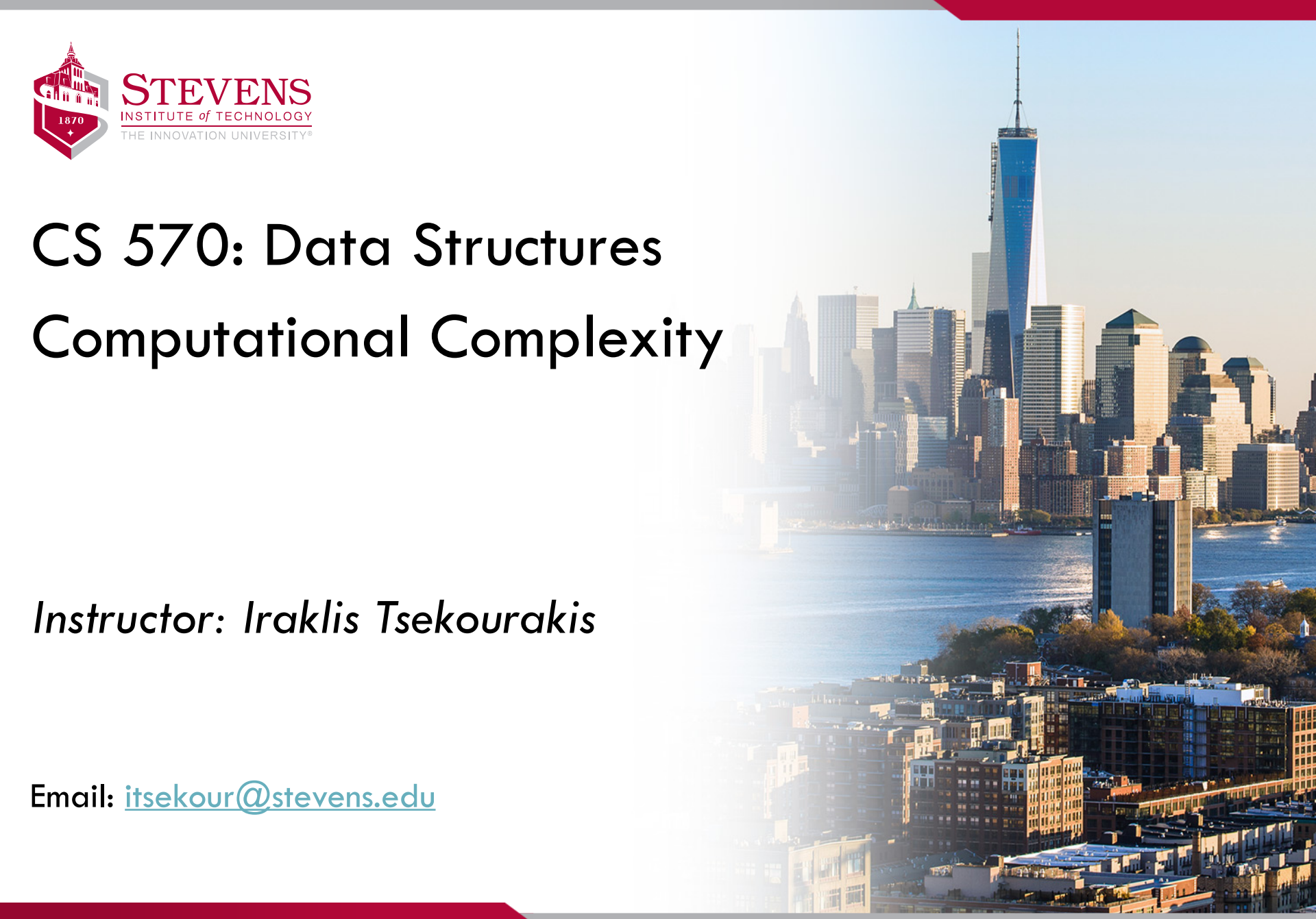


CS 570: Data Structures

Computational Complexity

Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu

Week 3

- Reading Assignment: Koffman and Wolfgang, Sections 1.6, 1.7, 2.1 & Chapter 3

A Java Inheritance Example—The `Exception` Class Hierarchy

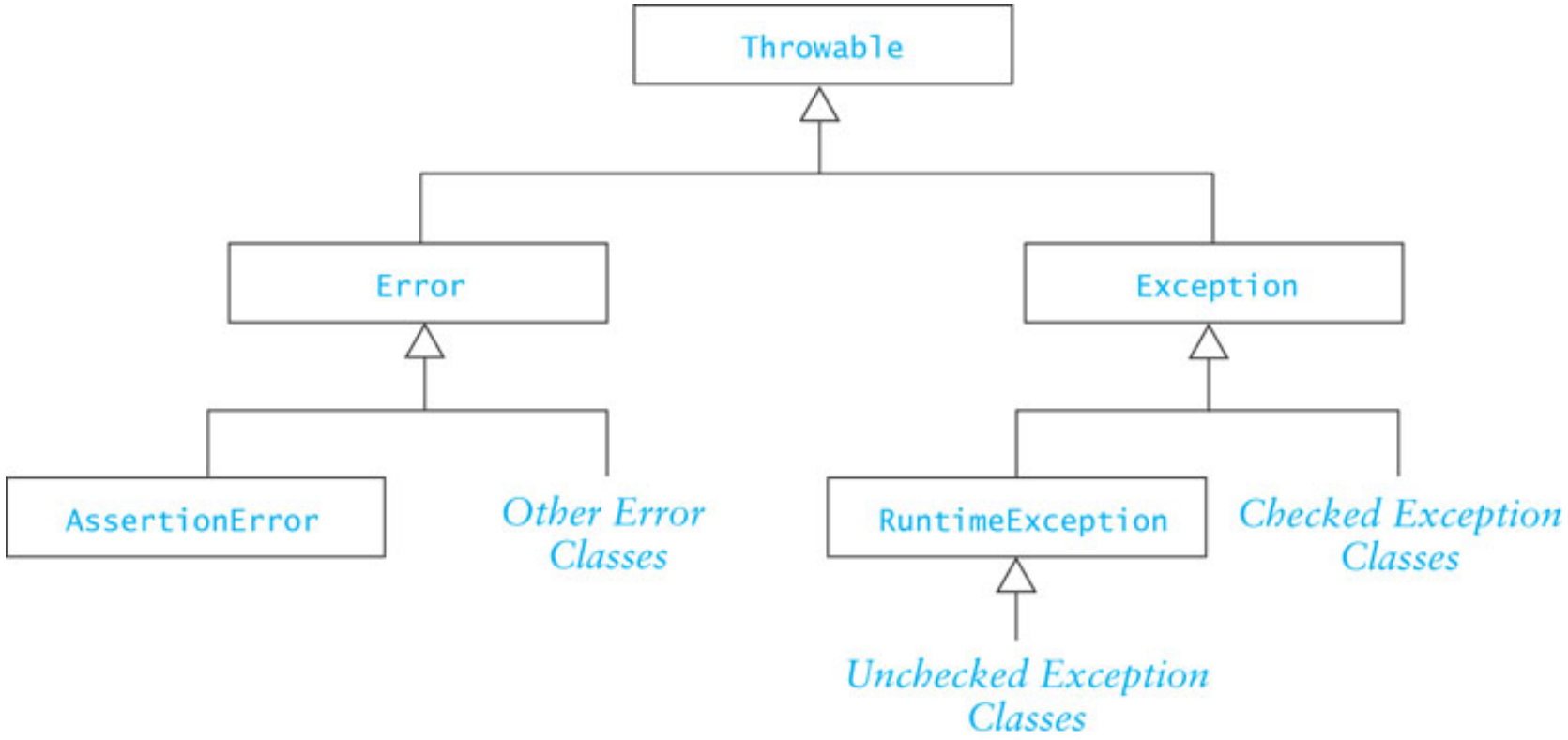
Section 1.6

Run-time Errors or Exceptions

- Run-time errors
  - occur during program execution (i.e. at run-time)
  - occur when the JVM detects an operation that it knows to be incorrect
  - cause the JVM to throw an exception
- Examples of run-time errors include
  - division by zero
  - array index out of bounds
  - number format error
  - null pointer exception

Class `Throwable`

- `Throwable` is the superclass of all exceptions
- All exception classes inherit its methods



Checked and Unchecked Exceptions

- *Checked* exceptions
  - normally not due to programmer error
  - generally beyond the control of the programmer
  - all input/output errors are checked exceptions
  - Examples: `IOException`, `FileNotFoundException`
- *Unchecked* exceptions result from
  - programmer error (try to prevent them with defensive programming)
  - a serious external condition that is unrecoverable
  - Examples: `NullPointerException`, `ArrayIndexOutOfBoundsException`

Checked and Unchecked Exceptions (cont.)

- The class `Error` and its subclasses represent errors due to serious external conditions; they are unchecked
  - Example: `OutOfMemoryError`
  - You cannot foresee or guard against them
  - While you can attempt to handle them, it is generally not a good idea as you will probably be unsuccessful

- The class `Exception` and its subclasses can be handled by a program
  - `RuntimeException` and its subclasses are unchecked
  - *All others* must be either:
    - explicitly *caught* or
    - explicitly *mentioned as thrown* by the method

Some Common Unchecked Exceptions

- `ArithmeticException`: division by zero, etc.
- `ArrayIndexOutOfBoundsException`
- `NumberFormatException`: converting a "bad" string to a number
- `NullPointerException`

Handling Exceptions

- When an exception is thrown, the normal sequence of execution is interrupted
- Default behavior (no handler)
  - Program stops
  - JVM displays an error message
- The programmer may provide a *handle*
  - Enclose statements in a `try` block
  - Process the exception in a `catch` block

The `try-catch` Sequence

- The **`try-catch`** sequence resembles an **`if-then-else`** statement

```
try {
  // Execute the following statements until an
  // exception is thrown
  ...
  // Skip the catch blocks if no exceptions were thrown

} catch (ExceptionTypeA ex) {
  // Execute this catch block if an exception of type
  // ExceptionTypeA was thrown in the try block
  ...

} catch (ExceptionTypeB ex) {
  // Execute this catch block if an exception of type
  // ExceptionTypeB was thrown in the try block
  ...

}
```

The `try-catch` Sequence

- The **`try-catch`** sequence resembles an **`if`**

```
try {
  // Execute the following statements unt
  // exception is thrown
  ...
  // Skip the catch blocks if no exceptio

} catch (ExceptionTypeA ex) {
  // Execute this catch block if an excep
  // ExceptionTypeA was thrown in the try
  ...

} catch (ExceptionTypeB ex) {
  // Execute this catch block if an excep
  // ExceptionTypeB was thrown in the try block
  ...

}
```

PITFALL!
Unreachable `catch` block

`ExceptionTypeB` cannot be a subclass of **`ExceptionTypeA`**. If is was, its exceptions would be caught be the first **`catch`** clause and its **`catch`** clause would be unreachable.

Using `try-catch`

- User input is a common source of exceptions

```
public static int getIntValue(Scanner scan) {
  int nextInt = 0;          // next int value
  boolean validInt = false; // flag for valid input
  while(!validInt) {
    try {
      System.out.println("Enter number of kids: ");
      nextInt = scan.nextInt();
      validInt = true;
    } catch (InputMismatchException ex) {
      scan.nextLine();   // clear buffer
      System.out.println("Bad data-enter an integer");
    }
  }
  return nextInt;
}
```

Throwing an Exception When Recovery is Not Obvious

- In some cases, you may be able to write code that detects certain types of errors, but there may not be an obvious way to recover from them
- In these cases an the exception can be *thrown*
- The calling method receives the thrown exception and must handle it

Throwing an Exception When Recovery is Not Obvious (cont.)


```
public static void processPositiveInteger(int n) {
  if (n < 0) {
    throw new IllegalArgumentException("Invalid negative
                argument");
  } else {
    // Process n as required
    ...
  }
}
```

Throwing an Exception When Recovery is Not Obvious (cont.)


```
public static void main(String[] args) {
  Scanner scan = new Scanner(System.in);
  try {
    int num = getIntValue(scan);
    processPositiveInteger(num);
  } catch (IllegalArguementException ex) {
    System.err.println(ex.getMessage());
    System.exit(1);  // error indication
  }
  System.exit(0);  // normal exit
}
```

Packages and Visibility

Section 1.7

Packages

- A Java *package* is a group of *cooperating classes*
- The Java API is organized as packages
- Indicate the package of a class at the top of the file:

  `package classPackage;`
- Classes in the *same package* should be in the *same directory* (folder)
- The folder must have the same name as the package
- Classes in the *same folder* must be

  in the *same package*

Packages and Visibility

- Classes *not* part of a package can only access `public` members of classes in the package
- If a class is not part of the package, it must access the public classes by their complete name, which would be `packagename.className`
- For example,

```
x = Java.awt.Color.GREEN;
```

- If the package is imported, the `packageName` prefix is not required.

```
import java.awt.Color;
...
x = Color.GREEN;
```

The Default Package

- Files which do not specify a package are part of the default package
- If you do not declare packages, all of your classes belong to the default package
- The default package is intended for use during the early stages of implementation or for small prototypes
- When you develop an application, declare its classes to be in the same package

Visibility

- We have seen three visibility layers, `public`, `protected`, `private`
- A fourth layer, *package visibility*, lies between `private` and `protected`
- Classes, data fields, and methods with package visibility are accessible to all other methods of the same package, but are not accessible to methods outside the package
- Classes, data fields, and methods that are declared protected are visible within subclasses that are declared *outside* the package (in addition to being visible to all members *inside* the package)
- There is no keyword to indicate package visibility
- Package visibility is the default in a package if `public`, `protected`, `private` are not used

Visibility Supports Encapsulation

- Visibility rules enforce encapsulation in Java
- `private`: for members that should be invisible even in subclasses
- package: shields classes and members from classes outside the package
- `protected`: provides visibility to extenders or classes in the package
- `public`: provides visibility to all

Visibility Supports Encapsulation (cont.)

Visibility	Applied to Classes	Applied to Class Members
`private`	Applicable to inner classes. Accessible only to members of the class in which it is declared.	Visible only within this class.
Default or package	Visible to classes in this package.	Visible to classes in this package.
`protected`	Applicable to inner classes. Visible to classes in this package and to classes outside the package that extend the class in which it is declared.	Visible to classes in this package and to classes outside the package that extend this class.
`public`	Visible to all classes.	Visible to all classes. The class defining the member must also be public.

Visibility Supports Encapsulation (cont.)

- Encapsulation insulates against change
- Greater visibility means less encapsulation

- So… use the most restrictive visibility possible to get the job done!

Algorithm Efficiency and Big-O

Section 2.1

Data structure

Data structure

- a way to store and organize data in order to facilitate access and modification.
- no single data structure optimal for all purposes.
- usually optimized for a specific problem setting.
- important to know the strength and limitations of several of them.

Examples:

- Trees (binary search trees, red-black trees, b-trees, ...).
- Stacks (last in, first out), queues (first in, first out), priority queues.

Algorithms

Algorithms:

- well-defined computational procedure.
- takes value or set of values as input.
- produces value or set of values as output.
- tool for solving a well-specified computational problem.
- instance of a problem consists of the input needed to compute a solution to the problem.
- correct algorithm solves the given computational problem.

Sorting problem:

Input: A sequence of n numbers $a_1,\ldots,a_n$.

Output: A permutation $a_1,\ldots,a_n$ of the input sequence such that $a_1 \leq \cdots \leq a_n$.

Efficiency

Computing time and memory are bounded resources.

Efficiency:

- Different algorithms that solve the same problem often differ in their efficiency.
- More significant than differences due to hardware (CPU, memory, disks, ...) and software (OS, programming language, compiler, ...).

Example:

- Insertion sort (c_1n^2) vs. merge sort ($c_2$nlgn).

Asymptotic Performance

- In this course, we care most about *asymptotic performance*
  - How does the algorithm behave as the problem size gets very large?
    - Running time
    - Memory/storage requirements
    - Bandwidth/power requirements/logic gates/etc.

Asymptotic Notation

- By now you should have an intuitive feel for asymptotic (big-O) notation:
  - *What does O(n) running time mean? O(n^2)? O(n lg n)?*
  - *How does asymptotic running time relate to asymptotic memory usage?*
- Our first task is to define this notation more formally and completely

Input Size

- Time and space complexity
  - This is generally a function of the input size
    - E.g., sorting, multiplication
  - How we characterize input size depends:
    - Sorting: number of input items
    - Multiplication: total number of bits
    - Graph algorithms: number of nodes & edges
    - Etc

Running Time

- Number of primitive steps that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
    - y = m * x + b
    - c = 5 / 9 * (t - 32)
    - z = f(x) + g(y)
- We can be more exact if need be
- Worst case vs. average case

Algorithm Efficiency and Big-O

- Getting a precise measure of the performance of an algorithm is difficult
- Big-O notation expresses the performance of an algorithm as a function of the number of items to be processed
- This permits algorithms to be compared for efficiency
- For more than a certain number of data items, some problems cannot be solved by any computer

Linear Growth Rate

- If processing time increases in proportion to the number of inputs *n*, the algorithm grows at a linear rate

```
public static int search(int[] x, int target)
{
  for(int i=0; i<x.length; i++) {
    if (x[i]==target)
      return i;
  }
  return -1; // target not found
}
```

Linear Growth

- If processing time in
  number of inputs *n*, t
  rate

- If the target is not present, the `for` loop will execute `x.length` times
- If the target is present the `for` loop will execute (on average) `(x.length + 1)/2` times
- Therefore, the total execution time is directly proportional to `x.length`
- This is described as a growth rate of order *n* OR O(n)

```
public static int search(int[] x, int target) {
  for(int i=0; i<x.length; i++) {
    if (x[i]==target)
      return i;
  }
  return -1; // target not found
}
```

n x m Growth Rate

- Processing time can be dependent on two different inputs

```
public static boolean areDifferent(int[] x, int[] y) {
  for(int i=0; i<x.length; i++) {
    if (search(y, x[i]) != -1)
      return false;
  }
  return true;
}
```

n x m Growth Rate (cont.)

- Processing time o
  inputs.

- The `for` loop will execute `x.length` times
- But it will call `search`, which will execute `y.length` times
- The total execution time is proportional to (`x.length * y.length`)
- The growth rate has an order of n x m or O(n x m)

```
public static boolean areDifferent(int[] x, int[] y) {
  for(int i=0; i<x.length; i++) {
    if (search(y, x[i]) != -1)
      return false;
  }
  return true;
}
```

Quadratic Growth Rate

- If processing time is proportional to the square of the number of inputs *n*, the algorithm grows at a quadratic rate

```
public static boolean areUnique(int[] x) {
  for(int i=0; i<x.length; i++) {
    for(int j=0; j<x.length; j++) {
      if (i != j && x[i] == x[j])
        return false;
    }
  }
  return true;
}
```

Quadratic Growth Rate (cont.)

- If processing time is
  number of inputs *n*,

- The `for` loop with `i` as index will execute `x.length` times
- The `for` loop with `j` as index will execute `x.length` times
- The total number of times the inner loop will execute is $(x.length)^2$
- The growth rate has an order of n^2 or $O(n^2)$

```
public static boolean
  for(int i=0; i<x.length; i++) {
    for(int j=0; j<x.length; j++) {
      if (i != j && x[i] == x[j])
        return false;
    }
  }
  return true;
}
```

Big-O Notation

- The O() in the previous examples can be thought of as an abbreviation of "order of magnitude"
- A simple way to determine the big-O notation of an algorithm is to look at the loops and to see whether the loops are nested
- Assuming a loop body consists only of simple statements,
  - a single loop is O(n)
  - a pair of nested loops is $O(n^2)$
  - a nested pair of loops inside another is $O(n^3)$
  - and so on . . .

Big-O Notation (cont.)

- You must also examine the *number of times* a loop is executed

```
for(i=1; i < x.length; i *= 2) {
   // Do something with x[i]
}
```

- The loop body will execute k-1 times, with `i` having the following values:

  $1, 2, 4, 8, 16, \ldots, 2^k$

  until 2^k is greater than `x.length`
- Since $2^{k-1} =$ `x.length` $< 2^k$ and $\log_2 2^k$ is k, we know that k-1 $= \log_2($`x.length`$) < k$
- Thus we say the loop is O(log n) (in analyzing algorithms, we use logarithms to the base 2)
- Logarithmic functions grow slowly as the number of data items n increases

Formal Definition of Big-O

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
      Simple Statement
  }
}
for (int i = 0; i < n; i++) {
   Simple Statement  1
   Simple Statement 2
   Simple Statement 3
   Simple Statement 4
   Simple Statement 5
}
Simple Statement 6
Simple Statement 7
. . .
Simple Statement 30
```

Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
      Simple Statement
  }
}
for (int i = 0; i < n; i++) {
   Simple Statement 1
   Simple Statement 2
   Simple Statement 3
   Simple Statement 4
   Simple Statement 5
}
Simple Statement 6
Simple Statement 7
. . .
Simple Statement 30
```

This nested loop executes a *Simple Statement* n^2 times

Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
      Simple Statement
  }
}
for (int i = 0; i < n; i++) {
   Simple Statement 1
   Simple Statement 2
   Simple Statement 3
   Simple Statement 4
   Simple Statement 5
}
Simple Statement 6
Simple Statement 7
. . .
Simple Statement 30
```

This loop executes 5 *Simple Statements* n times ($5n$)

Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
      Simple Statement
  }
}
for (int i = 0; i < n; i++) {
   Simple Statement 1
   Simple Statement 2
   Simple Statement 3
   Simple Statement 4
   Simple Statement 5
}
Simple Statement 6
Simple Statement 7
. . .
Simple Statement 30
```

Finally, 25 *Simple Statements* are executed

Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
      Simple Statement
  }
}
for (int i = 0; i < n; i++) {
   Simple Statement 1
   Simple Statement 2
   Simple Statement 3
   Simple Statement 4
   Simple Statement 5
}
Simple Statement 6
Simple Statement 7
. . .
Simple Statement 30
```

We can conclude that the relationship between processing time and *n* (the number of date items processed) is:

$$T(n) = n^2 + 5n + 25$$

Formal Definition of Big-O (cont.)

- In terms of T(n),

$$T(n) = O(f(n))$$

- means that there exist
  - two constants, n_0 and c, greater than zero, and
  - a function, f(n),
- such that for all $n > n_0$, $\quad cf(n) >= T(n)$
- In other words, as n gets sufficiently large (larger than n_0), there is some constant c for which the processing time will always be less than or equal to cf(n)
- cf(n) is an upper bound on performance

Formal Definition of Big-O (cont.)

- The growth rate of f(n) will be determined by the fastest growing term, which is the one with the largest exponent
- In the example, an algorithm of

$$O(n^2 + 5n + 25)$$

is more simply expressed as

$$O(n^2)$$

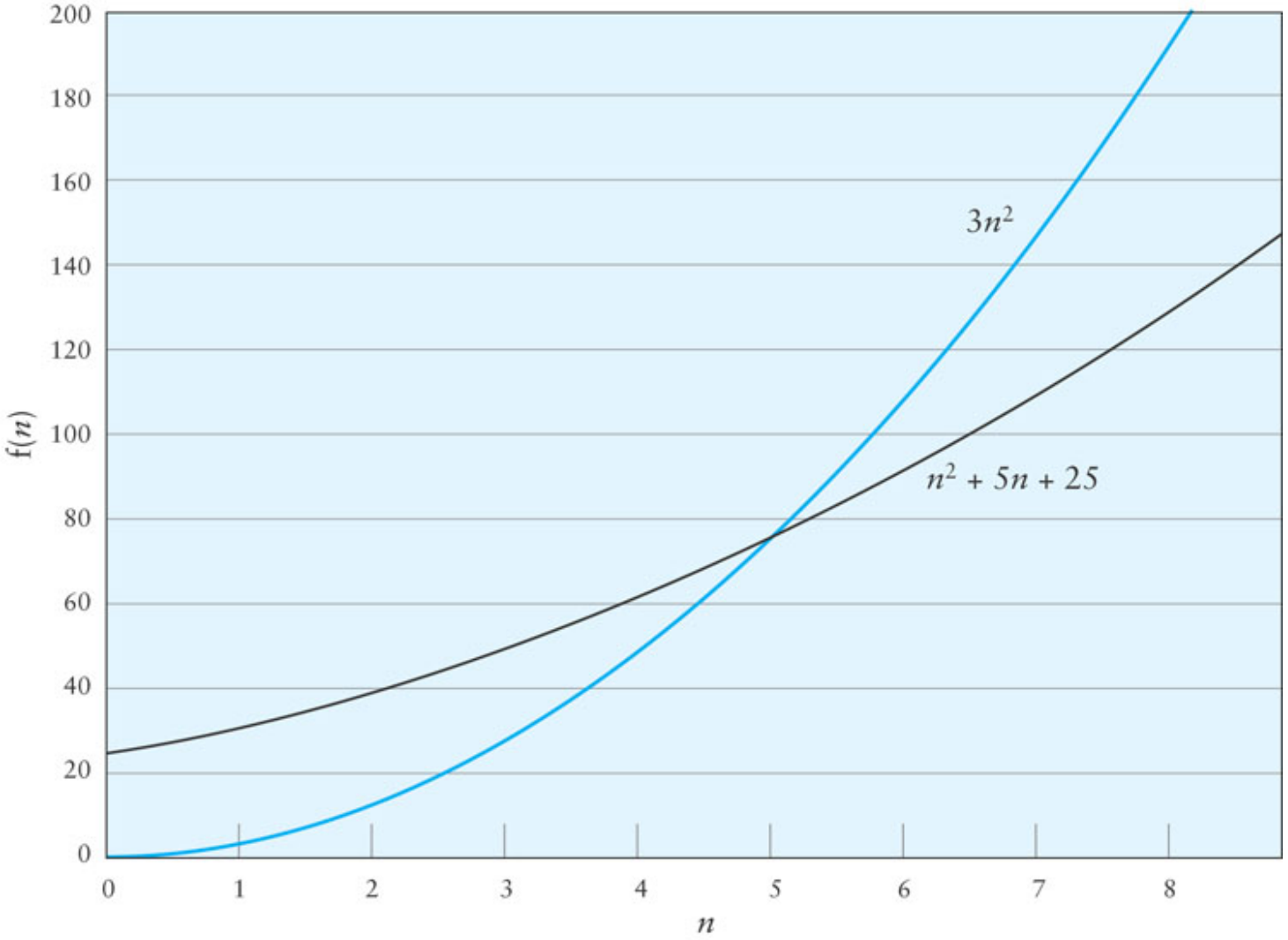
- In general, it is safe to ignore all constants and to drop the lower-order terms when determining the order of magnitude

Big-O Example 1

- Given T(n) = $n^2 + 5n + 25$, show that this is O(n^2)
- Find constants n_0 and c so that, for all $n > n_0$, $cn^2 > n^2 + 5n + 25$
  - Find the point where $cn^2 = n^2 + 5n + 25$
  - Let $n = n_0$, and solve for c

    $c = 1 + 5/ n_0 + 25/ n_0^2$
- When n_0 is 5, the RHS is (1 + 5/5 + 25/25), c is 3
- So, $3n^2 > n^2 + 5n + 25$ for all $n > 5$
- Other values of n_0 and c also work

Big-O Example 1 (cont.)



Big-O Example 2

- Consider the following loop

```
for (int i = 0; i < n; i++) {
  for (int j = i + 1; j < n; j++) {
      3 simple statements
  }
}
```

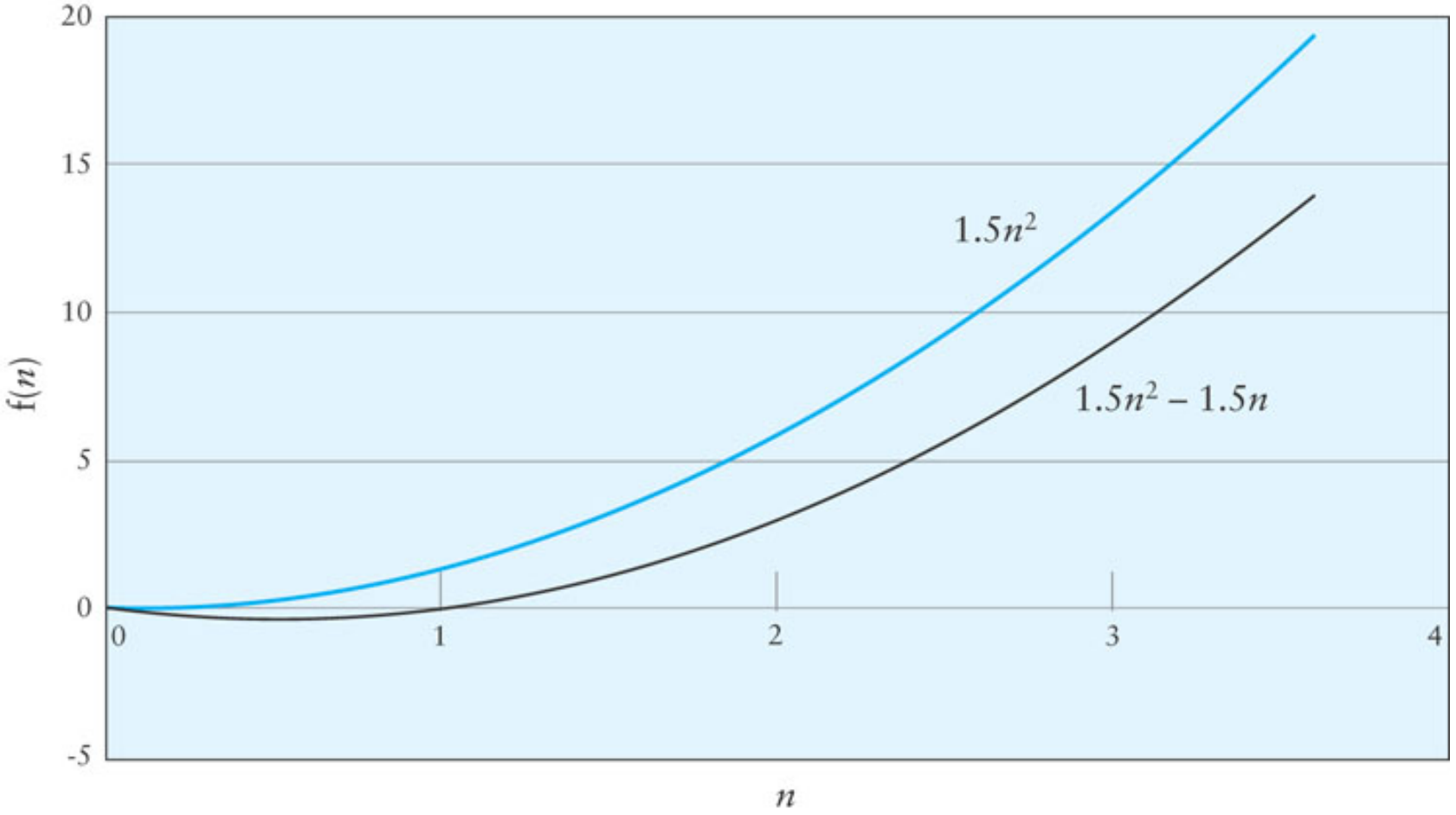
- $T(n) = 3(n - 1) + 3(n - 2) + \ldots + 3$
- Factoring out the 3,

  $3(n - 1 + n - 2 + \ldots + 1)$
- $1 + 2 + \ldots + n - 1 = (n \times (n-1))/2$

Big-O Example 2 (cont.)

- Therefore T(n) = $1.5n^2 - 1.5n$
- When $n = 1$, the polynomial has value 0
- For values of $n > 1$, $1.5n^2 > 1.5n^2 - 1.5n$
- Therefore T(n) is O(n^2) when n_0 is 1 and c is 1.5

Big-O Example 2 (cont.)



Big-O of polynomials

- Suppose runtime is $an^2 + bn + c$
  - If any of a, b, and c are less than 0 replace the constant with its absolute value
- $an^2 + bn + c \leq (a + b + c)n^2 + (a + b + c)n + (a + b + c)$
- $\leq 3(a + b + c)n^2$ for $n \geq 1$
- Let $c' = 3(a + b + c)$ and let $n_0 = 1$

Big-O of polynomials

- **A polynomial of degree k is $O(n^k)$**
- Proof:
  - Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$
    - Let $a_i = | b_i |$
  - $f(n) \leq a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

$$\leq \quad n^k \sum a_i \frac{n^i}{n^k} \quad \leq \quad n^k \sum a_i \quad \leq \quad cn^k$$

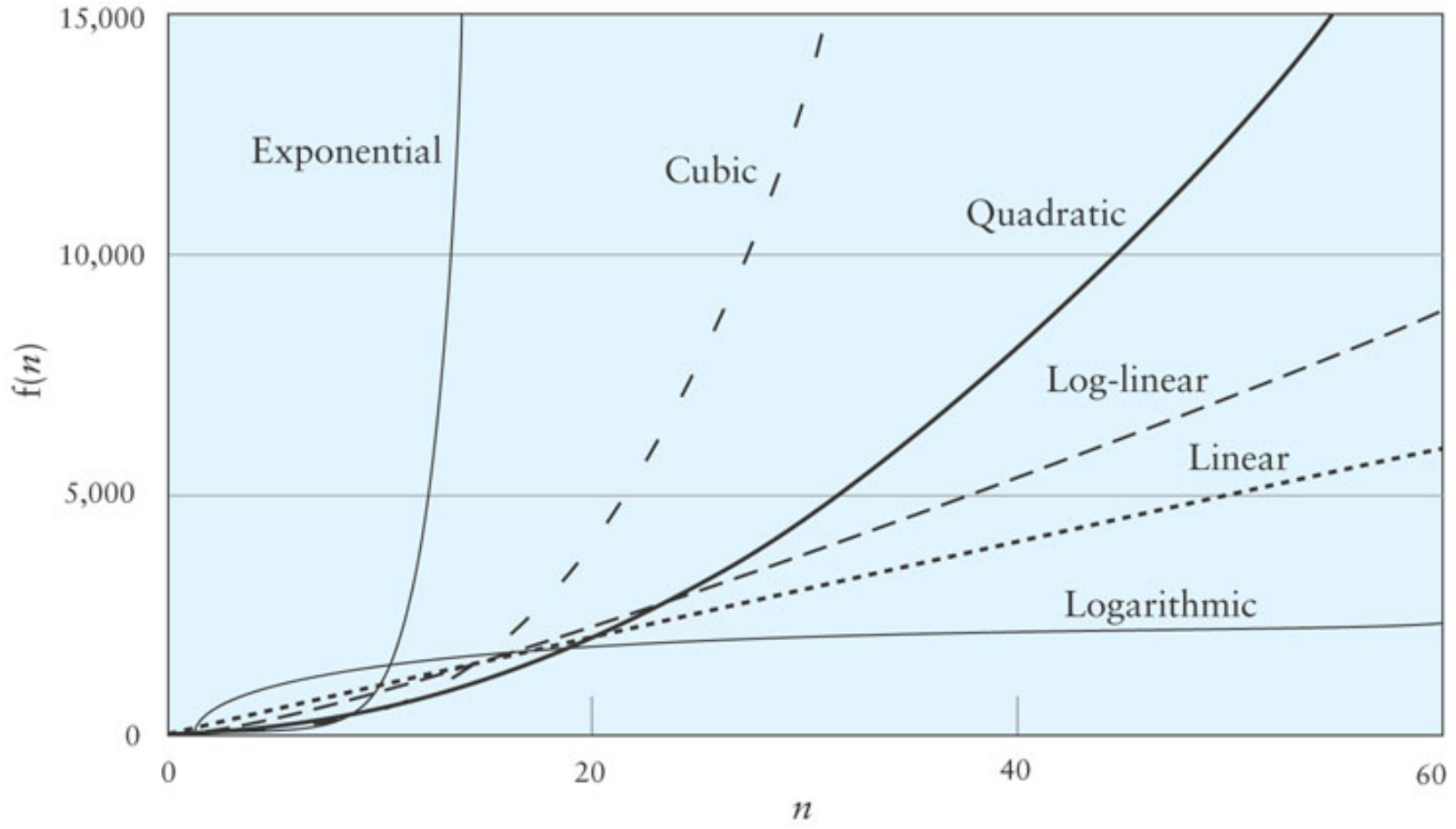
Symbols Used in Quantifying Performance

Symbol	Meaning
$T(n)$	The time that a method or program takes as a function of the number of inputs, n. We may not be able to measure or determine this exactly.
$f(n)$	Any function of n. Generally, $f(n)$ will represent a simpler function than $T(n)$, for example, n^2 rather than $1.5n^2 - 1.5n$.
$O(f(n))$	Order of magnitude. $O(f(n))$ is the set of functions that grow no faster than $f(n)$. We say that $T(n) = O(f(n))$ to indicate that the growth of $T(n)$ is bounded by the growth of $f(n)$.

Common Growth Rates

Big-O	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

Different Growth Rates



Effects of Different Growth Rates

O(f(n))	f(50)	f(100)	f(100)/f(50)
O(1)	1	1	1
O(log n)	5.64	6.64	1.18
O(n)	50	100	2
O(n log n)	282	664	2.35
O(n^2)	2500	10,000	4
O(n^3)	12,500	100,000	8
O(2^n)	1.126×10^{15}	1.27×10^{30}	1.126×10^{15}
O($n!$)	3.0×10^{64}	9.3×10^{157}	3.1×10^{93}

Algorithms with Exponential and Factorial Growth Rates

- Algorithms with exponential and factorial growth rates have an effective practical limit on the size of the problem they can be used to solve
- With an $O(2^n)$ algorithm, if 100 inputs takes an hour then,
  - 101 inputs will take 2 hours
  - 105 inputs will take 32 hours
  - 114 inputs will take 16,384 hours (almost 2 years!)

Algorithms with Exponential and Factorial Growth Rates (cont.)

- Encryption algorithms take advantage of this characteristic
- Some cryptographic algorithms can be broken in $O(2^n)$ time, where n is the number of bits in the key
- A key length of 40 is considered breakable by a modern computer,
- but a key length of 100 bits will take a billion-billion (10^{18}) times longer than a key length of 40

Testing

Section 2.11

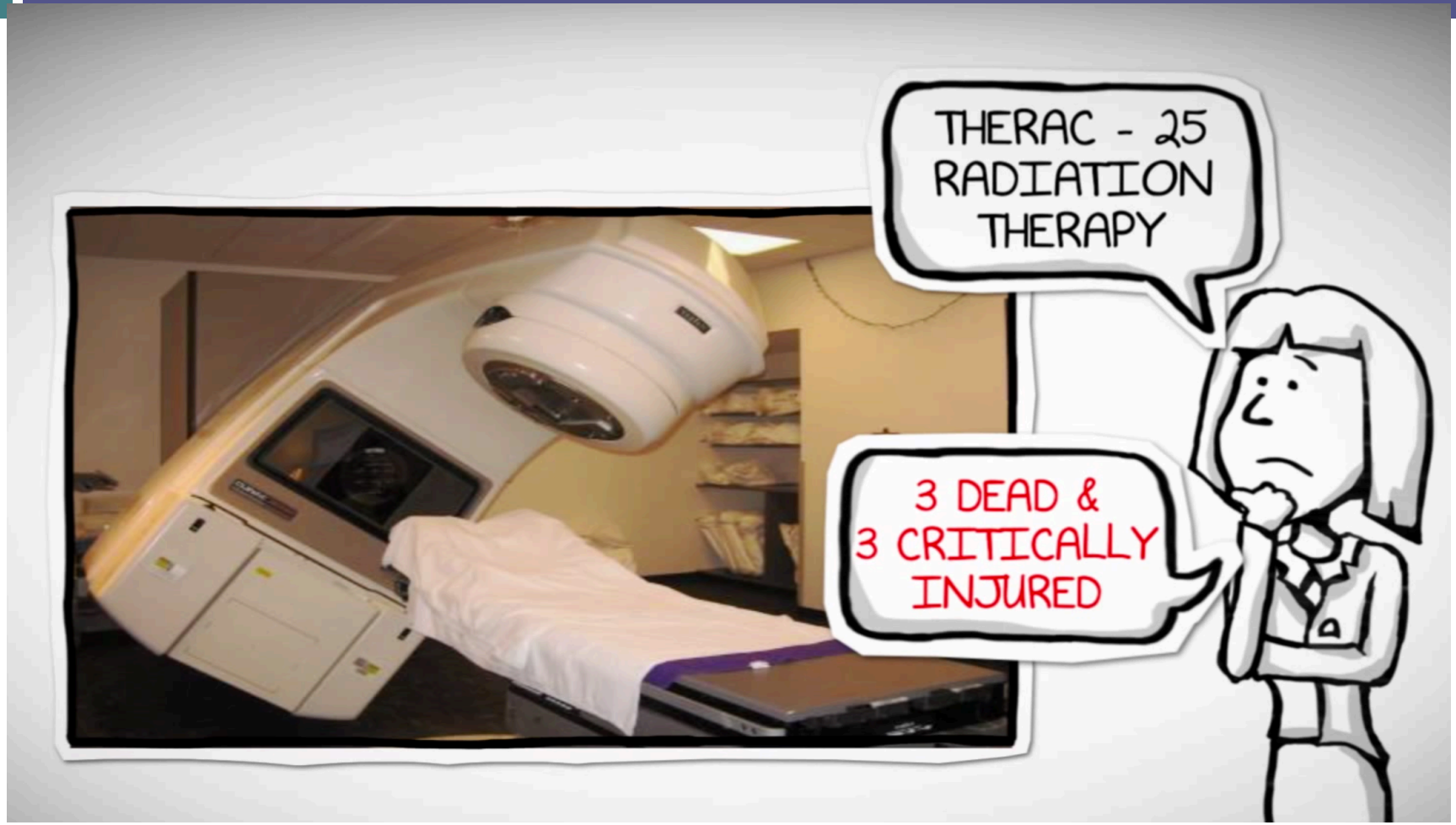
Testing

- Testing runs a program or part of a program under controlled conditions to verify that results are as expected
- Testing detects program defects after the program compiles (all syntax error have been removed)
- While extremely useful, testing cannot detect the absence of all defects in complex programs

Why is Testing Important?

China Airlines Flight 140, April 26th, 1994

Why is Testing Important?



Why is Testing Important?

April, 1999

Why is Testing Important?

May, 1996

Paul Ehrlich:
"*To err is human, but to really foul things up you need a computer*."

Testing Levels

- **Unit testing:** tests the smallest testable piece of the software, often a class or a sufficiently complex method
- **Integration testing:** tests integration among units
- **System testing:** tests the whole program in the context in which it will be used
- **Acceptance testing:** system testing designed to show that a program meets its functional requirements

Types of Testing

- Black-box testing:
  - tests the item (method, class, or program) based on its interfaces and functional requirements
  - is also called *closed-box* or *functional* testing
  - is accomplished by varying input parameters across the allowed range and outside the allowed range, and comparing with independently calculated results

Types of Testing (cont.)

- White-box testing:
  - tests the item (method, class, or program) with knowledge of its internal structure
  - is also called *glass-box*, *open-box*, or *coverage* testing
  - exercises as many paths through the element as possible
  - provides appropriate coverage
    - statement – ensures each statement is executed at least once
    - branch – ensures each choice of branch (if , switch, loops) is taken
    - path – tests each path through a method

Preparations for Testing

- A test plan should be developed early in the design stage—the earlier an error is detected, the easier and less expensive it is to correct it
- Aspects of test plans include deciding:
  - how the software will be tested
  - when the tests will occur
  - who will do the testing
  - what test data will be used

Testing Tips

1. Carefully document method operation, parameter, and class attributes using comments; follow Javadoc conventions
2. Leave a trace of execution by displaying the method name as you enter it
3. Display values of all input parameters upon entering a method and values of any class attributes accessed by the method
4. Display values of all method outputs after returning from a method, together with any class attributes that are modified by a method

Testing Tips (cont.)

- An efficient way to display values of parameters, return values, and class attributes:

```
private static final boolean TESTING = true; // or false to
                                             // disable

if (TESTING) {
  // Code for output statements
}
```

- Remove these features when you are satisfied with the testing results
- You can define different `boolean` flags for different tests

Developing the Test Data

- In black-box testing, test data should check for all expected inputs as well as unanticipated data
- In white-box testing, test data should be designed to ensure all combinations of paths through the code are executed

Testing Boundary Conditions

- Example

```
for (int i = 0; i < x.length; i++) {
  if (x[i] == target)
    return i;
}
```

- Test the boundary conditions (for white-box and black-box testing) when `target` is:
  - first element (`x[0] == target` is `true`)
  - last element (`x[length-1] == target` is `true`)
  - not in array (`x[i] == target` is always `false`)
  - present multiple times (`x[i] == target` for more than one value of `i`)

Testing Boundary Conditions (cont.)


```
for (int i = 0; i < x.length; i++) {
  if (x[i] == target)
    return i;
}
```

- Test for the typical situation when `target` is:
  - somewhere in the middle
- and for the boundary conditions when the array has
  - only one element
  - no elements

Testing Boundary Conditions (cont.)


```
private static void verify(int[] x, int target, int
   expected) {
   int actual = search(x, target);
   System.out.print("search(x, " + target + ") is "
             + actual + ", expected " + expected);
   if (actual == expected)
       System.out.println(": Pass");
   else
       System.out.println(": ****Fail");
}
```

Testing Boundary Conditions (cont.)


```
public static void main(String[] args) {
  // Array to search.
  int[] x = {5, 12, 15, 4, 8, 12, 7};
  // Test for target as first element.
  verify(x, 5, 0);
  // Test for target as last element.
  verify(x, 7, 6);
  // Test for target not in array.
  verify(x, -5, -1);
  // Test for multiple occurrences of target.
  verify(x, 12, 1);
  …
```

Testing Boundary Conditions (cont.)


```
public static void main(String[] args) {
   int[] x = {5, 12, 15, 4, 8, 12, 7}; // Array to search.
   …
   // Test for target somewhere in middle.
   verify(x, 4, 3);
   // Test for 1-element array.
   x = new int[1];
   x[0] = 10;
   verify(x, 10, 0);
   verify(x, -10, -1);
   // Test for an empty array.
   x = new int[0];
   verify(x, 10, -1);
}
```

Stubs

- *Stubs* are method placeholders for methods called by other classes, but not yet implemented
- Stubs allowing testing as classes are being developed
- A sample stub:

```
public void save() {
   System.out.println("Stub for save has
         been called");
   modified = false;
}
```

Stubs (cont.)

- Stubs can
  - print out value of inputs
  - assign predictable values to outputs
  - change the state of variables

Preconditions and Postconditions

- A *precondition* is a statement of any assumptions or constraints on the input parameters before a method begins execution
- A *postcondition* describes the result of executing the method, including any change to the object's state
- A method's preconditions and postconditions serve as a contract between a method caller and the method programmer

Preconditions and Postconditions


```
/** Method Save
    pre: the initial directory contents are read
      from a data file
    post: writes the directory contents back to
      a data file
*/
public void save() {
. . .
}
```

Drivers

- Another testing tool
- A driver program
  - declares any necessary object instances and variables
  - assigns values to any of the method's inputs (specified by the preconditions)
  - calls the method
  - displays the outputs returned by the method
- Driver code can be added to a class's `main` method (each class can have a `main` method; only one `main` method - the one you designate to execute - will run)

JUnit

- JUnit, a popular program for Java projects, helps you develop testing programs (see Appendix D)
- Many IDEs are shipped with debugger programs you can use for testing
- Use Javadoc!