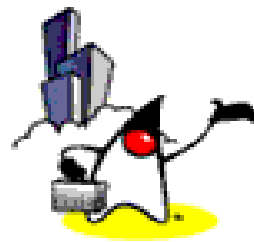
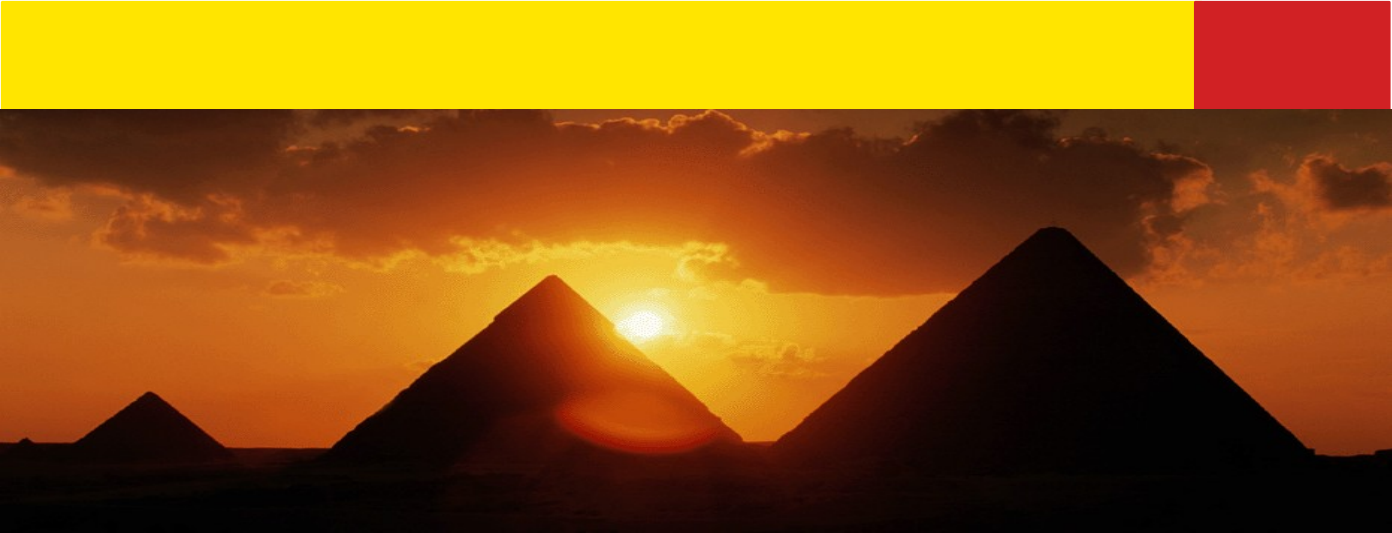


# **Java Programming with Performance in mind**

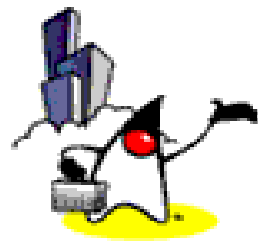


# Common Programming Pitfalls

- Primitive and Objects
- Abuse of the **String** class
- Creating intermediate objects
- Mutable return types
- Using the wrong collections
- Array copy



# Primitives vs. Objects



# Primitives vs. Objects

```

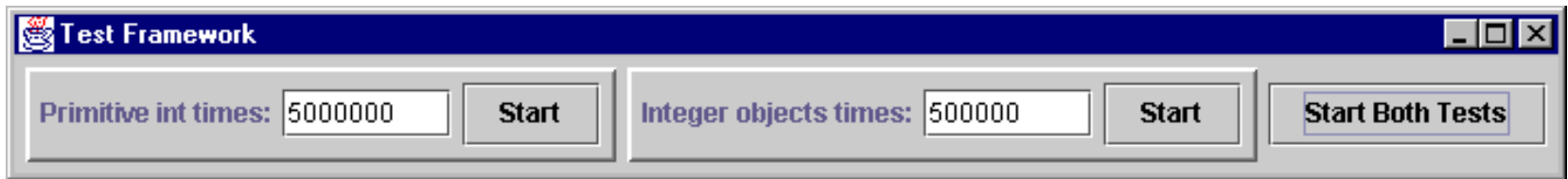
protected void testOne(int n) {
    fPrimArray = new int[n];
    fPrimSum = 0;
    for (int i = 0; i < n; i++) {
        fPrimArray[i] = i;           // insert
        fPrimSum += fPrimArray[i];  // get
    }
}

//-----
protected void testTwo(int n) {
    fObjectArray = new Integer[n];
    fObjectSum = 0;
    for (int i = 0; i < n; i++) {
        fObjectArray[i] = new Integer(i);
        fObjectSum += fObjectArray[i].intValue();
    }
}

```

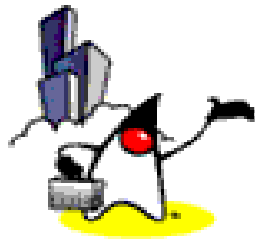
# Primitives vs. Objects

- Primitive `int` performs almost 3 times fast than `Integer` object
  - > Tested on JDK1.5.0 for running the loops 1,000,000 times
- Overall "java -server" gives better result than "java"





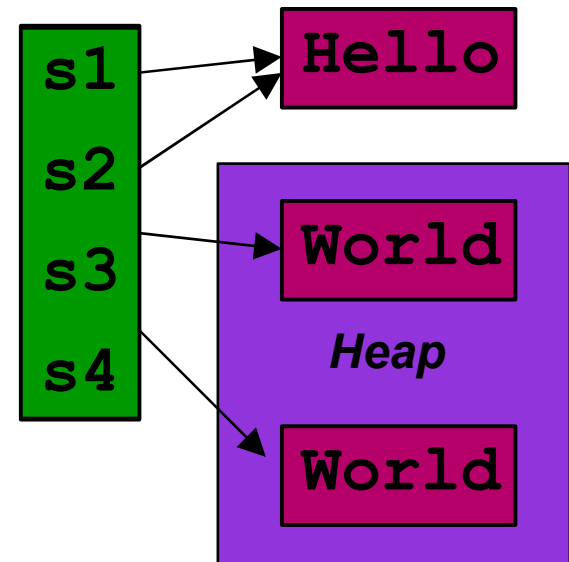
# String vs. StringBuffer



# Two Kinds of String Objects

- String **literals** are unique to each class
  - > Only one copy of each unique string
- Heap-based strings are less efficient

```
String s1 = "Hello";  
String s2 = "Hello";  
String s3 = new String("World");  
String s4 = new String("World");  
  
boolean b1 = (s1 == s2);  
boolean b2 = (s3 == s4);  
boolean b3 = s3.equals(s4);
```



# String vs. StringBuffer vs. StringBuilder

- **String** objects are immutable.
- **String** concatenation creates multiple, intermediate representations - not efficient
- Use mutable **StringBuilder** for all cases if no synchronization is needed (single thread)
- Use mutable **StringBuffer** if needs synchronization

```
String badStr = new String();
StringBuilder goodBuff = new StringBuilder(1000);
for (int i = 0; i < 1000; i++) {
    badStr += myArray[i]; // creates new strings
    goodBuff.append(myArray[i]); // same buffer
}
String goodStr = new String(goodBuff);
```

# Methods May Return Copies

- Mutable objects can be changed.
- Some JDK methods return copies of mutable objects to ensure encapsulation.
  - > e.g., Dimension

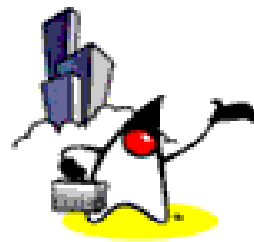
```
Public void paint(Graphics g) {  
    g.fillRect(0,0, getSize().width,  
               getSize().height);  
}
```

```
public void paint(Graphics g) {  
    Dimension size = getSize();  
    g.fillRect(0,0, size.width,  
               size.height);  
}
```





# Java Collections



# Java Collections

- **Vector** and **Hashtable** are **synchronized** on all methods
- You pay for safety whether or not you need it
  - > No need if only one thread accessing the collection
- Java collection classes not synchronized by default (but can be synchronized).
  - > **ArrayList**, **LinkedList** replace **Vector**
  - > **HashSet**, **HashMap** replace **Hashtable**
  - > Synchronization: use of wrapper classes.
    - > a static factory method:
      - `Collections.synchronizedList(new ArrayList())`

# Benchmark: ArrayList & LinkedList

```
// do also for LinkedList, Vector and Hashtable  
// timing code not shown
```

```
List list = new ArrayList();  
final int kNum = 50000;  
for (int i = 0; i < kNum; i++)  
    list.add(new Integer(i));  
  
for (int i = 0; i < kNum; i++)  
    Object result = list.get(i);  
  
for (int i = 0; i < kNum; i++)  
    list.remove(0);
```

# Benchmark Results: 50,000

- Adding new elements is fast for both types of List.
- **ArrayList:**  
 Random look up using `get ()` is fast.  
 Removing elements is slow.
- **LinkedList:**  
 Removing or editing elements in the list is fast.  
 Lookup using `get ()` is very slow.
- **HashSet/HashMap:** overall very fast.
- **Vector behaves the same as ArrayList**
- **Hashtable:** overall very fast.

(sec)	ArrayList	LinkedList	HashSet	HashMap	Vector	Hashtable
Add	0.11	0.23	0.63	0.62	0.11	0.37
Get	0.01	160	0.03	0.07	0.01	0.06
Delete	5.52	0.02	0.13	0.13	5.57	0.08

# Copying Array Elements

- Two ways copy elements from one arrays to another

```
Public class ArrayCopier {  
  
    private double[]    fValues;  
    private String[]    fLabels;  
    private final int    kNum;  
    //-----  
    public ArrayCopier(int number)    {  
        kNum = number;  
        fValues = new double[kNum];  
        fLabels = new String[kNum];  
        for (int i = 0; i < kNum; i++) {  
            fValues[i] = (double) (i*i);  
            fLabels[i] = "" + i;  
        }  
    }  
}
```

# Two Ways To Copy Elements

```
//-----  
public void useLoopCopy() {  
    double[] copyD = new double[kNum];  
    String[] copyS = new String[kNum];  
    for (int i = 0; i < kNum; i++) {  
        copyD[i] = fValues[i];  
        copyS[i] = fLabels[i];  
    }  
}  
  
//-----  
public void useArraycopy() {  
    double[] copyD = new double[kNum];  
    String[] copyS = new String[kNum];  
    System.arraycopy(fValues, 0, copyD, 0, kNum);  
    System.arraycopy(fLabels, 0, copyS, 0, kNum);  
}
```

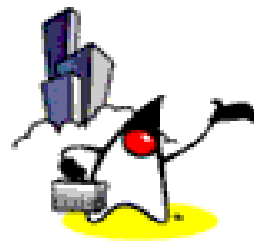
# Array Copy Results

500, 000	Java -server (sec)	Java (sec)
Ceate arrays	0.58	0.58
For loop	0.03	0.02
Arraycopy()	0.01	0.01

- Use `System.arraycopy(...)` to efficiently copy elements from one array to another
- Use static methods in `Arrays`
  - > `equals()`, `fill()`, `sort()`



# Try-Catch



# Try/Catch Blocks Are Free (or Not)

- Myth: try/catch blocks are free?
- Reality is more complex
  - Defeats bounds-check opts on some JVMs
  - Free otherwise

vectorsum

```
for( int i=0; i < A.length; i++ )  
    sum += A[i];
```

try/catch

```
for( int i=0; i < A.length; i++ )  
    try { sum += A[i]; }  
    catch( Error e ) {}
```

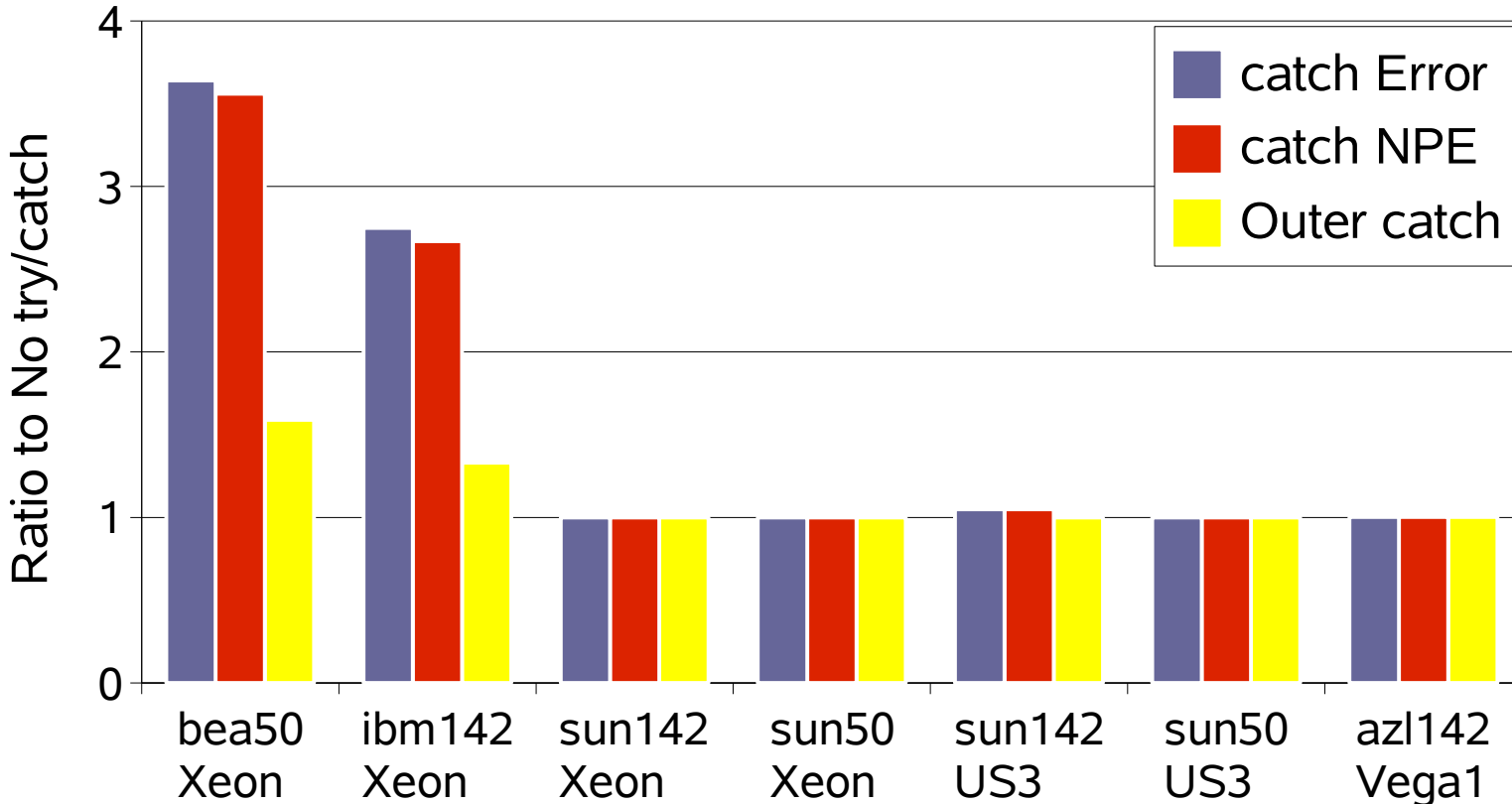
NPE

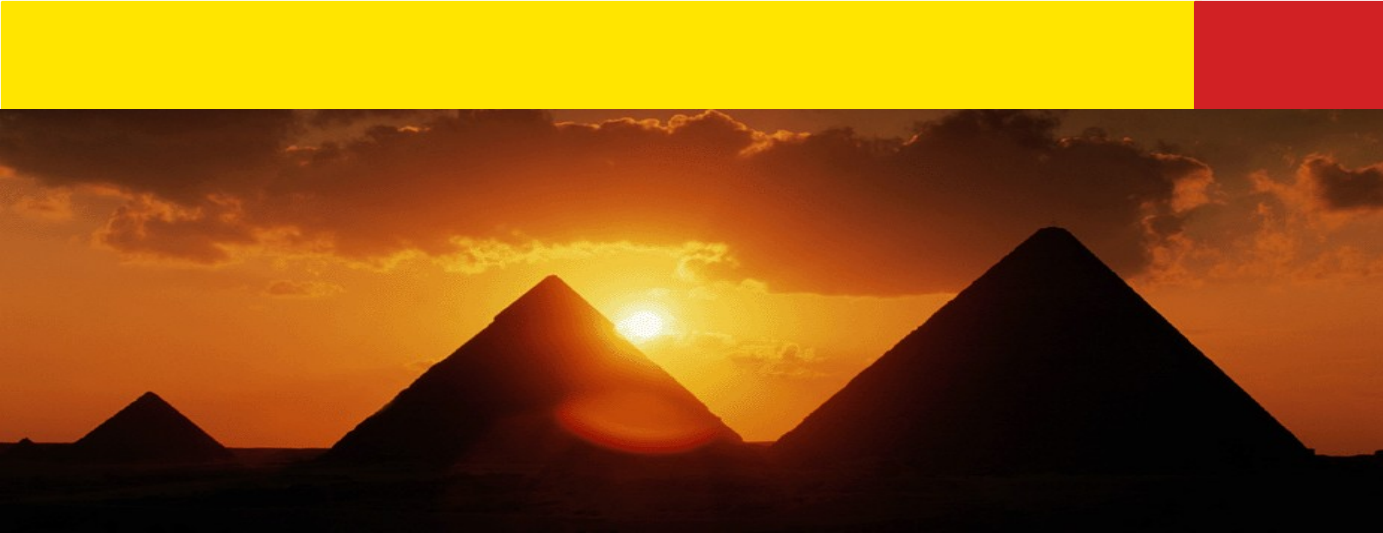
```
catch( NullPointerException e ) {}
```

# Try/Catch Is Not Free

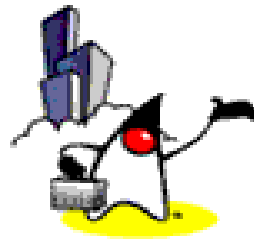
Array Intensive

smaller is faster





# Exception Handling



# Exception Handling

- Use exceptions to make your code robust.
  - > Exception handling is one of Java's best features.
- Use exceptions to handle unexpected conditions.
  - > Conditions beyond your control.
    - > IOException, RemoteException, SQLException, ConnectException.
- Special case: do not use exceptions when normal program logic will suffice.

**try** is not free

**throw** is expensive.

# E.g., Array Out of Bounds

```

private    int[]  fArray = new int[100000];

protected void testOne(int n) {
    fSum = 0;
    for (int i = 0; i < n; i++) {
        if (i < fArray.length)
            fSum += fArray[i];
    }
}

protected void testTwo(int n) {
    fSum = 0;
    for (int i = 0; i < n; i++) {
        try {
            fSum += fArray[i];
        } catch (IndexOutOfBoundsException e) {}
    }
}

```

- What happens when n is larger than the array?

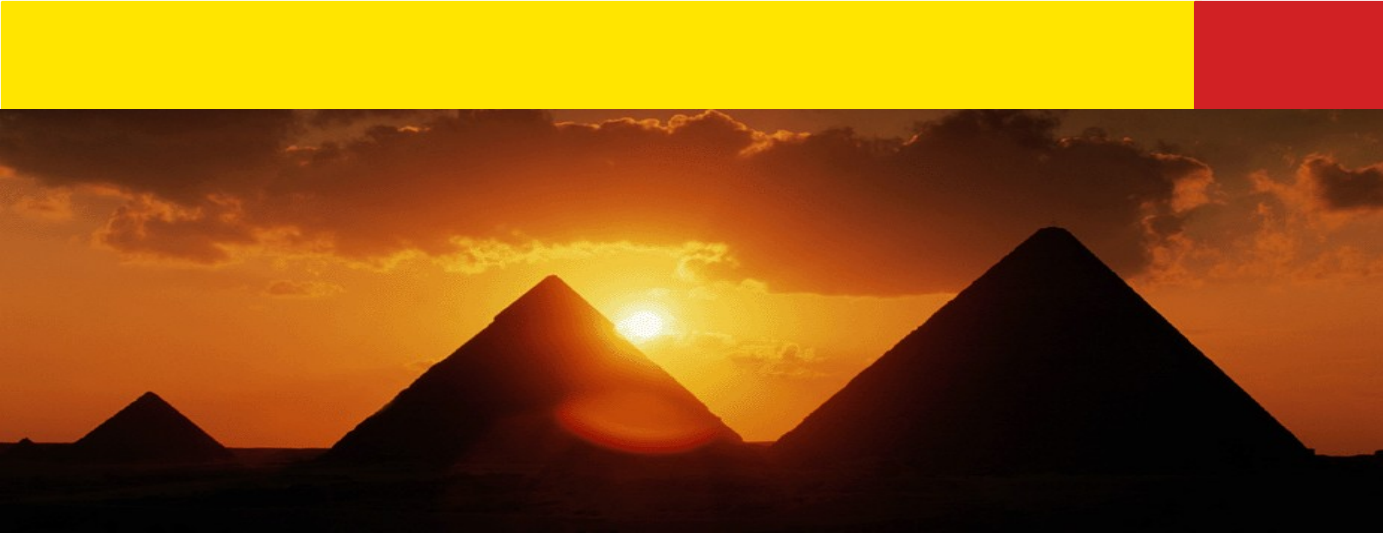
# Logic vs. Exceptions

- Test used array with 1,000,000 primitive ints.
- When exceed array bounds ("# overrun"):
  - > logic faster than exceptions (HotSpot).

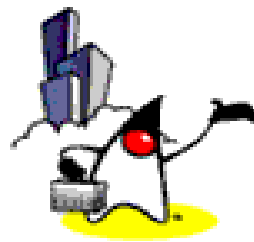


# Other Issues

- Thread Synchronization
- Writing to console window
- Accessing SQL databases using JDBC
- Perceived performance



# Thread Synchronization



# Thread Synchronization

- Spraying around the `synchronized` keyword doesn't ensure your code is thread-safe
- Synchronization has a cost
  - > Methods execute more slowly, because acquiring and releasing a monitor lock is expensive
  - > Synchronization may cause deadlock
- Use `synchronized` key word:
  - > only for critical section
  - > hold lock as short a time as possible

# Synchronize Critical Section

- E.g., **shared resource** is an customer account. Certain methods called by multiple threads.
- **Hold monitor lock for as short a time as possible.**

```
synchronized double getBalance() {
    Account acct = verify(name, password);
    return acct.balance;
}
```

Lock held for long time

```
double getBalance() {
    synchronized (this) {
        Account acct = verify(name, password);
        return acct.balance;
    }
}
```

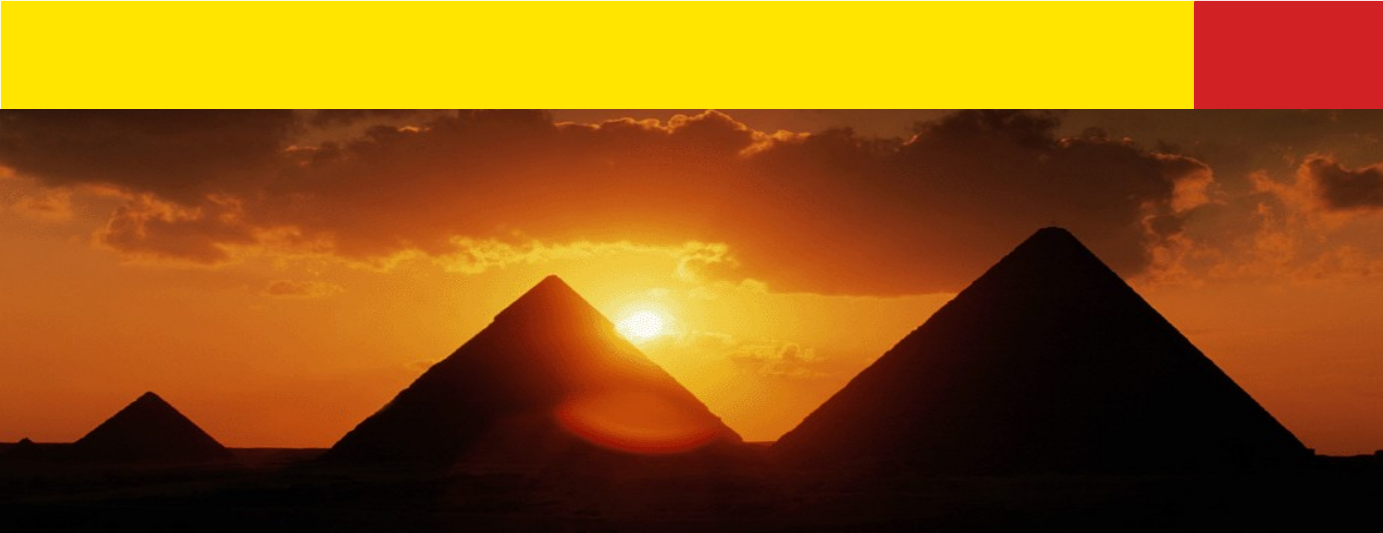
Equivalent to above

Current object is locked

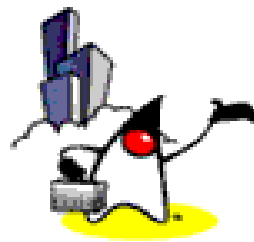
```
double getBalance() {
    Account acct = verify(name, password);
    synchronized (acct) { return acct.balance; }
}
```

Better

Only acct object is locked – for shorter time



# Writing to Console



# Be Careful with println()

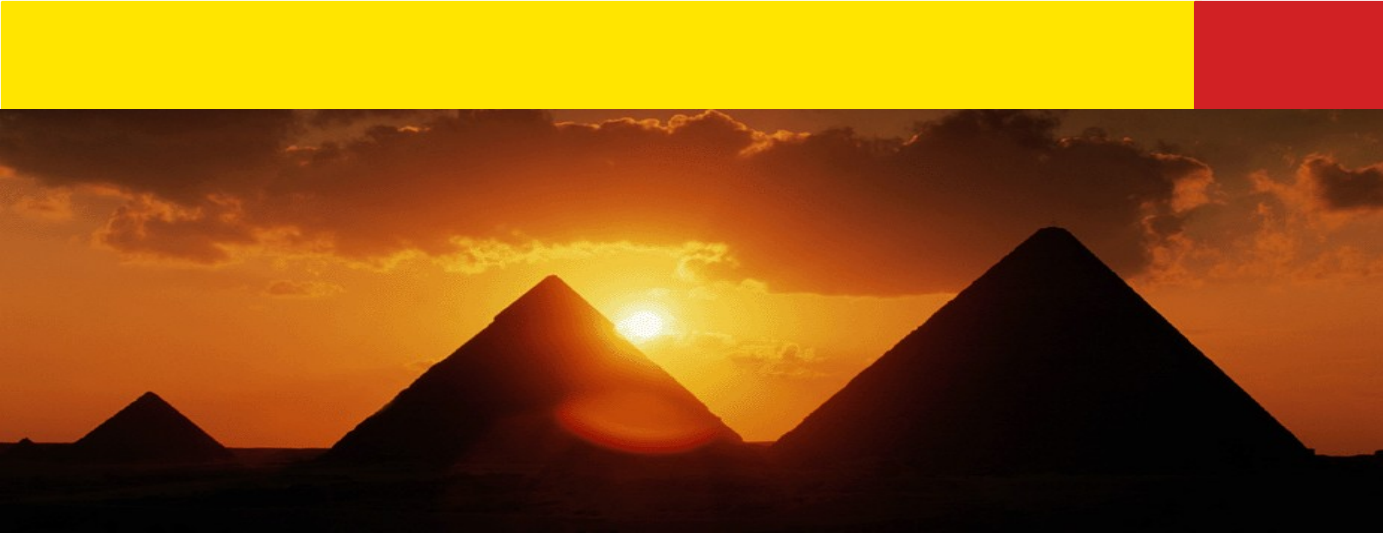
- Hiding console windows when not needed
- Control the debugging code
  - pass debug option on command line
  - test debug boolean before writing to console
  - set `debug = false` for shipping code

```
//use boolean to control debugging message
static final boolean debug = true;
if (debug) {
    debug("debugBar: " + X + Y + "error message");
}

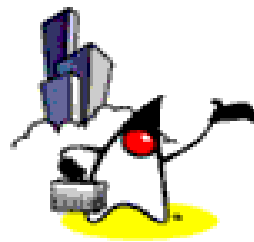
public static void debug(String a) {
    System.err.println(a); }
}
```

# Using JDBC

- Use JDBC `PreparedStatement` for SQL commands instead of ordinary statements
- Combining a number of related SQL operations into a single JDBC transactions
  - > set “auto commit” mode to false.
    - > `setAutoCommit(false)`
  - > SQL statement executes are bundled into a single transaction.
    - > `ExecuteUpdate()`



# Byte code reduction



# Byte Code Reduction

- Package: compressed Jar files
- Optimize: remove unused code
- Obfuscate: E.g.,
  - > `getCustomerAddress()` -> `a()`
- Vendor claims Jar file reductions of 30 - 70 % are common

# Byte Code Size Reducers

- Byte Code Size Reducers
  - > [DashO-Pro](http://www.preemptive.com) <www.preemptive.com>
  - > [Jax](http://www.alphaworks.ibm.com/formula/jax) <www.alphaworks.ibm.com/formula/jax>
  - > [Jshrink](http://www.e-t.com) <www.e-t.com>
  - >
- Byte Code Obfuscators
  - > [DashO-OE](http://www.preemptive.com) <www.preemptive.com>
  - > [Jshrink](http://www.e-t.com) <www.e-t.com>
  - > [SourceGuard](http://www.4thpass.com) <www.4thpass.com>