time can run that section of code.



Example (cont'd)

```
public class SyncDeposit extends Thread{
  private SyncAccount account;

  public SyncDeposit(SyncAccount a) {
     account = a;
  }

  public void run() {
     System.out.println("Thread " + this.getId()
           + " BEFORE deposit balance: "
           + account.getBalance());
     account.deposit(100);
     System.out.println("Thread " + this.getId()
           + " AFTER deposit balance: "
           + account.getBalance());
}
```

Again, we check the balance, add 100 dollars, and check the balance again.



Example (cont'd)

```
public static void main(String[] args) {
    SyncAccount acc = new SyncAccount();
    Thread th1 = new SyncDeposit(acc);
    Thread th2 = new SyncDeposit(acc);
    Thread th3 = new SyncDeposit(acc);

    th1.start();   th2.start();  th3.start();
    try {
          th1.join();  th2.join();  th3.join();
    }
    catch (InterruptedException e){}
    System.out.println("Account balance is: "
          + acc.getBalance());
}
}
```

What value is printed out?
100 or **300**

Example (cont'd)

- The appropriate use of a lock ensures that at most one thread can be running the critical section of code in the `deposit()` method at any given time.
- Now, every time we run this program, the balance on the account will be $300.

Synchronized Methods of Old Versions of Java

- Older versions of Java (prior to 1.5)do not have lock objects.
- Instead, every object has a lock that behaves like a ReentrantLock.
- If the lock is available, it is acquired when a synchronized method is called.
- A ***synchronized method*** is declared as

```
public synchronized void push(Object item)
 {
     // code for the method goes here
 }
```

and is synchronized on the lock of its implicit argument (**this**)

Synchronized Methods

- Synchronized instance methods allow at most one thread to run *any* of the object's synchronized methods at any time.
- Synchronized methods are simpler but less flexible.
- The `Account` class would be defined as follows if we use synchronized methods...

Account Example with Synchronized Methods

```
public class SyncAccount {
  private double balance;

  public synchronized void deposit(double amount) {
      double tempBalance = balance;
      System.gc(); // run an expensive process
      balance = tempBalance + amount;
   }

  public double getBalance() {
        return balance;
  }
}
```

Another deadlock example

- Imagine a simple BankAccount class with deposit() and withdraw() methods.

```
public void withdraw(double amount)
{
  balanceChangeLock.lock();
  try
   {
      while (balance < amount)
      . . .
   finally
{
      balanceChangeLock.unlock();
}
}
```

Our BankAccount class has a lock because its methods access a shared resource, the balance. So in the withdraw method, we acquire the lock.

We put in a while loop to wait until the balance is sufficient to allow the withdrawal.

But how do we wait? If we put the thread to sleep, the lock will not be released and no threads will be able to call deposit because they will be unable to get the lock. We will be in a **deadlock** situation.

Synchronization Using Conditions

- To resolve this problem we should use ***condition objects***
- A condition object allows a thread to release a lock temporarily, so another thread can get that lock and run
- Each condition object belongs to a lock object and is created as follows:

```
Condition myCondition =
  lock.newCondition();
```

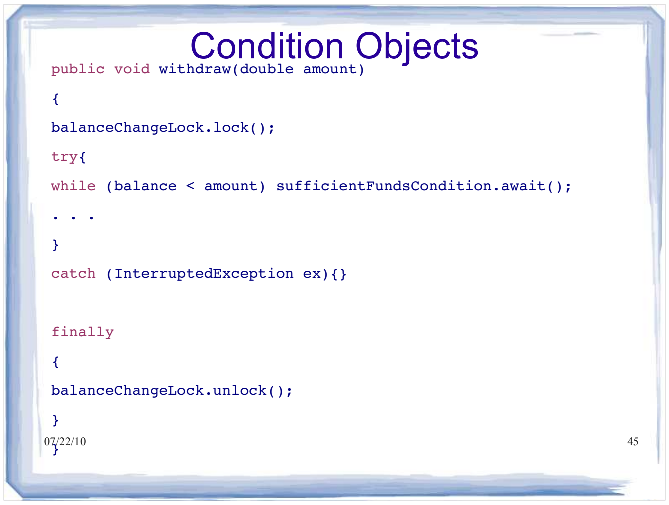
Synchronization Using Conditions

- A condition object implements the `Condition` interface that includes:
  - `await()`
    - the current thread releases the associated lock
    - the current thread moves to the *wait/blocked* state until another thread calls `signal()` or `signalAll()` on this condition
  - `signal()` or `signalAll()`
    - causes one or all of the threads that are blocked waiting on the condition to move to the *runnable* state
    - these threads will compete to get the lock again
    - one of them will get the lock and continue to run

Condition Objects

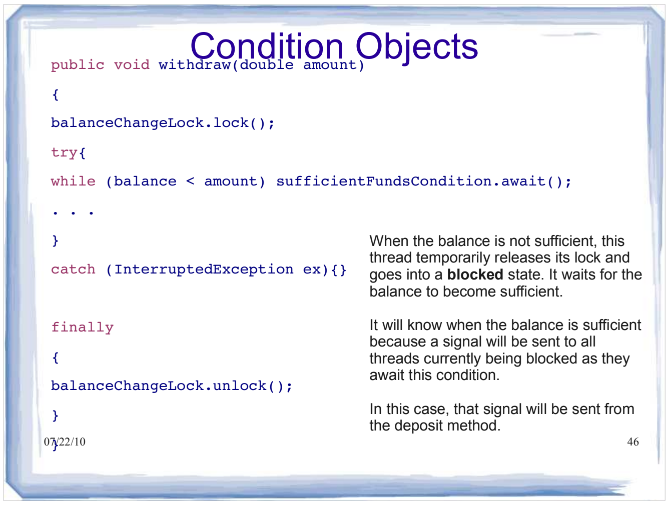
- We can again use a condition object

```
public class BankAccount {
   private Lock balanceChangeLock;
   private Condition sufficientFundsCondition;
   private int balance;

   public BankAccount()
   {
      balanceChangeLock = new ReentrantLock();
      SufficientFundsCondition = balanceChangeLock.newCondition();
      . . .
   }
 . . .
}
```



Condition Objects

```
public void withdraw(double amount)
{
balanceChangeLock.lock();
try{
while (balance < amount) sufficientFundsCondition.await();
. . .
}
catch (InterruptedException ex){}

finally
{
balanceChangeLock.unlock();
}
}
```



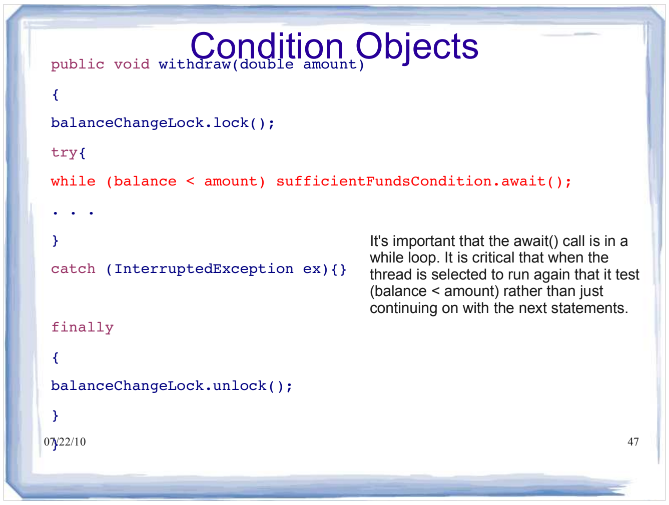
Condition Objects

```
public void withdraw(double amount)
{
balanceChangeLock.lock();
try{
while (balance < amount) sufficientFundsCondition.await();
. . .
}
catch (InterruptedException ex){}

finally
{
balanceChangeLock.unlock();
}
}
```

When the balance is not sufficient, this thread temporarily releases its lock and goes into a **blocked** state. It waits for the balance to become sufficient.

It will know when the balance is sufficient because a signal will be sent to all threads currently being blocked as they await this condition.

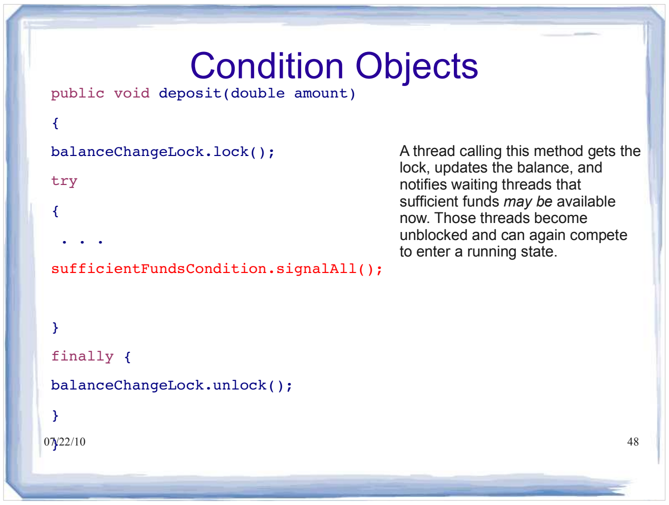
In this case, that signal will be sent from the deposit method.



Condition Objects

```
public void withdraw(double amount)
{
balanceChangeLock.lock();
try{
while (balance < amount) sufficientFundsCondition.await();
. . .
}
catch (InterruptedException ex){}

finally
{
balanceChangeLock.unlock();
}
}
```

It's important that the await() call is in a while loop. It is critical that when the thread is selected to run again that it test (balance < amount) rather than just continuing on with the next statements.



Condition Objects

```
public void deposit(double amount)
{
balanceChangeLock.lock();
try
{
 . . .
sufficientFundsCondition.signalAll();

}
finally {
balanceChangeLock.unlock();
}
}
```

A thread calling this method gets the lock, updates the balance, and notifies waiting threads that sufficient funds *may be* available now. Those threads become unblocked and can again compete to enter a running state.

Common Errors

- Calling await() without calling signalAll()
  - If a thread calls await() there needs to be a matching signalAll() that can be called by other threads, otherwise it will wait forever
- Calling signalAll() without locking the Object
  - A thread must own the lock that belongs to the condition object on which signalAll() is called. You'll get an exception otherwise.

Conclusion

- We’ve discussed how to build programs with multiple threads.
- To synchronize threads we can use Java’s primitives:
  - lock objects
  - condition objects

Streams and Persistent Objects

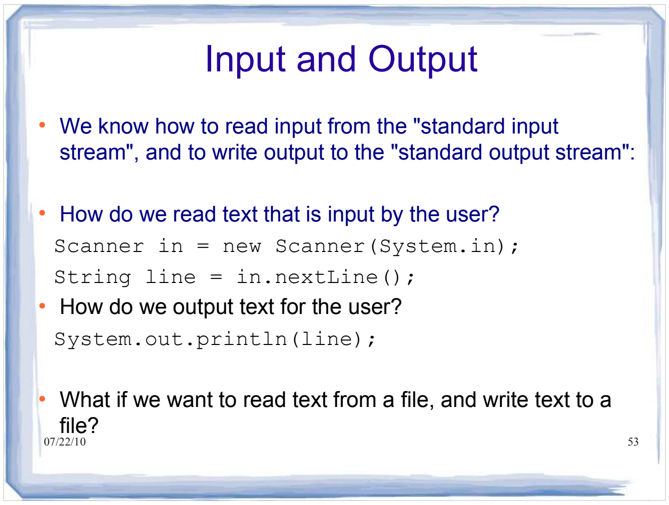
Reading:

- 2nd Ed: 16.1-16.3, 16.5
- 3rd Ed: 11.1, 19.1, 19.4

- http://java.sun.com/docs/books/tutorial/essential/io/streams.html

Learning Objectives

- describe stream abstraction used in Java for byte and character input/output
- write programs that use streams to read and write data
- incorporate data persistence in a program using Java's serialization mechanism



Input and Output

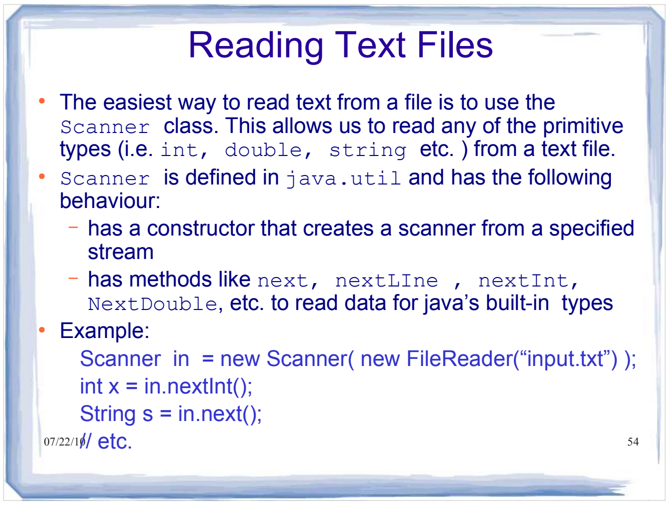
- We know how to read input from the "standard input stream", and to write output to the "standard output stream":
- How do we read text that is input by the user?

```
Scanner in = new Scanner(System.in);
String line = in.nextLine();
```

- How do we output text for the user?

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

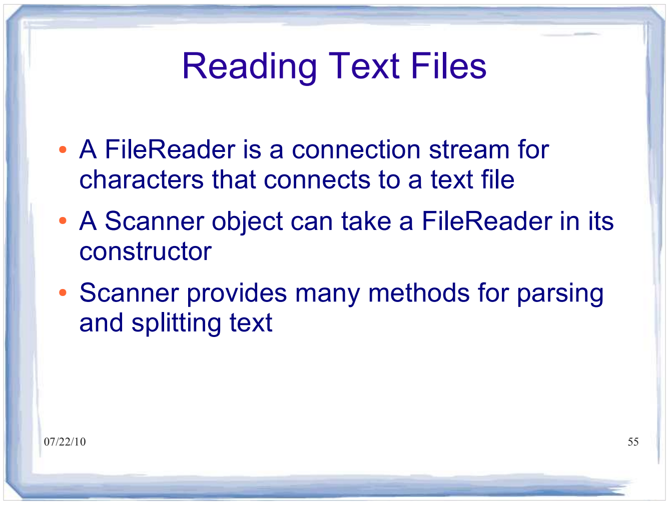
- What if we want to read text from a file, and write text to a file?



Reading Text Files

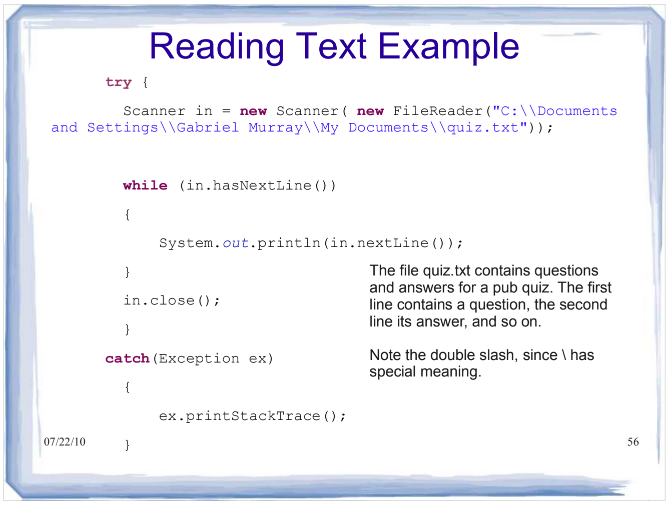
- The easiest way to read text from a file is to use the `Scanner` class. This allows us to read any of the primitive types (i.e. `int, double, string` etc.) from a text file.
- `Scanner` is defined in `java.util` and has the following behaviour:
  - has a constructor that creates a scanner from a specified stream
  - has methods like `next, nextLIne , nextInt, NextDouble`, etc. to read data for java's built-in types
- Example:

```
Scanner in = new Scanner( new FileReader(“input.txt”) );
int x = in.nextInt();
String s = in.next();
// etc.
```



Reading Text Files

- A FileReader is a connection stream for characters that connects to a text file
- A Scanner object can take a FileReader in its constructor
- Scanner provides many methods for parsing and splitting text

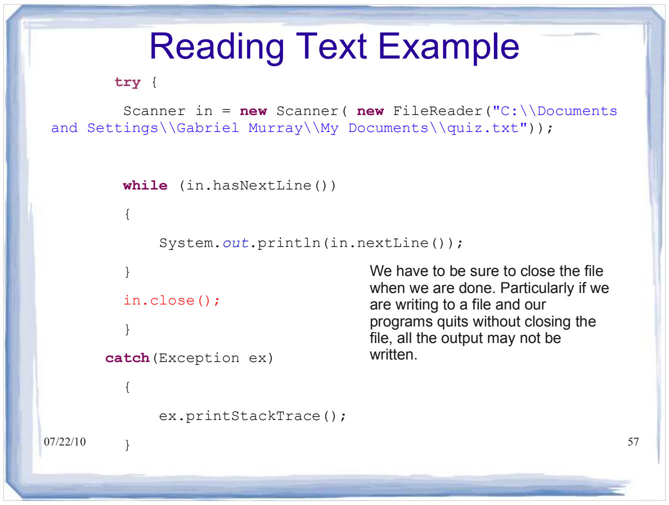


Reading Text Example

```
try {
  Scanner in = new Scanner( new FileReader("C:\\Documents and Settings\\Gabriel Murray\\My Documents\\quiz.txt"));

  while (in.hasNextLine())
  {
      System.out.println(in.nextLine());
  }
  in.close();
  }
catch(Exception ex)
  {
      ex.printStackTrace();
  }
```

The file quiz.txt contains questions and answers for a pub quiz. The first line contains a question, the second line its answer, and so on.

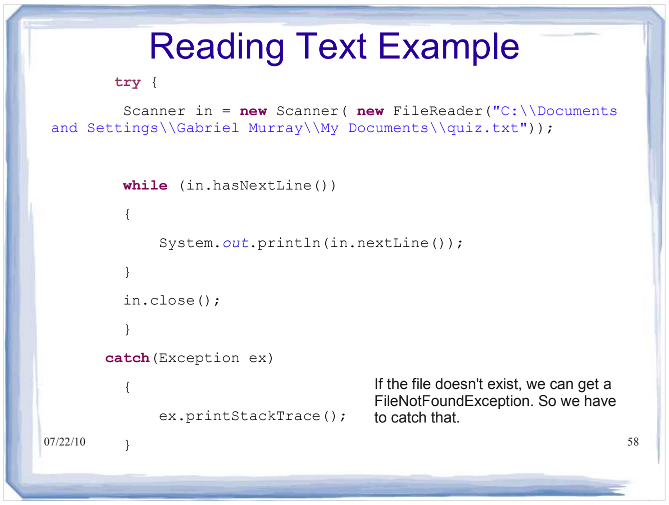
Note the double slash, since \ has special meaning.



Reading Text Example

```
      try {
        Scanner in = new Scanner( new FileReader("C:\\Documents
and Settings\\Gabriel Murray\\My Documents\\quiz.txt"));


        while (in.hasNextLine())
        {
            System.out.println(in.nextLine());
        }
        in.close();
        }
    catch(Exception ex)
        {
            ex.printStackTrace();
        }
```

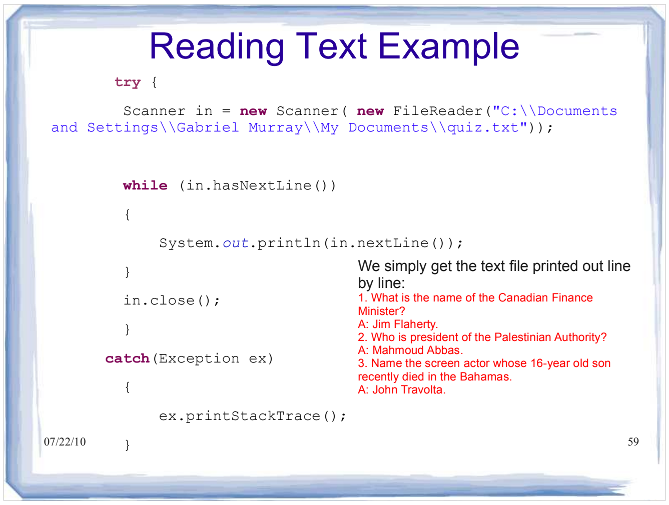
We have to be sure to close the file when we are done. Particularly if we are writing to a file and our programs quits without closing the file, all the output may not be written.



Reading Text Example

```
      try {
        Scanner in = new Scanner( new FileReader("C:\\Documents
and Settings\\Gabriel Murray\\My Documents\\quiz.txt"));


        while (in.hasNextLine())
        {
            System.out.println(in.nextLine());
        }
        in.close();
        }
    catch(Exception ex)
        {
            ex.printStackTrace();
        }
```

If the file doesn't exist, we can get a FileNotFoundException. So we have to catch that.

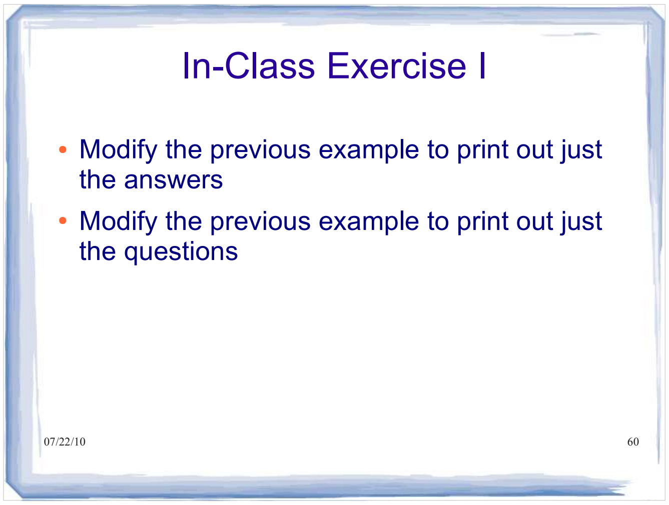


Reading Text Example

```
      try {
        Scanner in = new Scanner( new FileReader("C:\\Documents
and Settings\\Gabriel Murray\\My Documents\\quiz.txt"));


        while (in.hasNextLine())
        {
            System.out.println(in.nextLine());
        }
        in.close();
        }
    catch(Exception ex)
        {
            ex.printStackTrace();
        }
```

We simply get the text file printed out line by line:

```
1. What is the name of the Canadian Finance Minister?
A: Jim Flaherty.
2. Who is president of the Palestinian Authority?
A: Mahmoud Abbas.
3. Name the screen actor whose 16-year old son recently died in the Bahamas.
A: John Travolta.
```



In-Class Exercise I

- Modify the previous example to print out just the answers
- Modify the previous example to print out just the questions

Writing Text Files

- The easy way to write values of primitive types to a file is to use a `PrintWriter`
- A `PrintWriter`
  - can be directly associated with a file or with an `OutputStream`
  - has a constructor with the file name as parameter
  - has methods `print` and `println` which accept any of the primitive types and convert their values to strings
  - `print` and `println` convert general objects to strings using `toString()`
- Example:

```
PrintWriter  out  = new PrintWriter(“output.txt”);
out.println(123.05);
out.println(“Hello World”);
 out.println( new Account(“john”, 100) );
// etc.
```

What Are These Streams?

- What exactly are "FileReader" and "PrintWriter", "Reader", "OutputStream", and for that matter, "System.in", and "System.out"?

- `FileReader` and `PrintWriter` are examples of streams.

- `System.in` is also a stream, it's an instance of `InputStream`

- `System.out` is also a stream, it's an instance of `PrintStream`

What are streams?

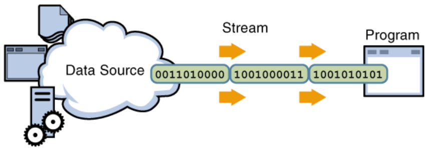
- Most programs exchange data either with other programs, or with devices, or both
  - e.g., the *myUBC* portal must load information about who you are when you login (likely from a database)
  - e.g., your web browser remembers recently visited sites, cookies, etc. on the local file system
  - e.g., if you google a term, you are sending data to a web server somewhere in the world and it is returning data to your web browser
- We use ***streams*** to abstract the concept of data flowing from one program/device to another
- Each stream has a ***source*** (from which data flows) and a ***sink*** (into which data flows)

source → sink

Data flows in FIFO order

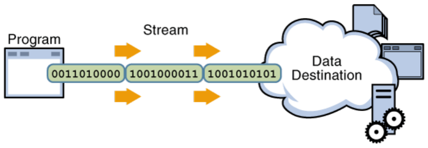
Streams

- Use an *input* stream to read data from a source



Streams

- Use an *output* stream to write data to a destination



Using a Stream

- To communicate with another program/device/file, a program
  - ➢ creates and opens a stream to (or from) the other program/device/file
  - ➢ transfers data through the stream (usually, one piece at a time)
  - ➢ closes the stream

Java Stream Types

- Java provides many stream classes (in the `java.io` package) to support I/O from devices with different characteristics
  - These stream classes are all used in basically the same way (it does not matter if the streams goes to a file, another computer, or somewhere else).
  - Two basic categories:
    1. ***Character*** streams, which are used to communicate (16-bit) characters in a platform independent way
    2. ***Byte*** streams, which are a sequence of (8-bit) bytes used to read/write binary data (images, sound), manipulate raw files, and for object serialization

Stream Classes

- Character Input Streams:
  - `FileReader`: input comes from a file.
    - Example: `Reader r = new FileReader("filename.txt");`
  - `StringReader`: input comes from a string.
    - Example: `Reader r = new StringReader("this is input");`
  - `CharArrayReader`: input comes from an array of `char`.
    - Example: `char[] myArray = {‘i’,’n’,’p’,’u’,’t’};`
      `Reader r = new CharArrayReader(myArray);`

Stream Classes

- Character Output Streams: same names, replacing `Reader` by `Writer`. There is one additional useful class:
  - `PrintWriter`: has `print()` and `println()` methods
- Byte Input and Output Streams: same names, using `InputStream` and `OutputStream` instead of `Reader` and `Writer`
- For more detail, see appendix at the end of these lecture notes

Example: Copying the contents of a file

- Copy the contents of the file `recentPhoto.jpg` to a new file called `myDog.jpg`:

```
int data;
try {
   FileInputStream in = new FileInputStream("recentPhoto.jpg");
   FileOutputStream  out = new FileOutputStream("myDog.jpg");
   while ((data = in.read()) != -1)
        out.write(data);
   in.close();
   out.close();
}
catch (IOException e) {
   System.err.println("File error: " + e.getMessage());
}
```

returns -1 when end of file is reached

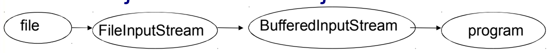
Source/Sink Streams

- Streams are associated with a source/sink, i.e. a device like a file:

Source/sink	**Char Streams**	**Byte Streams**
File	FileReader	FileInputStream
	FileWriter	FileOutputStream
	PrintWriter	PrintStream
Memory	CharArrayReader	ByteArrayInputStream
	CharArrayWriter	ByteArrayOutputStream
	StringReader	
	StringWriter	

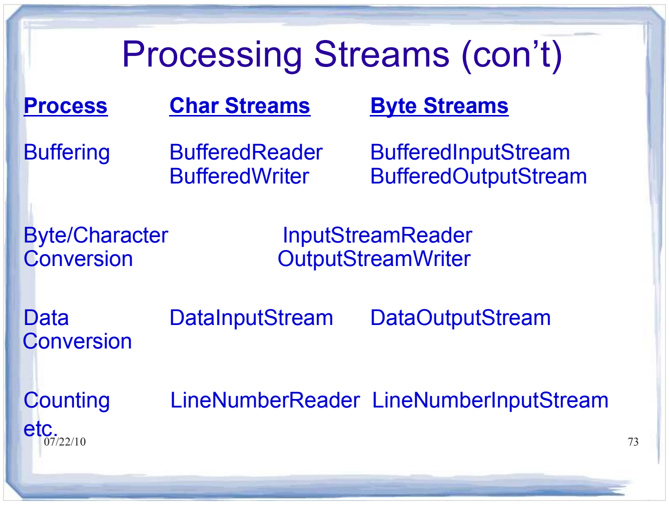
Processing Streams

- Placed between source/sink streams and perform data transformations
- For example, we may want to buffer the data as it's read and written
- We can do this by embedding our input and output stream objects in buffer objects.



- The code would look like:

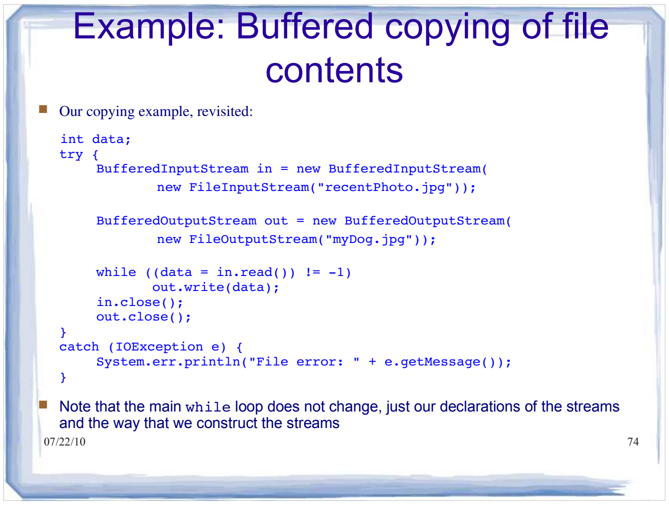
```
BufferedInputStream in =
   new BufferedInputStream(new FileInputStream("recentPhoto.jpg"));
BufferedOutputStream out =
   new BufferedOutputStream(new FileOutputStream("myDog.jpg"));
```



Processing Streams (con't)

Process	Char Streams	Byte Streams
Buffering	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Byte/Character Conversion	InputStreamReader OutputStreamWriter	
Data Conversion	DataInputStream	DataOutputStream
Counting	LineNumberReader	LineNumberInputStream

etc.

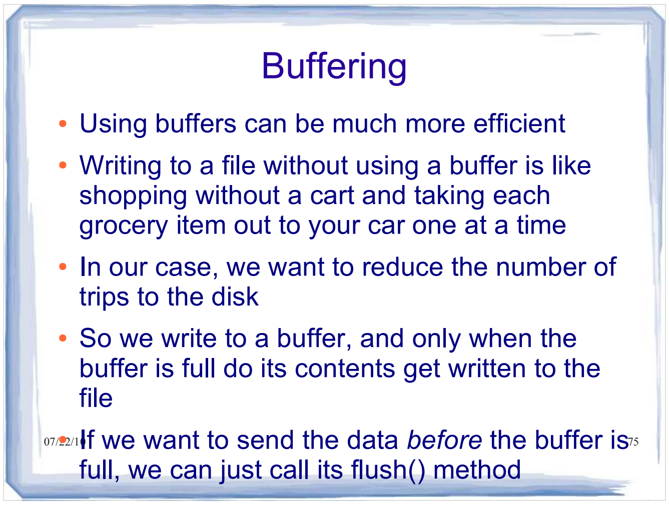


Example: Buffered copying of file contents

- Our copying example, revisited:

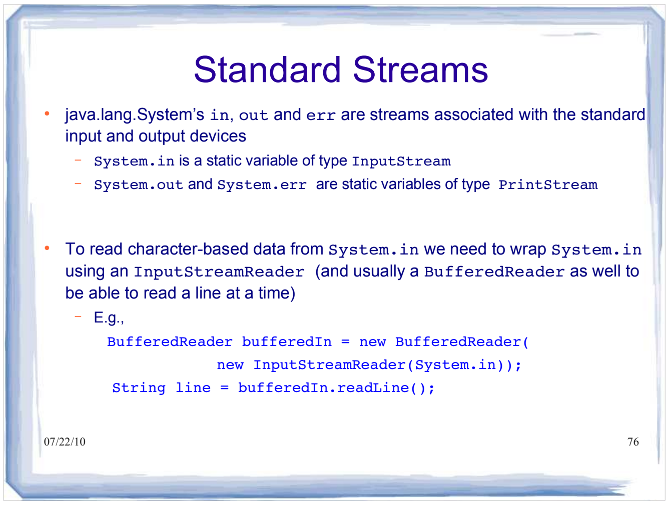
```
int data;
try {
     BufferedInputStream in = new BufferedInputStream(
             new FileInputStream("recentPhoto.jpg"));

     BufferedOutputStream out = new BufferedOutputStream(
             new FileOutputStream("myDog.jpg"));

     while ((data = in.read()) != -1)
            out.write(data);
     in.close();
     out.close();
}
catch (IOException e) {
     System.err.println("File error: " + e.getMessage());
}
```

- Note that the main `while` loop does not change, just our declarations of the streams and the way that we construct the streams



Buffering

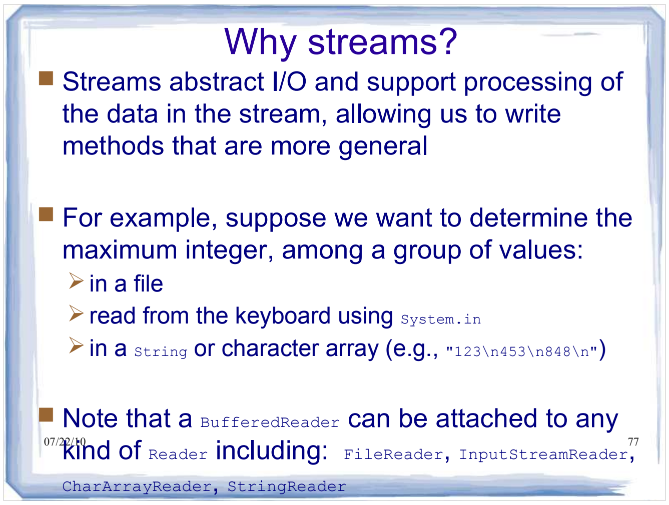
- Using buffers can be much more efficient
- Writing to a file without using a buffer is like shopping without a cart and taking each grocery item out to your car one at a time
- In our case, we want to reduce the number of trips to the disk
- So we write to a buffer, and only when the buffer is full do its contents get written to the file
- If we want to send the data *before* the buffer is full, we can just call its flush() method



Standard Streams

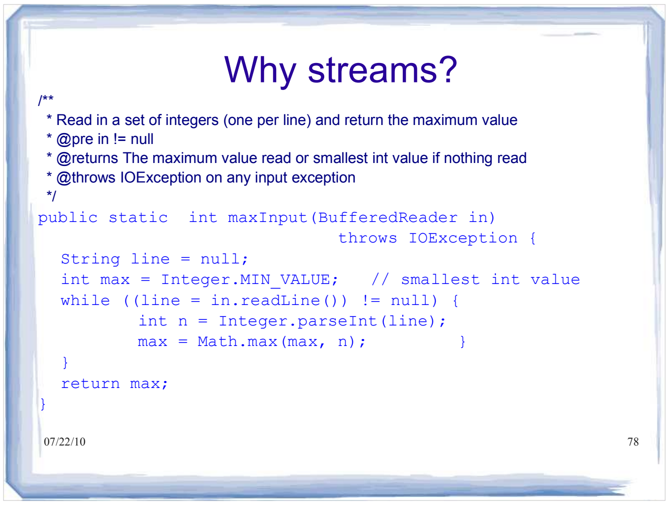
- java.lang.System's `in`, `out` and `err` are streams associated with the standard input and output devices
  - `System.in` is a static variable of type `InputStream`
  - `System.out` and `System.err` are static variables of type `PrintStream`
- To read character-based data from `System.in` we need to wrap `System.in` using an `InputStreamReader` (and usually a `BufferedReader` as well to be able to read a line at a time)
  - E.g.,

```
BufferedReader bufferedIn = new BufferedReader(
              new InputStreamReader(System.in));
 String line = bufferedIn.readLine();
```



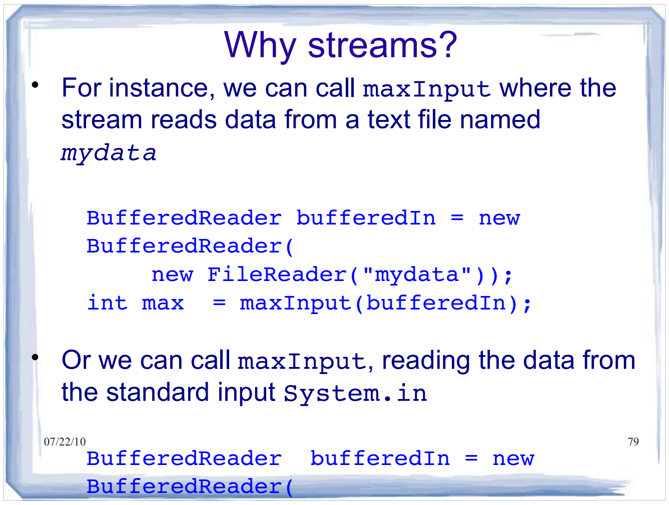
Why streams?

- Streams abstract I/O and support processing of the data in the stream, allowing us to write methods that are more general
- For example, suppose we want to determine the maximum integer, among a group of values:
  - in a file
  - read from the keyboard using `System.in`
  - in a `String` or character array (e.g., `"123\n453\n848\n"`)
- Note that a `BufferedReader` can be attached to any kind of `Reader` including: `FileReader`, `InputStreamReader`, `CharArrayReader`, `StringReader`



Why streams?

```
/**
  * Read in a set of integers (one per line) and return the maximum value
  * @pre in != null
  * @returns The maximum value read or smallest int value if nothing read
  * @throws IOException on any input exception
  */
public static  int maxInput(BufferedReader in)
                                  throws IOException {
   String line = null;
   int max = Integer.MIN_VALUE;    // smallest int value
   while ((line = in.readLine()) != null) {
          int n = Integer.parseInt(line);
          max = Math.max(max, n);          }
   }
   return max;
}
```



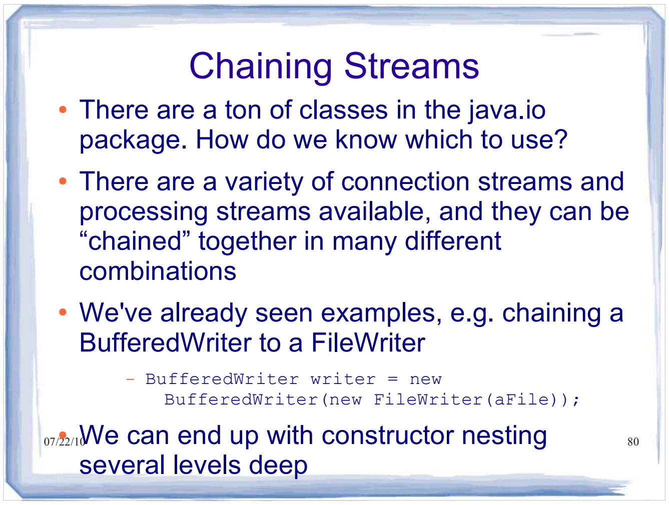
Why streams?

- For instance, we can call `maxInput` where the stream reads data from a text file named *`mydata`*

```
BufferedReader bufferedIn = new
BufferedReader(
     new FileReader("mydata"));
int max  = maxInput(bufferedIn);
```

- Or we can call `maxInput`, reading the data from the standard input `System.in`

```
BufferedReader  bufferedIn = new
BufferedReader(
```



Chaining Streams

- There are a ton of classes in the java.io package. How do we know which to use?
- There are a variety of connection streams and processing streams available, and they can be “chained” together in many different combinations
- We've already seen examples, e.g. chaining a BufferedWriter to a FileWriter
  - `BufferedWriter writer = new BufferedWriter(new FileWriter(aFile));`
- We can end up with constructor nesting several levels deep

Chaining Streams

- Most often, though, you'll use a small handful of classes including the ones we've seen
- Usually with input and output, there is more than one way to accomplish the task
- How do you know which classes can be chained?
  - Check their constructors in the API

Parsing with String split()

- What if our pub quiz file was formatted differently, with one question-answer pair per line, with the Q and A separated by a symbol (e.g. /)?
- We can separate the Q and A by using a String's split() method, which asks for a separator and splits the String when it finds that separator

```
String qa = "Who is the Prime Minister?/Stephen Harper";
String[] result = qa.split( "/" );
for (String token: result)
{
    System.out.println(token);
}
```

Who is the Prime Minister?
Stephen Harper

In-Class Exercise II

- Write code that reads in a pub quiz file in this format (one QA pair per line, separated by a forward slash) and writes out just the answers to a new file

Object Serialization

- Suppose you are writing a program that allows the user to store the names, telephone numbers and addresses of their contacts.
- When the user enters data, they expect it will be available to them the next time they run the program.
- To do this, the program needs to store the data (likely on a hard disk) from one session to the next.
- You can do this easily using object serialization.

Saving Objects

- There are actually two approaches we could take
- If you want to save the data of an object so that it can be used by other programs, you can just write to a plain text file, writing the value of each instance variable for that object in some sort of consistent format
- We now know how to write text to a file
- This data could then be used by a spreadsheet, database or other program

Saving Objects

- But if we want to be able to save an object and reload it in Java at a later date, it is much easier to use serialization

Serialization: Object Streams

- Java's serialization API supports the saving of the state of an object to a sequence of bytes; those bytes can later be used to restore the object
- The ability to save an object is sometimes called "persistent objects"
- Serialization makes it possible to save an object, stop the program, restart it, and then restore the object
- To make objects of a class serializable, you just need to implement the `Serializable` interface. `Serializable` is a **marker interface** (that means it has no methods)
  - E.g., to make the Account class serializable:

```
class Account implements Serializable {
… }
```

Saving an Object

- To save serializable objects in a file we need to
  - associate a `FileOutputStream` with the file
  - wrap an `ObjectOutputStream` around it
  - use `writeObject()` to store the objects sequentially
  - e.g., if `Account` implements Serializable and `a1,a2` are accounts we can save them in the file named ***account.dat***

```
ObjectOutputStream out = new ObjectOutputStream(
       new FileOutputStream("account.dat"));
out.writeObject(a1);
out.writeObject(a2);
```

- Most Java library classes are serializable (but not all)
  - Java takes care of serializing the variables in the

Saving an Object

- Notice another example of chaining
- ObjectOutputStream is chained to FileOutputStream
- FileOutputStream knows how to connect to (and create) a file
- ObjectOutputStream lets us write objects, but it can't directly connect to a file

Chaining

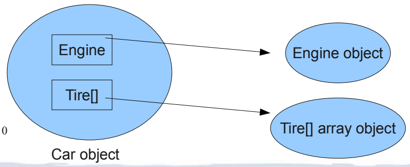
- By the way, why don't we just have a single stream that does exactly what we want?
- For one, it's good OO to have these specialized classes. Each class does one thing well.
- It also gives us a lot of flexibility as far as combining connection and processing streams.

What gets saved?

- All of an object's instance variables get saved
  - Why don't we save methods as well? What makes an object unique?
- This seems straightforward in the case of primitive values like 37 and 70, but what about instance variables that are object references?
- For example, what if our Car object has instance variables that refer to an Engine object and a Tire[] array?

What gets saved?

- Serialization saves the entire object graph. All objects referenced by instance variables, starting with the object being serialized
- So when you save the Car object, the Engine object and Tire objects are also saved



Serialization is “all or nothing”

- Either the entire object graph is serialized correctly or serialization fails
- This poses a problem:
  - We can't serialize an object if it contains an object reference for a class that is not serializable
  - For example, if we design a class Pond that implements Serializable, but it contains objects of the class Duck and Duck does not implement Serializable, then we will get an exception when we try to serialize Pond

Serialization is “all or nothing”

- But what if someone else designed the Duck class and it's not possible for us to make it Serializable?
- One option is to mark it as `transient`
- Anything marked as transient will be skipped over during the serialization process
  - `transient String currentID;`

Transient

- If we mark some instance variables as transient, what happens when we bring the object back to life (deserialize it)?
- Those instance variables will be brought back as `null` (primitives are brought back w/ default values)
- Your options then are to
  - reinitialize that null instance variable back to some default state
  - Or, if it's important that it have the same key values that it had before, then save those values so that you can create a new instance variable that's identical to the original, e.g. a new Duck with the same colour and size

Serialization is “all or nothing”

- Another option is to subclass the non-serializable class and make that subclass implement Serializable

Saving Objects

- If you try to save an object multiple times, the object will only get written once during serialization but there can be multiple references that will be resolved during deserialization

Reading Objects from a File

- To read back in the objects we have saved in a file we need to
  - associate a `FileInputStream` with that file
  - wrap an `ObjectInputStream` around it
  - use `readObject()` to get the objects sequentially, in the order they were saved
  - e.g., to get the first two accounts stored in ***account.dat***

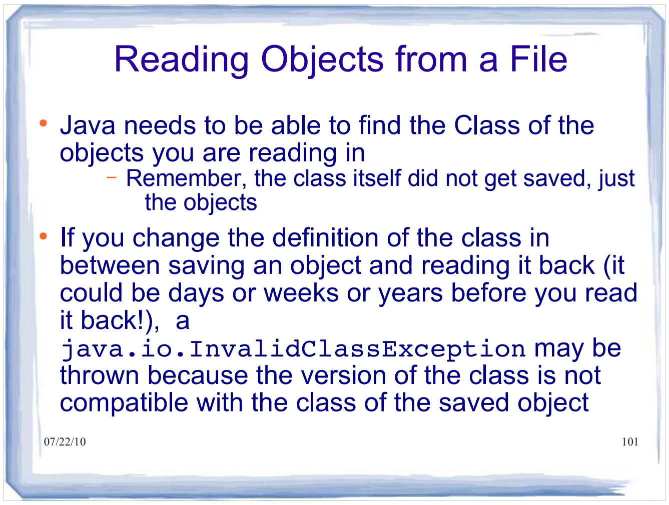
```
ObjectInputStream in =
   new ObjectInputStream(
      new FileInputStream("account.dat"));
Account a1 =  (Account) in.readObject();
Account a2 =  (Account) in.readObject();
```

Reading Objects from a File

- If you try to read back more objects than you wrote, you'll get an exception
- The return type of readObject() is Object, so you need to cast it back to the type you know it really is
- A new object is given space on the heap, but the serialized object's constructor does not run
  - Why not? What might happen to its values?

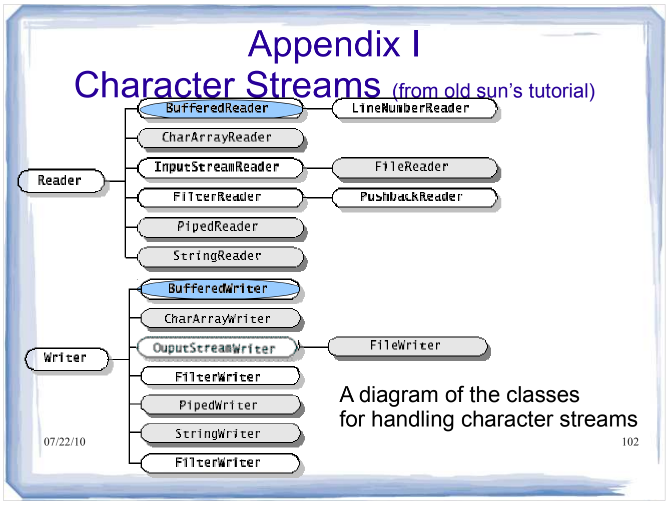
Reading Objects from a File

- However, if the object has a non-serializable class somewhere up its inheritance tree, the constructor for that non-serializable class will run along with any constructors above that (even if they're serializable)



Reading Objects from a File

- Java needs to be able to find the Class of the objects you are reading in
  - Remember, the class itself did not get saved, just the objects
- If you change the definition of the class in between saving an object and reading it back (it could be days or weeks or years before you read it back!), a `java.io.InvalidClassException` may be thrown because the version of the class is not compatible with the class of the saved object



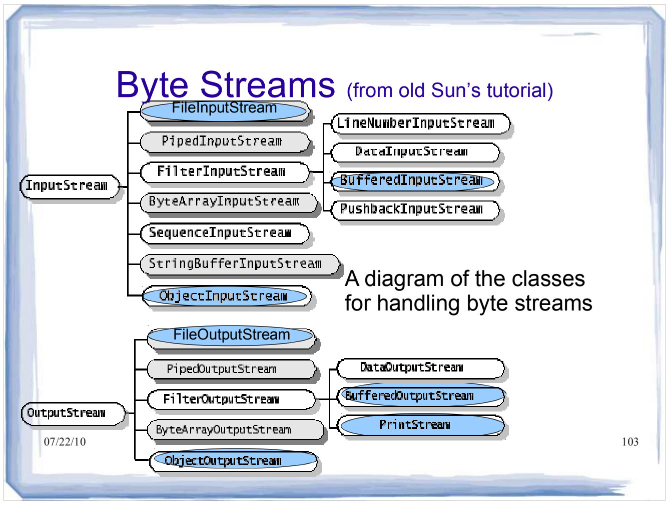
Appendix I

Character Streams (from old sun's tutorial)

Reader
BufferedReader
LineNumberReader
CharArrayReader
InputStreamReader
FileReader
FilterReader
PushbackReader
PipedReader
StringReader

Writer
BufferedWriter
CharArrayWriter
OuputStreamWriter
FileWriter
FilterWriter
PipedWriter
StringWriter
FilterWriter

A diagram of the classes for handling character streams



Byte Streams (from old Sun's tutorial)

InputStream
FileInputStream
PipedInputStream
FilterInputStream
LineNumberInputStream
DataInputStream
BufferedInputStream
PushbackInputStream
ByteArrayInputStream
SequenceInputStream
StringBufferInputStream
ObjectInputStream

A diagram of the classes for handling byte streams

OutputStream
FileOutputStream
PipedOutputStream
FilterOutputStream
DataOutputStream
BufferedOutputStream
PrintStream
ByteArrayOutputStream
ObjectOutputStream