Administrivia

- Java project due Wednesday, 6PM
- questions?

Last time - Java

- Types – primitive, class, array, interface
- I/O
- Packages
  - and which files go where
  - CLASSPATH
- java.lang
  - wrapper classes (for primitive types)
    - String and StringBuffer

Exceptions and Inner Classes

- class Throwable
  - Just another class of objects
  - That can be thrown
  - Two subtypes
    - Exception
    - Error
      - which can always be thrown without being declared

- IOException
- InterruptedException
- RuntimeException
  - can be thrown without being declared (all standard ones are subclasses)
  - NullPointerException
  - IndexOutOfBoundsException
  - NegativeArraySizeException

Exception
Error

- Can be thrown without being declared
- Generally unreasonable to catch and ignore an error
- VirtualMachineError
  - OutOfMemoryError
  - StackOverflowError
- VerifyError
- NoClassDefFoundError

Method throws declarations

- A method declares the exceptions it might throw
  - public void openNext() throws UnknownHostException, EmptyStackException
  
- Must declare any exception the method might throw
  - unless it is caught in the method
  - includes exceptions thrown by called methods

Throw (cont.)

- C++ does run-time check that function doesn’t throw an unexpected exception
  - better for backward compatibility
- Java uses compile-time check
  - forces you to sometimes deal with exceptions you know can’t occur

Creating New Exceptions

- User-defined exception is just a class that is a subclass of Exception

```java
class MyOwnException extends Exception {}
class MyClass {
  void oops() throws MyOwnException {
    if (some_error_occurred) {
      throw new MyOwnException();
    }
  }
}
```

Throwing an Exception/Error

- Create a new object of the appropriate Exception/Error type, and throw it
- If it’s not a subtype of Error or RuntimeException
  - must declare the method throws the exception
- Exceptions thrown are part of return type
  - when overriding a method in a superclass
  - can’t throw anything that would surprise a superclass object

Exception/Error Handling

- All exceptions eventually get caught
- First catch with supertype of the exception catches it
- Don’t catch errors
- finally is always executed

```java
try { if (i == 0) return; myMethod(a[i]); }
catch (ArrayIndexOutOfBoundsException e) {
  System.out.println("a[] out of bounds");
}
catch (MyOwnException e) {
  System.out.println("Caught my error");
}
catch (Exception e) {
  System.out.println("Caught a generic exception");
  throw e;
}
finally { /* stuff to do regardless of whether an exception */
  // or a return taken */
}
```
java.lang.Throwable

- Many objects of class Throwable have a message
  - specified when constructed, as String
  - String getMessage() returns the message
- String toString()
- void printStackTrace()
- void printStackTrace(PrintWriter s)

Inner Classes

- Allow a class to be defined within a class or method
- New class has access to all variables in scope
- Classes can be anonymous
- 4 kinds of inner classes
  - nested classes/interfaces
  - standard inner classes
  - method classes and anonymous classes
- Lots of important details

Nested classes/interfaces

- Not really an inner class
  - not associated with an instance of the outer class
- Defined like a static class method/variable
- Can refer to all static methods/variables of outer class, transparently
- Used to localize/encapsulate classes only used by the outer class
  - information hiding/packaging
- Used to package helper classes/interfaces
  - like a mini-package for each class

Standard Inner Classes

- Defined like a class method/variable
- Each instance associated with an instance of the outer class
- If class A is outer class
  - use A.this to get this for instance of outer class
- Can refer to all methods/variables of outer class
  - transparently
- Can’t have any static methods/variables

Example

```java
public class LinkedList {
    // Keep this private; no one else see the implementation
    private static class Node {
        Object value;  Node next;
        Node(Object v)  { value = v;  next = null;  }  
    }
    // Put here to show that this is the Transformer for LinkedList
    public static interface Transformer {
        public Object transform(Obj ect v); 
    }
    Node head, tail;
    public void applyTransformer (Transformer t) {
        for (Node n = head; n != null; n = n.next)
            n.value = t.transform(n.value);
    }
    public void append(Object v) {
        Node n = new Node(v);
        if (tail == null) head = n;
        else tail.next = n;
        tail = n;  }
    public class(getStringRep implements LinkedList.Transformer {
        public Object transform(Object o) {
            return o. toString (); }
    }
}
```

Example

```java
public class FixedStack {
    Object [] array;
    int top = 0;
    class MyEnum implements java.util.Enumerator {
        int count = top;
        public boolean hasMoreElements () { return count > 0; }
        public Object nextElement () {
            if (count == 0)
                throw new NoSuchElementException (FixedStack); 
            return array[ -- count];  }
        }
        public java.util.Enumerator enumerateAll () {
            return new MyEnum();  }
    }
    public class LinkedStack {
        // Keep this private; no one else see the implementation
        private static class Node {
            Object value;  Node next;
            Node(Object v)  { value = v;  next = null;  }  
        }
        // Put here to show that this is the Transformer for LinkedList
        public static interface Transformer {
            public Object transform(Obj ect v); 
        }
        Node head, tail;
        public void applyTransformer (Transformer t) {
            for (Node n = head; n != null; n = n.next)
                n.value = t.transform(n.value);
        }
        public void append(Object v) {
            Node n = new Node(v);
            if (tail == null) head = n;
            else tail.next = n;
            tail = n;  }
    }
```

Example

```java
public class FixedStack {
    Object []. array;
    int top = 0;
    class MyEnum implements java.util.Enumerator {
        int count = top;
        public boolean hasMoreElements () { return count > 0; }
        public Object nextElement () {
            if (count == 0)
                throw new NoSuchElementException (FixedStack); 
            return array[ -- count];  }
        }
        public java.util.Enumerator enumerateAll () {
            return new MyEnum();  }
    }
```
Method and Anonymous Classes

- Can refer to all methods/variables of outer class
- Can refer to `final` local variables
- Can’t have any static methods/variables
- Method classes defined like a method variable
- Anonymous classes defined in new expression
  
  ```java
  new BaseClassOrInterface() { extensions }
  ```

Anonymous class Example

```java
public class FixedStack {
    Object [] array;
    int top = 0;
    public java.util.Enumerator enumerateOldestK (final int k) {
        class MyEnum implements java.util.Enumerator {
            int pos = 0;
            public boolean hasMoreElements () {
                return pos < k && pos < top; }
            public Object nextElement () {
                if (! hasMoreElements () )
                    throw new NoSuchElementException (“FixedStack”);
                return array[pos++];  }
        }
        return new MyEnum (); }
}
```
Writing Multi-threaded Code

- Need to control which events can happen simultaneously
  - e.g., update and display methods for a class
- Usually only covered in OS/DB courses
  - so few programmers have lots of training
- Can get inconsistent results or deadlock
  - problems often not easily reproduced
- Easy to get multi-threading, without trying
  - Graphical User Interfaces (GUI’s)
  - Remote Method Invocation

Extending class Thread

- Can build a thread class by extending `java.lang.Thread`
- Must supply a `public void run()` method
- Start a thread by invoking the `start()` method
- When a thread starts, executes `run()`
- When `run()` returns, thread is finished/dead

Simple thread methods

- `void start()`
- `boolean isAlive()`
- `void setDaemon(boolean on)`
  - if only daemon threads running, VM terminates
- `void setPriority(int newPriority)`
  - thread scheduler might respect priority
- `void join()` throws `InterruptedException`
  - waits for a thread to die/finish

Simple static thread methods

- Apply to thread invoking the method
  - `void yield()`
  - `void sleep(long milliseconds)`
    throws `InterruptedException`
  - `Thread currentThread()`

Runnable interface

- Extending `Thread` means can’t extend any other class
- Instead implement `Runnable`
  - declares that the class has a `void run()` method
- Can construct a new `Thread`
  - and give it an object of type `Runnable` as an argument to the constructor
  - `Thread(Runnable target)`
  - `Thread(Runnable target, String name)`

Thread Example

```java
public class ThreadDemo implements Runnable {
    public void run() {
        for (int i = 5; i > 0; i --) {
            System.out.println(i);
            try { Thread.sleep(1000);  }
            catch( InterruptedException e) { };  }
        System.out.println("exiting " + Thread.currentThread());
    }
    public static void main(String [] args) {
        Thread t = new Thread(new ThreadDemo(),"Demo Thread");
        System.out.println("main thread: " + Thread.currentThread());
        System.out.println("Thread created: " + t);
        t.start();
        try { Thread.sleep(3000);  }
        catch (InterruptedException e) { ];
        System.out.println("existing " + Thread.currentThread());
    }
}
```
InterruptedException

• A number of thread methods throw it
  − really means: wakeUpCall
• interrupt() sends a wakeUpCall to a thread
• Won’t disturb the thread if it is working
  − but if thread attempts to sleep
  − it will get immediately woken up
• Will also wake up the thread if it is already asleep
• Thrown by sleep(), join(), wait()

Be careful with threads

• Under some implementations of JVM
  − a thread stuck in a loop will never yield by itself
• Preemptive scheduling would guarantee it
  − but not supported on all platforms
• Put yield() into loops
• I/O has highest priority, so should be able to get time on CPU

CMSC 433, Spring 2001 - Alan Sussman

Administrivia

• Java #2 project on multi-threading posted
  − due Wednesday, March 14, 6PM
• Exam #1 coming up in 1 week
  − on C++ and basic Java
• Office hours today shifted earlier by 1 hour
  − from 3-4, instead of 4-5
• Readings
  − Threads – Chapter 14 in Eckel

Another thread example

class UnSyncTest extends Thread {
    String msg;
    public UnSyncTest(String s) {
        msg = s; start();
    }
    public void run() {
        System.out.println("[");
        try { Thread.sleep(1000); }
        catch(InterruptedException e) {}
        System.out.println("]");
    }
    public static void main(String [] args) {
        new UnSyncTest("Hello");
        new UnSyncTest("UnSynchronized");
        new UnSyncTest("World");
    }
}
Daemon threads

- A thread can be marked as a daemon thread
- By default, thread acquires status of thread that spawned it
- When no threads running except daemons
  - program execution terminates

Synchronization issues

- Locks
- synchronized statements and methods
- wait and notify
- Deadlock

Locks

- All objects can be locked
- Only one thread can hold a lock on an object
  - other threads block until they can acquire it
- If your thread already holds a lock on an object
  - can lock it a second time
  - object not unlocked until both locks released
- No way to only attempt to acquire a lock

Synchronized methods

- A method can be synchronized
  - add synchronized modifier before return type
- Obtains a lock on object referenced by this, before executing method
  - releases lock when method completes
- For a static synchronized method
  - locks the class object

Synchronized statement

- synchronized (obj) { statements }
- Obtains a lock on obj before executing statements in block
- Releases lock once block completes
- Provides finer grained control than synchronized method
- Allows locking arguments to a method

Synchronization example

```java
class SyncTest extends Thread {
    String msg;
    public SyncTest(String s) {
        msg = s;
        start();
    }
    public void run() {
        synchronized (SyncTest.class) {
            System.out.print("[");
            try { Thread.sleep(1000); } catch (InterruptedException e) {};
            System.out.println ("]");
        }
    }
    public static void main(String [] args) {
        new SyncTest("Hello");  new SyncTest("Synchronized");
        new SyncTest("World");
    }
}
```
Wait and Notify

- Must be called inside `synchronized` method or block of statements
- `a.wait()`
  - gives up lock(s) on `a`
  - adds thread to wait set for `a`
  - suspends thread
- `a.wait(int m)`
  - limits suspension to `m` milliseconds

Wait and Notify (cont.)

- `a.notify()` resumes one thread from `a`’s wait list
  - and removes it from wait set
  - no control over which thread
- `a.notifyAll()` resumes all threads on `a`’s wait list
  - resumed threads must reacquire lock before continuing
- `wait` doesn’t give up locks on any other objects
  - e.g., acquired by methods that called this one

Producer/Consumer Example

```java
public class ProducerConsumer {
    private boolean ready = false;
    private Object obj;
    public ProducerConsumer() { }
    public ProducerConsumer(Object o) {
        obj = o;  ready = true;
    }
    synchronized void produce(Object o) {
        while (ready) wait();
        obj = o;  ready = true;
        notifyAll();
    }
    synchronized Object consume() {
        while (!ready) wait();
        ready = false;
        notifyAll();
        return obj;
    }
}
```

Changed example

```java
synchronized void produce(Object o) {
    while (ready) {
        synchronized (empty) {
            try {empty.wait(); }
            catch (InterruptedException e) { }
        }
        obj = o;  ready = true;
        synchronized (full) {
            full.notify();
        }  }
}
synchronized Object consume() {
    while (!ready) {
        synchronized (full) {
            try { full.wait(); }
            catch (InterruptedException e) { }
        }
        Object o = obj;  ready = false;
        synchronized (empty) {
            empty.notify();
        }
    }
}
```

A Better Solution

```java
synchronized void produce(Object o) {
    while (ready) {
        synchronized (empty) {
            try {empty.wait(); }
            catch (InterruptedException e) { }
        }
    }
    obj = o;  ready = true;
    synchronized (full) {
        full.notify();
    }
}
synchronized Object consume() {
    while (!ready) {
        synchronized (full) {
            try { full.wait(); }
            catch (InterruptedException e) { }
        }
        Object o = obj;  ready = false;
        synchronized (empty) {
            empty.notify();
        }
    }
}
```

Use two objects, `empty` and `full`, to allow two different wait sets.