CS151

Exceptions
Analysis

Exceptions

- Unexpected events during execution
  - unavailable resource
  - unexpected input
  - logical error
- In Java, exceptions are objects
- 2 options with an Exception
  - “Throw” it
    - this says that the exception must be handled elsewhere
  - “Catch” it.
    - handle the problem here and now

Catching Exceptions

```
try {
    guardedBody
} catch (exceptionType1  variable1) {
    remedyBody1
} catch (exceptionType2  variable2) {
    remedyBody2
} ...
    ...
```

- Exception handling
  - `try-catch`
- An exception is caught by having control transfer to the matching `catch` block
- If no exception occurs, all `catch` blocks are ignored

Throwing Exceptions

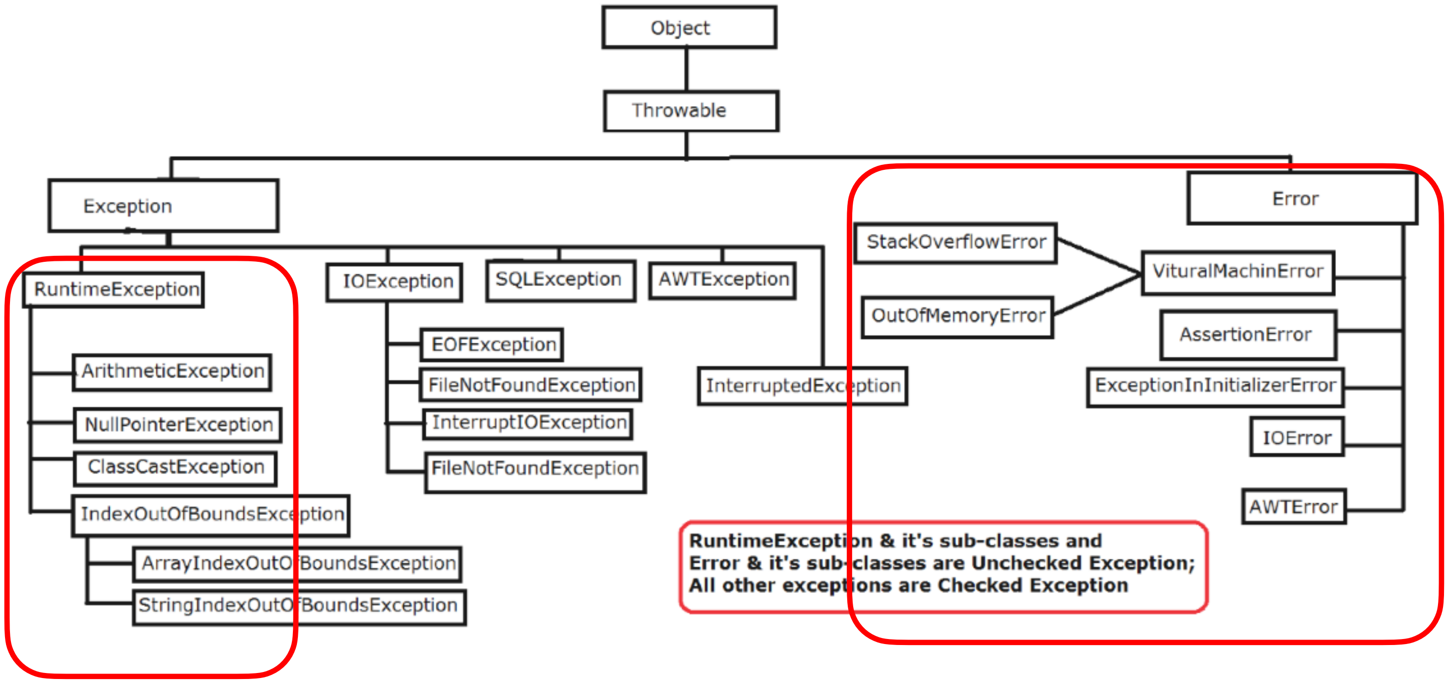
- An exception is thrown
  - implicitly by the JVM because of errors
  - explicitly by code
- If your code throws an exception it **must** catch that exception somewhere else
- Method signature – `throws`

```
public static int parseInt(String
s) throws NumberFormatException
```

Exceptions Example

```
public class ExceptThrower {
    public int divv(int numer, int denom) {
        try {
            return numer / denom;
        } catch (ArithmeticException e) {
            System.err.println("Caught in Func " + e);
        }
        return 0;
    }
    public int divvTh(int numer, int denom) throws ArithmeticExc
       return numer / denom;
    }
    public static void main(String[] args) {
        ExceptThrower except = new ExceptThrower();
        except.divv(2, 0);
        try {
            except.divvTh(4,0);
        } catch (ArithmeticException ae) {
            System.err.println("Caught in Main " + ae);
        }}}
```

Java's Exception Hierarchy



Running Time

- The run time of a program depends on
  - efficiency of the algorithm/implementation
  - size of input
  - what else?
- The running time typically grows with input size
- How do you measure running time?
  - CPU usage?
    - Reliability?

Timing Code

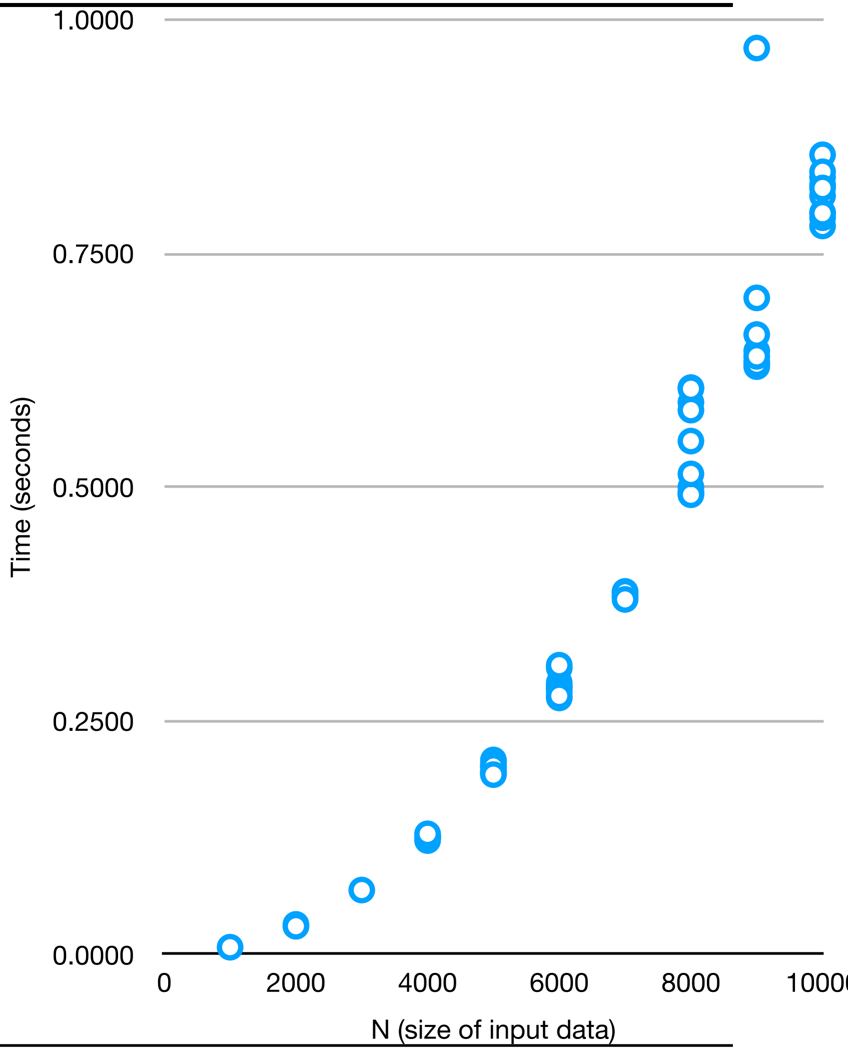
```
public class Timer {
    private static final int REPS = 2; // number of trials
    private static final int NANOS_SEC = 1000000000; // nanosec per sec

    public double doSomething(int[] data) {
        double k = 0;
        for (long i = 0; i < data.length; i++) {
            for (long j = 0; j < data.length; j++) {
                k += Math.sqrt(i * j);
            }
        }
        return k;
    }

    public static void main(String[] args) {
        Timer timer = new Timer();
        long data[] = new long[REPS];
        for (int j = 1000; j < 65000; j = j*2) {
            for (int i = 0; i < REPS; i++) {
                long start = System.nanoTime();
                timer.doSomething(new int[j]);
                long finish = System.nanoTime();
                data[i] = (finish - start);
                System.out.println(String.format("%d %.4f", j, (double) (finish - start) /
NANOS_SEC));
           }}}}
```

Experimental Studies

- Write a program implementing the algorithm
- Run it with different input sizes and compositions
- Record times and plot results



Limitation of Experiments

- You have to implement the algorithm
- You have to generate inputs that represent all cases
- Comparing two algorithms requires exact same hardware and software environments
  - Even then timing is hard
    - multiprocessing
    - file i/o

Theoretical Analysis

- Use a high-level description of algorithm
  - pseudo-code
- Running time as a function input size, n
- Ignore other details of the input
- Independent of the hardware/software environment

Primitive Operations

- Basic computations
  - * / + -
- Comparisons
  - ==, >, <
- Setting
  - x=y
- Assumed to take constant time
  - exact constant is not important
  - Because constant is not important, can do more than just this list

Example
Time required to compute an average

```
public double allAverage(long[] data){
        double res = 0;
        for (int i=0; i<data.length; i++)
        {
            res = res+data[i];
        }
        return res/data.length;
    }
public double posAverage(long[] data) {
        double res = 0;
        long pCount = 0;
        for (int i=0; i<data.length; i++) {
            long datum=data[i];
            if (0<datum) {
                res = res+datum;
                pCount=pCount+1;
            }
        }
        return res/pCount;
    }
```

How many operations? (In terms of the length of data)

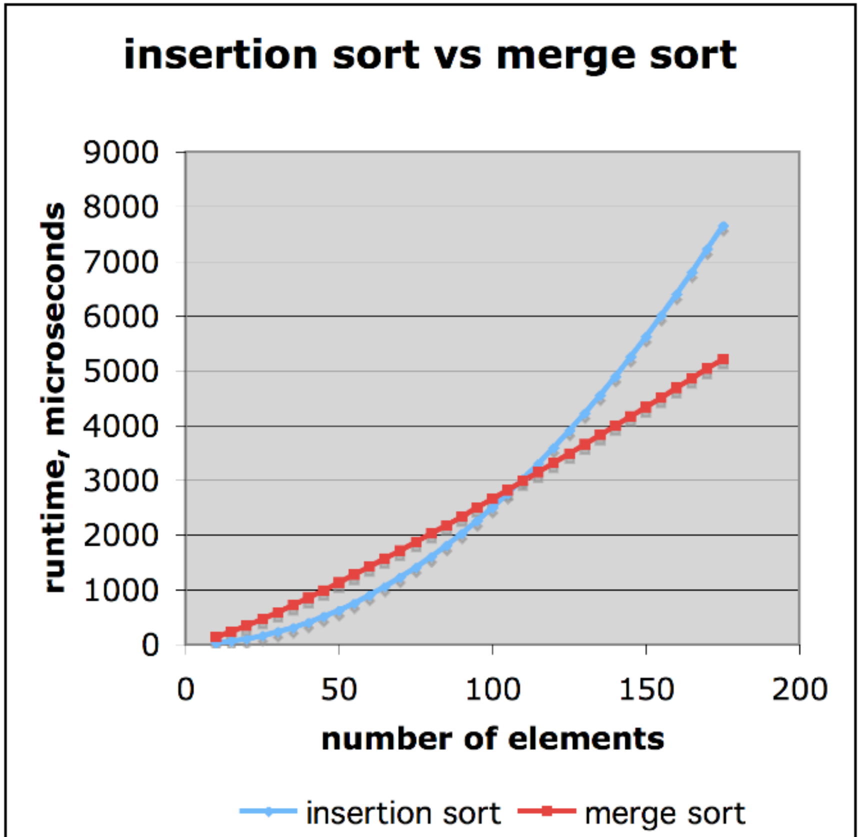
Estimate Running Time

- `allAverage` executes 5N+3 operations
- `posAverage` executes a total of 9N+3 primitive operations in the worst case, 5N+3 in the best case.
- Let **a** be the fastest primitive operation time, **b** be the slowest primitive operation time
- Let $T(n)$ denote the worst-case time of allAverage.
  Then:
  $a(5n+3) < T(n) < b(5n+3)$
- $T(n)$ is bounded by two functions
  - both are linear in terms of n

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of both algorithms.

Comparison of Two Algorithms



- insertion sort: $n^2/4$
- merge sort: $2n\lg n$
- suppose $n=10^8$
  - insertion sort: $10^8 * 10^8/4 = 2.5 * 10^{15}$
  - merge sort: $10^8 * 26 * 2 = 5.2 * 10^9$
  - or merge sort can be expected to be about 10^6 times faster
  - so if merge sort takes 10 seconds then insertion sort takes about 100 days

Asymptotic Notation

- Provides a way to simplify analysis
- Allows us to ignore less important elements
  - constant factors
- Focus on the largest growth of n
  - Focus on the dominant term

How do these functions grow?

- $f_1(x) = 43n^2 \log^4 n + 12n^3 \log n + 52n \log n$
- $f_2(x) = 15n^2 + 7n \log^3 n$
- $f_3(x) = 3n + 4 \log_5 n + 91n^2$
- $f_4(x) = 13 \cdot 3^{2n+9} + 4n^9$

Big O

- Constant factors are ignored
- Upper bound on time
- Goal is to have an easily understood summary of algorithm speed
  - not implementation speed

Sublinear Algorithms

- O(1)
  - runtime does not depend on input
- $O(\lg_2 n)$
  - algorithm constantly halves input

Linear Time Algorithms: $O(n)$

- The algorithm's running time is at most a constant factor times the input size
- Process the input in a single pass spending constant time on each item
  - max, min, sum, average, linear search
- Any single loop

$O(nlogn)$ time

Frequent running time in cases when algorithms involve:

- Sorting
  - only the “good” algorithms
    - e.g. quicksort, merge sort, …

Quadratic Time: $O(n^2)$

- Nested loops, double loops
  - The `doSomething` algorithm
- Processing all pairs of elements
- The less-good sorting algorithms
  - e.g., insertion sort

Slow!!!! Times

- polynomial time: $O(n^k)$
  - All subsets of n elements of size k
- exponential time: $O\left(2^n\right)$
  - All subsets of n elements (power set)
- factorial time: $O(n!)$
  - All permutations of n elements

Algorithm Run Times

N	log(n)	n	n log(n)	n*n	n*n*n	n!
10	3	10	33	100	1000	10^5
100	7	100	664	10000	10^6	10^94
1000	10	1000	9966	10^6	10^9	10^1435
10000	13	10000	132877	10^8	10^12	10^19355
100000	17	100000	1660964	10^10	10^15	10^(10^6)

Analyzing StuffBag

- add
- remove one
- count
- remove all of X

- Can these times be improved?
  - at what cost?