CS2110: SW Development Methods

Midterm 2
~REVIEW~

Exam on Friday, April 3, 2020
Spring 2020

Unlike Exam 1 … Exam 2 is:

- OPEN-book
- OPEN-notes
- OPEN-IDE (i.e. Eclipse or similar)
- OPEN-browser

- However…
  - You are **NOT** permitted to collaborate in any way
  - Completion of the test must represent your **individual** work
- If we find collaboration of ***any*** kind it will result in an automatic F / NC in the course (*unappealable*)

Exam Date, Duration, Location

- Given the test is given virtually, all students can start the exam ***any time*** on
  Friday, April 3, 2020 `(00:00:01am to 11:59:59pm)`

- You **must** COMPLETE the test **within one (1) hour**
  - The time you open the exam to the time you submit both parts of the exam should NOT exceed one (1) hour

- This test is to be take at your own location

Exam Format

- **100% electronic** (virtual) – no in-class component at all
- All of the exam will be on **Collab** (under Quizzes) and the **Ford** code testing site (which you'll be linked to)

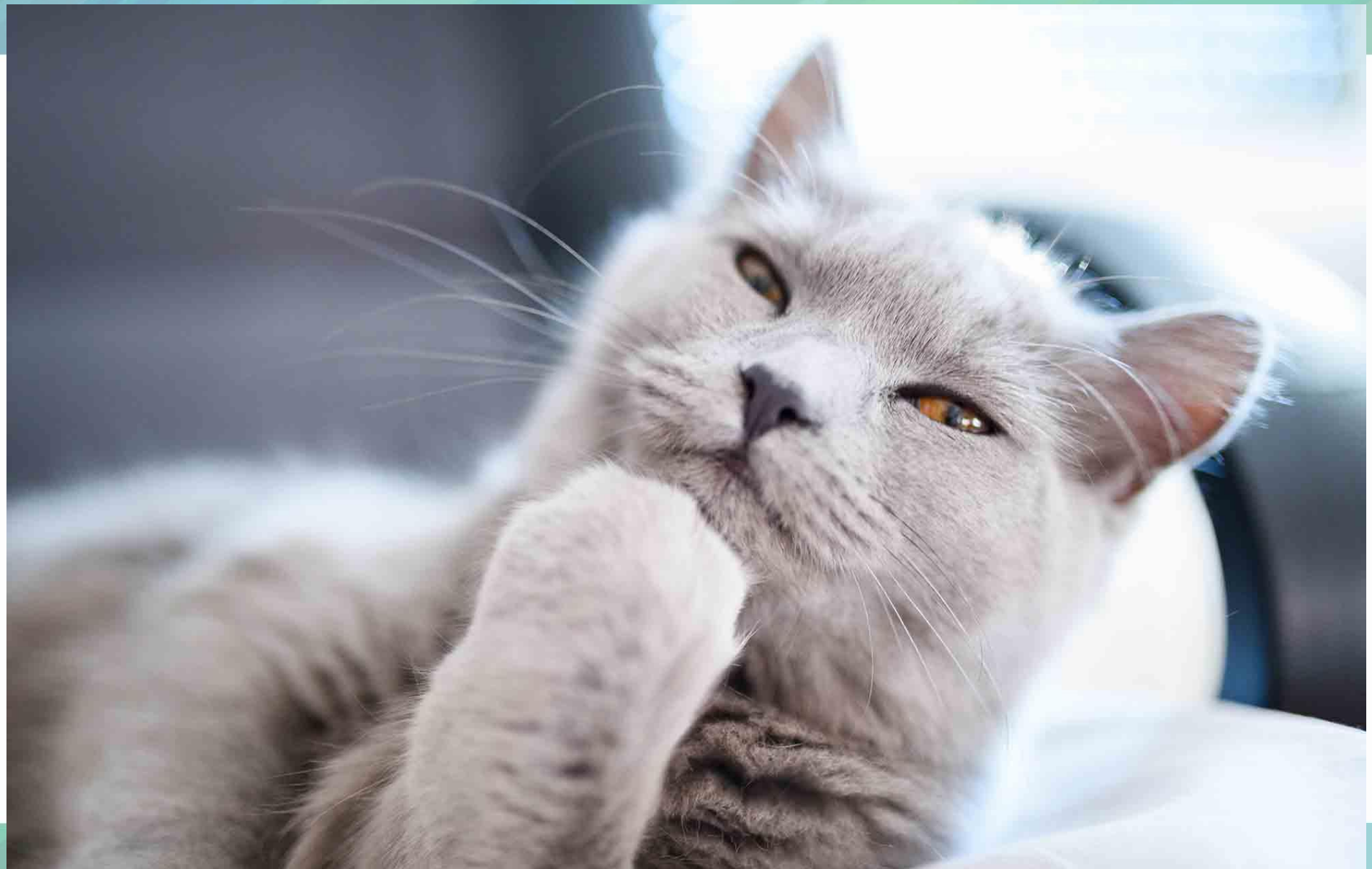
- The format of the exam will consist of multiple-choice, fill-in-the-blank, short answer, and coding questions

- **You will need:**
  - A **laptop** (sufficiently charged to run for one hour) or a **desktop** computer
  - A stable and **reliable internet connection**

SDAC

- **SDAC** students please make sure your SDAC arrangements have been *finalized* with SDAC so we can provide you with appropriate accommodations

SO… ANY TIPS…?



Collab Tip…

- When taking the exam, **use only one tab on Collab!**

- Opening multiple tabs will cause issues with the active quiz (exam) and you will most probably lose your work and/or mess up the timer.

- We will not make special accommodations, if we realize you had multiple tabs trying to access other pages on Collab (e.g., prior Quizzes).

Important Stipulations (I)

- **Electronic timestamps** are used and recorded starting at the time you open the exam and the time you submit.
  - After completing and submitting your code for the **coding questions on the Ford website**, you will **receive a code**. You must place this code into the text field of the last question on the Collab quiz portion of the exam.
  - If no code is entered here, you will get a zero in the coding portion of the exam (no exceptions will be made under ANY circumstance.)
    So, plan accordingly to submit on Ford a few minutes before the time expires on Collab.

Important Stipulations (II)

- **Electronic timestamps** are used and recorded starting at the time you open the exam and the time you submit.
  - If the **time spent on Ford** is **more than 1 hour** after your start time, it will **not** be accepted and you will get a zero on the coding portion of the exam (no exceptions will be made under ANY circumstance.)
    So, plan accordingly to submit on Ford a few minutes before the time expires on Collab.

Test Taking Tips / Strategies

- Complete the higher-point value questions first
- Complete the questions you feel confident about first
- Toggle back and forth between questions you feel confident about, and questions you feel less confident about
- You can go back and forth between the Collab quiz questions and the Ford coding questions

- Keep a watch/clock next to you, try not to spend too much time on any one question
- Budget time to answer coding questions
- Budget time to copy the code received after submitting the coding questions into the text field of the last Collab question


Can we talk about what's on the exam?

Possible Coding Topics

- (Note this may not be a complete list)

- Write compareTo() or compare() method for a class
- Write (parts of) a class that implements Comparable or Comparator
- Write constructors for classes that are in an inheritance relationship (remember to use “super”)
- Write try-catch code segments (exceptions)
- Override a method (with new implementation)

Implementing Comparable ~ fulfilling the contract

- Implement .compareTo(T o) to fulfill the contract

```
public int compareTo(T o) { … }
```

  - Format: `string1.compareTo(string2) //returns an int`

  - Programming convention: **Return value as follows**:

    - **zero** if the same ~ sameness should be same as .equals()
    - **negative value** if first item strictly **less** than second
    - **positive value** if first item strictly **greater** than second

  - We don't care about the actual value

compareTo()String Examples

- Consider the following code:

```
System.out.println("cat".compareTo("dog"));
System.out.println("eagle".compareTo("cat"));
System.out.println("cats".compareTo("catcher"));
System.out.println("bed".compareTo("bedroom"));
```

Prints	What it means
-1	cat < dog
2	eagle > cat
16	cats > catcher
-4	bed < bedroom

- When the first world comes alphabetically before the second word, the result is negative
- When the first world comes alphabetically after the second word, the result is positive
- If we use two identical strings, the result will be **zero**

More about compareTo()

- Digits are less than letters

```
System.out.println("999".compareTo("AAA")); //negative
```

- Capital letters are less than lowercase letters

```
System.out.println("AAA".compareTo("aaa")); //negative
```

Example: Writing compareTo()

- Imagine something like an entry in a **phonebook**
  - Order by last name, first name, then number

```
int compareTo(PhoneBookEntry item2 ) {
      int retVal= this.last.compareTo(item2.last);
      if ( retVal != 0 ) return retVal;
      retVal = this.first.compareTo(item2.first);
      if ( retVal != 0 ) return retVal;
      retVal = this.phNum - item2.phNum;
      return retVal;
}
```

The type is the type of the class! (Not "Object" like the equals() method!)

compareTo() for Strings!

Use of subtraction when dealing with numbers (a primitive) – will still be pos/neg/zero

PhoneBookEntry
last: String first: String phNum: int
compareTo(PhoneBookEntry, item) : int

compareTo() and various types

- Strings:
  - **compareTo()** with Strings uses alphabetical order to give you an "order" of Strings
  - Format: stringA.compareTo(stringB); // returns an int
- Numbers (ints) – e.g. sort students by score
  - Use **subtraction method** (*not* compareTo())
    - If "this.score" is 80 and "o.score" is 90
    - this.score – o.score is: 80-90 = -10 (negative)
  - This will sort student scores in ascending order (Question: how to sort in descending order??)
- Object /reference types: use **compareTo()** !

```
@Override
public int compareTo(Student o) {
    return this.score - o.score;
}
```

A Comparator Example: Student class

- Consider the Student class
  - What if we want to sort in a more specific way:
    - Sort by scores in descending order
    - If two students have the same score, sort by name
- We can create a Comparator as follows:

```
1
2  import java.util.Comparator;
3
4  public class StudentScoreNameComparator implements Comparator<Student> {
5      public int compare(Student s0, Student s1) {
6          if (s0.score == s1.score) {
7              return s0.name.compareTo(s1.name);
8          } else {
9              return s1.score - s0.score;
10         }
11     }
12 }
```

A Comparator Example: Dog class

```
public class CmpDogByBreedAndName implements Comparator<Dog> {
    // WAY 1: first by breed, then by name
    public int compare(Dog d1, Dog d2) {
      int retVal = d1.getBreed().compareTo(d2.getBreed());
      if (retVal != 0) return retVal;
      return d1.getName().compareTo(d2.getName());
    }
}

public class CmpDogByNameAndBreed implements Comparator<Dog> {
    // WAY 2:  first by name, then by breed
    public int compare(Dog d1, Dog d2) {
      int retVal = d1.getName().compareTo(d2.getName());
      if (retVal != 0) return retVal;
      return d1.getBreed().compareTo(d2.getBreed());
    }
}
```

Question Topics

- (Note this may not be a complete list)

- **Software Development Life Cycle**
  - Phases of the software development life cycle
    - Requirements
    - Design
    - Coding/implementation
    - Integration
    - Maintenance (~67% overall cost)
  - Programming in the small vs. programming in the large
  - Requirement statements: functional/non-functional/constraint/not a requirement

Requirements

Functional	Non-functional	Constraints
Describes a ***function*** or activity of the system; ***WHAT*** the system will do	Adds ***qualities*** to a given functional requirements; ***HOW*** something gets done (not design "how")	Come mostly directly from the client
Perhaps in response to (user or other) input(s)	E.g., efficiency, accuracy, reliability, usability, etc.	On system ***design*** e.g., running on a specific platform
Describes the state of the system (or parts) before (*preconditions*) and after (*postconditions*) the activity occurs		On how it is ***built*** e.g., following specific standards or programming environments

Object-Oriented (OO) Design & Programming

- Interfaces:
  - Comparable (compareTo() method)
  - Comparator (compare() method)
- Objects have state, behavior, and identity
- Terminology in inheritance
- Motivation for inheritance
- Inheritance hierarchies

Classes, subclasses, abstract classes, and interfaces

Type	Description
Class	Make a **CLASS** that does not extend anything (besides Object) when your new class ***does not pass the IS-A test*** for any other type
Subclass	Make a **SUBCLASS** (i.e., extend another class) ONLY when you need to make a ***more specific version*** of a class and need to override or add new behaviors
Abstract class	Use an **ABSTRACT CLASS** when you want to define a ***template (contract)*** for a group of subclasses, and you have at least some implementation code that all subclasses should use (not all abstract methods); also when you want to ensure that ***no objects of that type can be instantiated***
Interface	Use an **INTERFACE** when you want to define a ***role*** that other classes can play (i.e., subclass implements interface), regardless of where these classes are in the inheritance tree

Object-Oriented (OO) Design & Programming

- Run-time polymorphism
  - Benefits
  - How it works (read/right code that uses it)
  - Determine if statements are valid or invalid based on provided code, e.g. Animal a = new Cat(…); is a.method1() valid or not?
- Constructors of sub-classes (remember to use key word Super, etc…)

Run-time Polymorphism

- What is **run-time polymorphism**?
  - It is the different effects of invoking the **same method** on **different types** of objects
    - Java asks "**who are you?**" ("what is your data type?")
    - At **run-time**, Java calls the **appropriate** method
    - Could be many methods of the same name in different classes
  - Java provides this through *inheritance* and *interfaces*
    - (Also, related to ***substitutability principle***)

Another Run-time Polymorphism Example: UVAperson

```
public class UVAperson {     // UVAperson class
      public UVAperson() { }; // constructor

      public department() { S.O.P("Work for a dept"); }
      public getAddress() { S.O.P("Mailing Address (UVA)"); }
      public getJobOffer() { S.O.P("Offered a job!"); }
}
      Note: "S.O.P" means "System.out.println"
```

```
public class Student extends UVAperson {   // UVAperson is superclass
      public Student() { }; // constructor

      public department() { S.O.P("Student-This is my major!"); }
      public enroll() { S.O.P("Student-Took a class!"); }
}
```

```
public class UgradSt extends Student { // Student is superclass
      public UgradSt() { }; // constructor

      public getJobOffer() { S.O.P("Ugrad-Got a Job!"); }
}
```

In main somewhere (e.g. in UVA_person):

```
      UVAperson s1  = new Student();
      Student   uG1 = new UgradSt();
      UVAperson uG2 = new UgradSt();
```

QUESTIONS:

```
s1.department(); // Student(specific), so "Student-This is my major!"
uG2.enroll();    // ILLEGAL - UVAperson doesn't have enroll()!
s1.enroll();     // ILLEGAL - UVAperson doesn't have enroll()!
s1.getAddress();// "Mailing Address (UVA)" - doesn't exist in Student so in UVAp
uG1.getJobOffer(); // "Ugrad-Got a job!" - Student inherits mthd, Ugrad version used
uG2.department(); // "Student-This is my major!" - UVAp has mthd, Student overrides it
                     - Ugrad inherits Student.department()
uG1.getAddress(); // "Mailing Address (UVA)" - UgradSt inherits from UVAperson
```

Run-time Polymorphism

Many forms: adapting behavior during run-time

Light-morph jaguar

Dark-morph or melanistic

Calling the getColor() method on two Jaguar objects

```
class LJaguar extends Jaguar {
     public String getColor() {
          return "I'm the light-morph one!";    }
}

class BJaguar extends Jaguar {
     public String getColor() {
          return "I'm the melanistic one!";   }
}

Jaguar j1 = new LJaguar();
Jaguar j2 = new BJaguar();
System.out.println(j2.getColor())
```

Who are you?

Where have we seen this polymorphic behavior before? → Writing the equals() method!

```
public boolean equals (Object other) {
     if (other == null)
          return false; // other is null
     if (other instanceof Dog) {
        Dog d2 = (Dog) other; // cast other to Dog
        if (this.name.equals(d2.name) &&
           this.breed.equals(d2.breed) &&
           this.age == d2.age) {
               return true; // two Dogs are equal
        }
        else
               return false; // two Dogs are not equal
     }
     else {   // other is not a Dog, so return false
        return false;
     }
}
```

.equals()

Is this a recursive call?

Object-Oriented (OO) Design & Programming

- Interfaces vs abstract classes vs non-abstract classes
- How to extend a class (inheriting methods, overriding methods, overriding abstract methods, use of super in constructor, use of super in other methods)
- Overriding vs. Overloading methods
  - **Overriding** – Different Class, Same Method Name, Same Parameters
  - **Overloading** – Same Class, Same Method Name, Different Parameters
- Java's Object class (in particular, the toString() and equals() methods)
- How Java decides which class's version of a method to invoke (run-time polymorphism)
- Abstract methods

Polymorphism Types

Type	Description
Method *overriding*	Methods of a subclass override the methods of a superclass
Method overriding of *abstract* methods	Methods of a subclass implement the abstract methods of an abstract class
Method overriding through Java *interfaces*	Methods of a concrete class implement the methods of an *interface*

Java Collections Framework

- Abstraction: Data and Procedural abstraction
- Collection interface, Collections library, List, Set, Map interfaces
- Design: inheritance hierarchies, abstract classes, interfaces

Event Driven Programming (& GUI)

- What is "event driven" programming?
- Be aware of some common components (by name & utility)
- What is an action listener, and does it mean to add an action listener to a component
- What is the purpose of the **actionPerformed**() method
- What is an anonymous class? What is an inner class?

GUI and Event Driven Programming

- What is "event driven" programming?
  - a **programming paradigm** in which the flow of the program is determined by **events** such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads
  - In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected

- Be aware of some common components (by name & utility)
  - E.g. JFrame, JPanel, JButton, JLabel, … etc

GUI and Event Driven Programming

- What is an action listener, and does it mean to add an action listener to a component

```
public interface ActionListener {
    public void actionPerformed(ActionEvent e);
}
```

- What is the purpose of the actionPerformed() method
  - When implementing the ActionListener's **actionPerformed()** method, you are writing the event handling method – telling the program what to do when an event occured

- What is an anonymous class? What is an inner class?

Handling Events in Swing

- To handle events in a GUI application, you'll need to:
  1. Create the interactive **graphical component** (e.g. a button)
  2. *(optional)* Set the `actionCommand` property on the **interactive component**
  3. Add your `actionListener` to the **interactive component**
  4. Create an **inner class** that implements the `ActionListener` **interface**

```
public interface ActionListener {
        public void actionPerformed(ActionEvent
e);
    }
```

More on this next class!

  5. Implement the ActionListener's **actionPerformed()** method – this is the actual event handling method

Example of adding a button (1)

Inside a class that `extends` **`JFrame`**

- Create an instance variable of type **`JButton`**

```
private JButton actionButton;
```

- Create/initialize the button, and set the `ActionCommand` (give "click" command)

```
actionButton = new JButton("Action");
actionButton.setActionCommand("click");
```

- Add the custom **`ActionListener`** to the component (button)

```
actionButton.addActionListener(new ButtonListener());
```

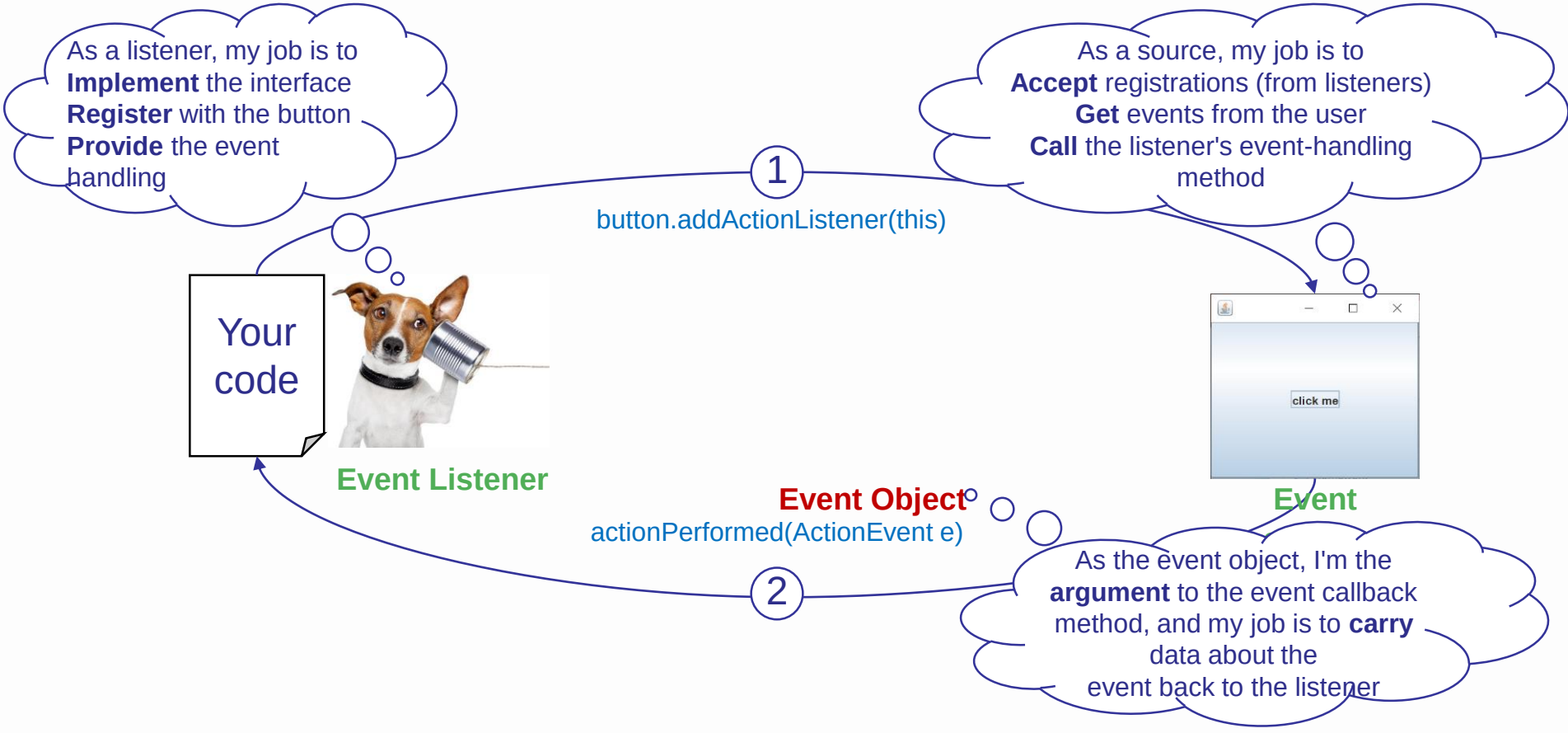
Example of adding a button (2)

- Create an **inner class** that **implements** the **ActionListener** **interface**

```
// create an inner class for the button action
class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("click")) {
                // ButtonListener triggered and the command was “click”
            infoLabel.setText("Button clicked"); // DO STUFF!
        }
    }
}
```

ActionListener (the *Interface*) requires one method: **actionPerformed**, which accepts an **event** object as a parameter. This class has the code that gets executed when the button is clicked (here, a *label* is updated).

Listeners, Sources, and Events (How to be a good listener)



Exceptions and Exception Handling

- What are exceptions?
- Identify and explain checked or unchecked exceptions
- Understand the **exception hierarchy** and how that factors into catching exceptions
  - (Catch more specific exceptions before more general exceptions!)

Exceptions and Exception Handling

- What are exceptions ("exceptional events")?
  - Events that disrupt the intended program flow
  - All exceptions must be detected and handled during coding
  - Exception handling provides a flexible mechanism for passing control from the point of error detection to a handler that can deal with the error.

Exceptions and Exception Handling

- Identify and explain checked or unchecked exceptions
  - Checked – by the compiler – "not in your hands"
    - **Checked** exceptions are due to **external** circumstances that the programmer cannot prevent.
      - The compiler checks that your program catches or handles these exceptions (program won't compile otherwise!)
  - Unchecked – not by the compiler – "your fault!" – related to the program that you should check for
    - The compiler does not check whether you handle an unchecked exception – **You are responsible for these exceptions!**

- Understand the exception hierarchy and how that factors into catching exceptions (Catch more specific exceptions before more general exceptions!)

Part of the Hierarchy of Exception Classes

Note:
All errors/ exceptions must inherit from **Throwable!**

Which means, the Throwable class is the superclass of all errors and exceptions in the Java language

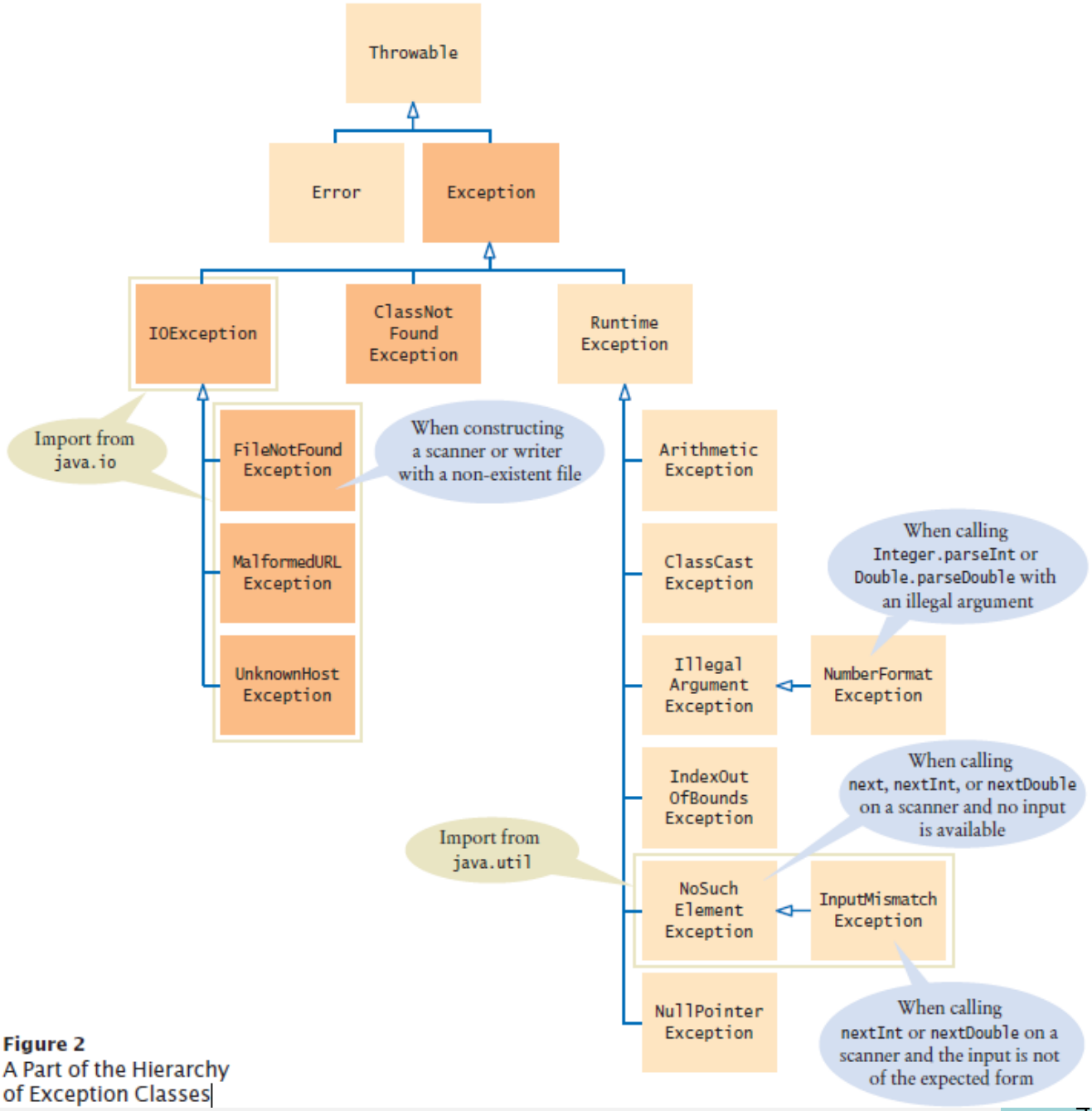
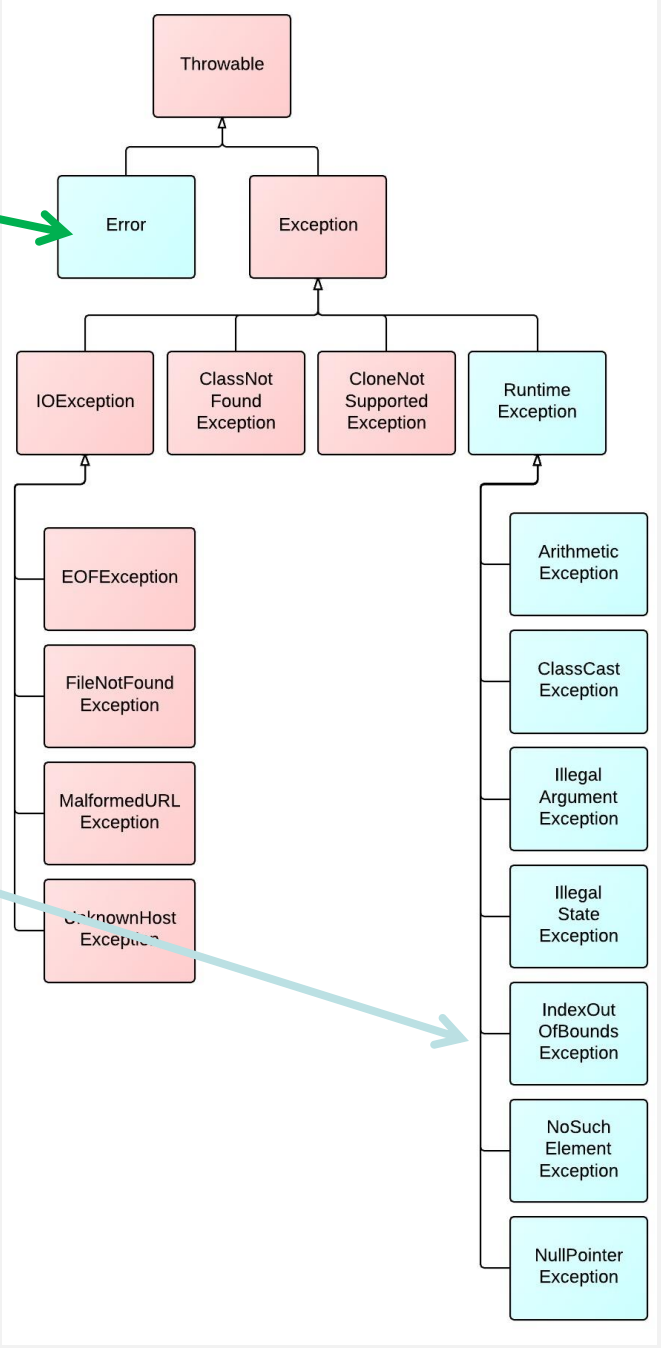


Figure 2
A Part of the Hierarchy of Exception Classes

- *Internal errors* - descendants of type error
- Red/pink colored are *checked* exceptions. Any checked exceptions that may be thrown in a method must either be **caught** or **declared** in the method's **throws** clause.
  - Checked exceptions must be caught at **compile time**. Checked exceptions are so called because both the Java compiler and the Java virtual machine check to make sure this rule is obeyed.
- Light blue colored are *uncheck* exceptions. They are exceptions that are not expected to be recovered, such as null pointer, divide by 0, etc.
  - You can still throw, catch, and declare them, but you don't have to and the compiler won't check!



Finding the Best Match

- Exceptions are ***polymorphic***!
- Java requires that **catch blocks** are arranged in ***decreasing precision***:
  - Most specific ErrorType to least specific.
- This works because of the ***Substitution Principle of inheritance.***
  - A subclass (child) exception can be assigned to an Exception (parent) reference (*child can go anywhere it's parent is accepted*)
- Just because you can catch everything with one big polymorphic catch (e.g. the throwable "warehouse", doesn't mean that you should!

IOException


Exception


Throwable

Catching Exceptions – Example

```
try {
//code that should run
}

catch ( ExceptionType e) {
//specific error handling code
}

…

catch (Exception e) {
// default error handling code
}
```

From ***most*** specific to ***least*** specific e.g., **FileNotFoundException** is a sub type of **IOException**

```
try {
 Scanner scannerfile = new Scanner( new
                                File("file.txt"));
}

catch (FileNotFoundException e) {
 System.out.println("File not found");
}


catch (IOException e) {
 System.out.println("Error reading the file");
}


catch (Exception e) {
 System.out.println("An error occurred");
}
```

Abstract Data Types (ADT) (Stacks and Queues)

- What is an ADT?
- **Stacks**
  - LIFO
  - Common stack operations: push(), pop(), peek()
- **Queues**
  - FIFO
  - Common queue operations: add(), remove()

ALGORITHMS

- Definition of "**algorithm**" and how to tell if something is one
  - An algorithm is a detailed step-by-step method for solving a problem
  - An algorithm's steps are precisely stated (no ambiguity; cannot be interpreted in more than one way), an algorithm is deterministic (behaves the same way), and an algorithm must terminate (stopping criteria/"the end" is clearly defined).

ALGORITHMS

- Define what a **critical section** of an algorithm is (understand that all algorithms have a critical section)
  - The critical section of an algorithm could be the entire algorithm (if relatively short), or it can be a subset (if a subset, it is always *consecutive* lines of code)
  - The part of the code that is often the most deeply nested
  - The part of the code that is often executed the most often
  - The part of the code that is fundamentally central to the working of that algorithm

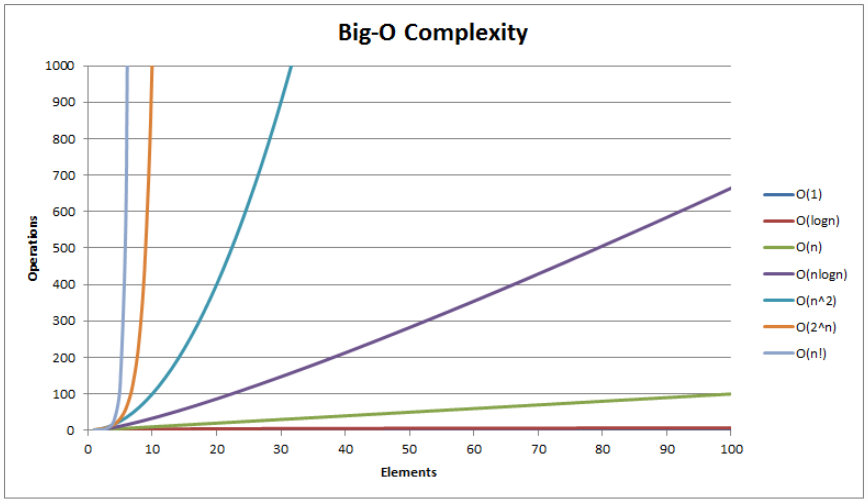
ALGORITHMS

- **Asymptotic analysis**
  - Big inputs vs small inputs. Worst-case inputs.
  - The meaning of Big-O (**not** Big Omega or Big Theta)
  - Comparison of common complexity classes (1, lg(n), n, n lg(n), n^2, n^3, …, 2^n)
  - Why we count operations or some **critical section** to get a formula f(n), and why we don't just time an algorithm
  - Given code (non-recursive only), find its complexity class and identify its critical section (*consecutive* lines of code)

Algorithms

- Order classes group "equivalently" efficient algorithms
  - Algorithms that grow at the SAME RATE

Class	Name	Example
O(1)	Constant time	Input size doesn't matter
O(log n)	Logarithmic time (Very efficient)	Binary search (on sorted list)
O(n)	Linear time	Linear search (unsorted list)
O(n log n)	Log-linear time	Best sorting algorithms
O(n^2)	Quadratic time	Poorer sorting algorithms
O(n^3)	Cubic time	3 variables equation solver
O(2^n)	Exponential time	Many important problems, often about optimization
O(n!)	Factorial time	Find all permutations of a given string/set



Increasing Complexity

Asymptotic Complexity (Example)

What is the time complexity of this algorithm?

```
public static int a (int[] array) {
    int sum = 0;
    for (int i = 0; i < array.length; i++) {
        for (int k = 0; k < 1000; k++) {
            sum += array[(i+k)%array.length];
        }
    }
    return sum;
}
```

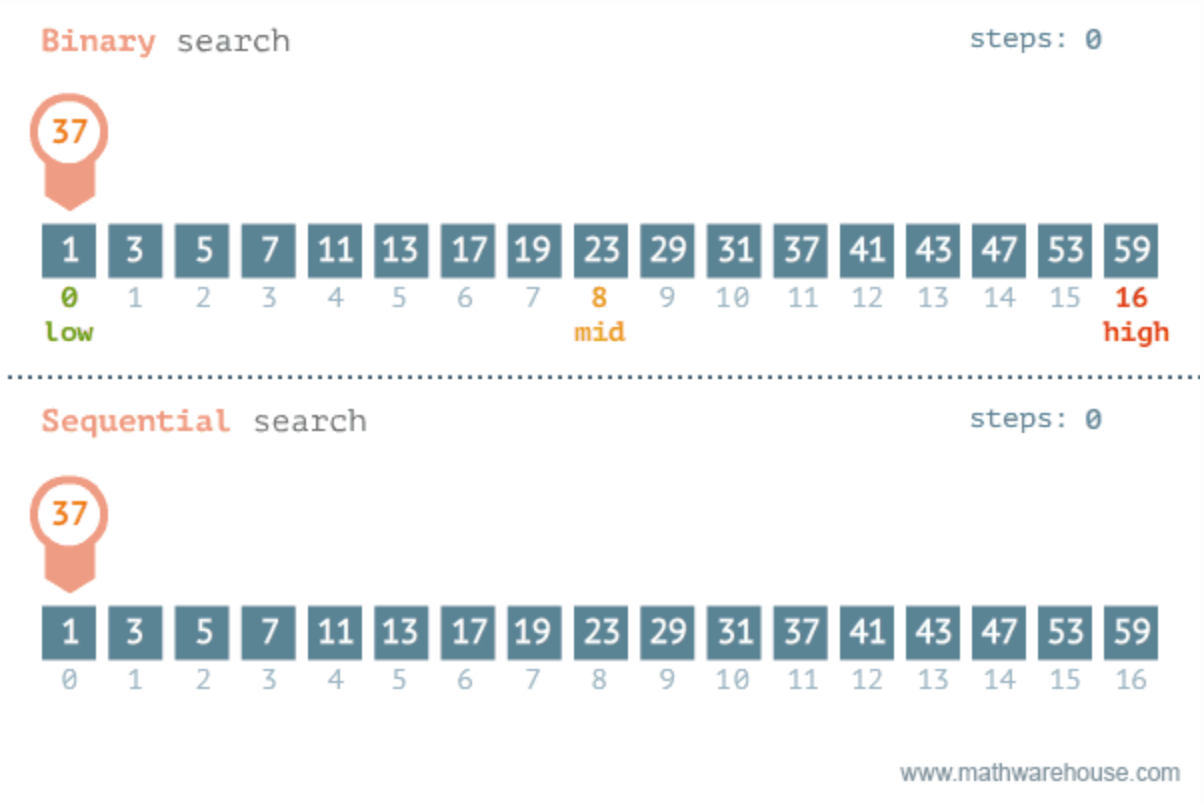
k is not dependent on the problem size!

Choice	Output
A	O(1)
B	O(log n)
C	O(n log n)
D	O(n)
E	$O(n^2)$
F	$O(n^3)$
G	$O(2^n)$

Algorithms: Searching and Sorting

- Searching
  - **Sequential / Linear search**: How does it work? What's its complexity? (**O(n)**) What's it's worst-case? Average case? Why? (You will not be expected to code this algorithm from scratch)
  - **Binary search**: How it works. What's its complexity? (**O(lg n)**) How does this compare to sequential search? What must be true of the list before we use this? (Sorted) (You will not be expected to code this algorithm from scratch)

Linear vs Binary Search



$O(\log_2 n)$

$O(n)$

https://www.mathwarehouse.com/programming/gifs/binary-vs-linear-search.php

Discussion Question: Given an unsorted list…

- **Binary search** is faster than sequential search
  - But extra cost! Must sort the list first!
  - It costs O(n log n) to sort --- if we use fastest sorting algorithm
  - Sorting + searching once = O(n log n) + O(log n) = O(n log n)
  - Linear search once = O(n)

So, when do you think it's worth using binary search?

- When you are searching many times, it's more efficient to sort once then perform MANY ***binary searches*** (compared to many *linear searches*)
- How many searches? Binary search overtakes the cost of the sort **after "*n*" searches!**

Comparison (Summary: Searching)

- Linear search vs. Binary search

Sequential (Linear) Search

Case	Best Case	Worst Case	Average Case
target present	$O(1)$	$O(n)$	$O(n)$
target not present	$O(n)$	$O(n)$	$O(n)$

Binary Search

Case	Best Case	Worst Case	Average Case
target present	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$
target not present	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$

Algorithms: Searching and Sorting

- Sorting
  - **Good sorting algorithms are in the O(n lg n) complexity** class
    (e.g. Quicksort and MergeSort).
  - Poorer (not as efficient) sorting algorithms are in the $O(n^2)$ complexity class
    (e.g. Insertion sort, Selection sort, Bubble sort)

Hashing / Hash Tables / Collision Resolution Techniques

- Properties of hash functions
- Operations on Hash Tables
  - Adding
  - Removing
  - Contains
- **Collision resolution techniques**
  - **Linear probing**
    - Flags(E-empty; D-deleted; O-occupied)
  - **Separate chaining**
- Understand how the operations of adding/removing/searching are linked to the collision resolution technique chosen

Hashing [Lab]

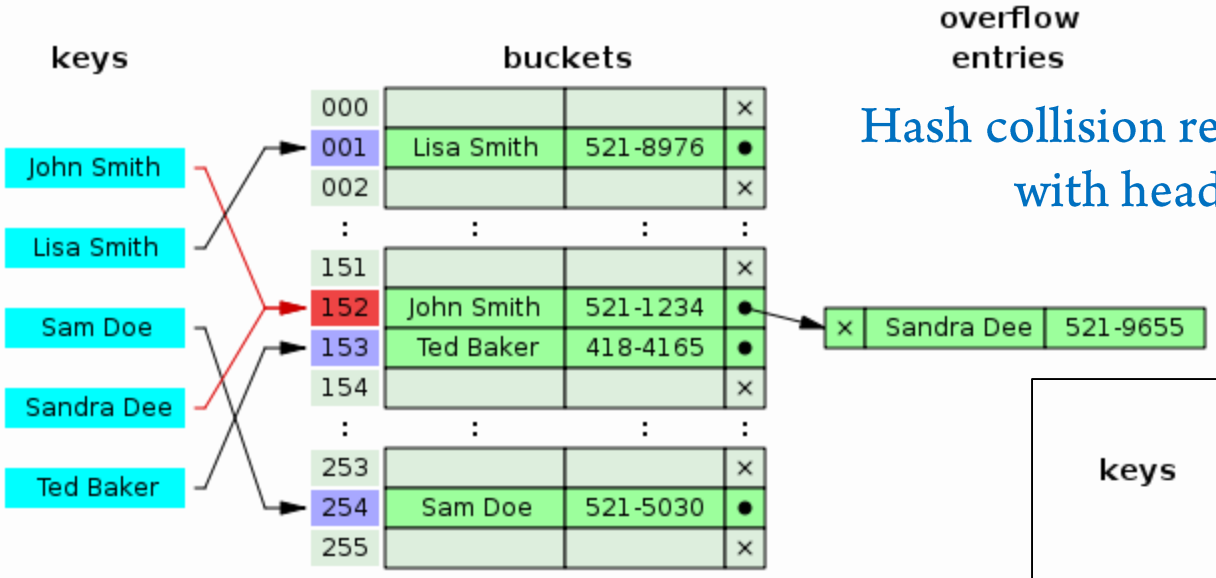
- Hashing and Hash Tables (This material comes directly from the Hashing lab - Lab 4.)
  - Properties of hash functions
  - Operations on Hash Tables
    - Adding
    - Removing
    - Contains
  - Collision resolution techniques
    - Linear probing
      - Flags(E-empty; D-deleted; O-occupied)
    - Separate chaining
  - Understand how the operations of adding/removing/searching are linked to the collision resolution technique chosen

Hashing

- A hash table (also hash map) is a data structure used to implement an associative array, a structure that can map keys to values
- A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found
- If it is a uniform hash, it is equally likely to go into any particular bucket

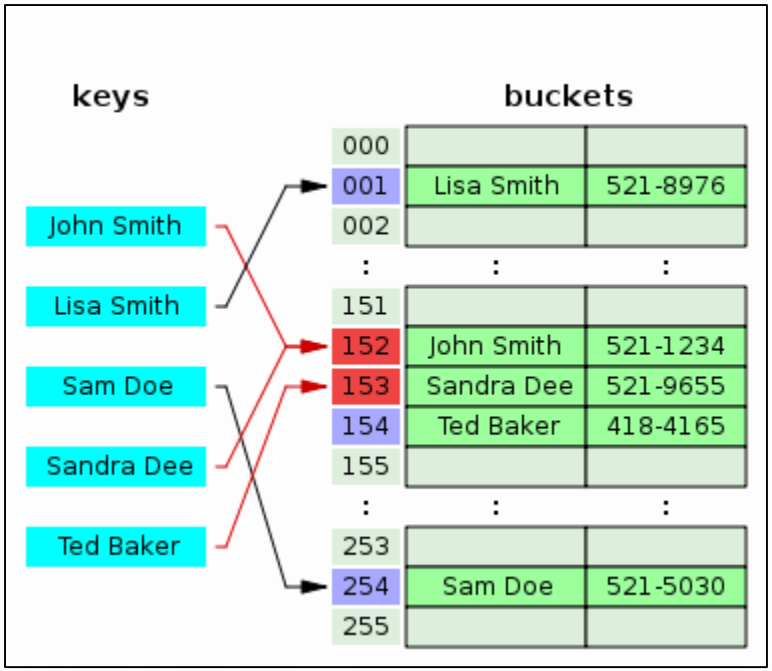
- *Ideally*, the hash function will assign each key to a unique bucket, but this ideal situation is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after it is created)
- Instead, most hash table designs assume that **hash collisions**—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way

Collision Resolution Techniques



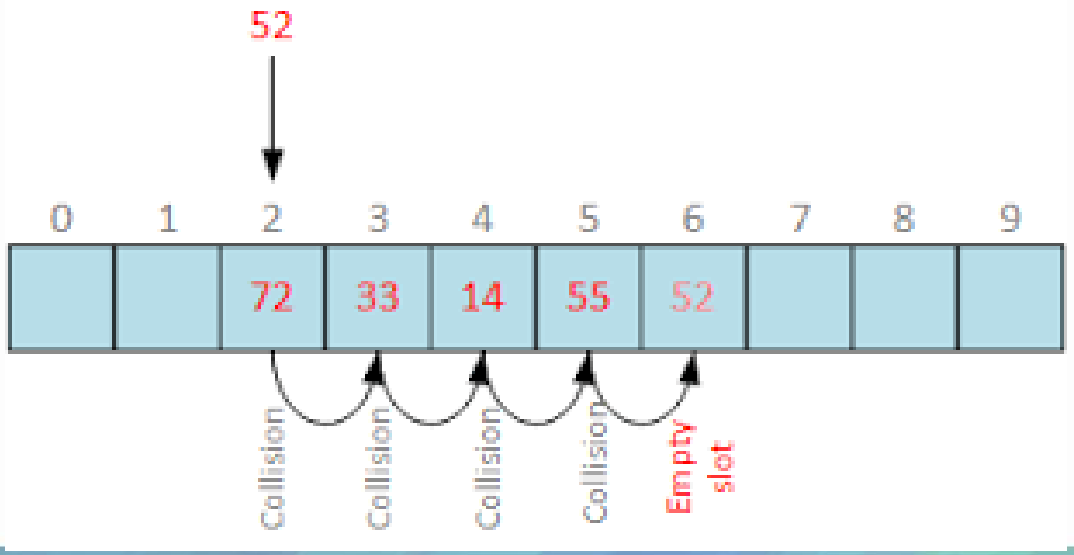
Hash collision resolved by separate chaining with head records in the bucket array

Hash collision resolved by open addressing with linear probing (interval=1). Note that "Ted Baker" has a unique hash, but nevertheless collided with "Sandra Dee", that had previously collided with "John Smith"



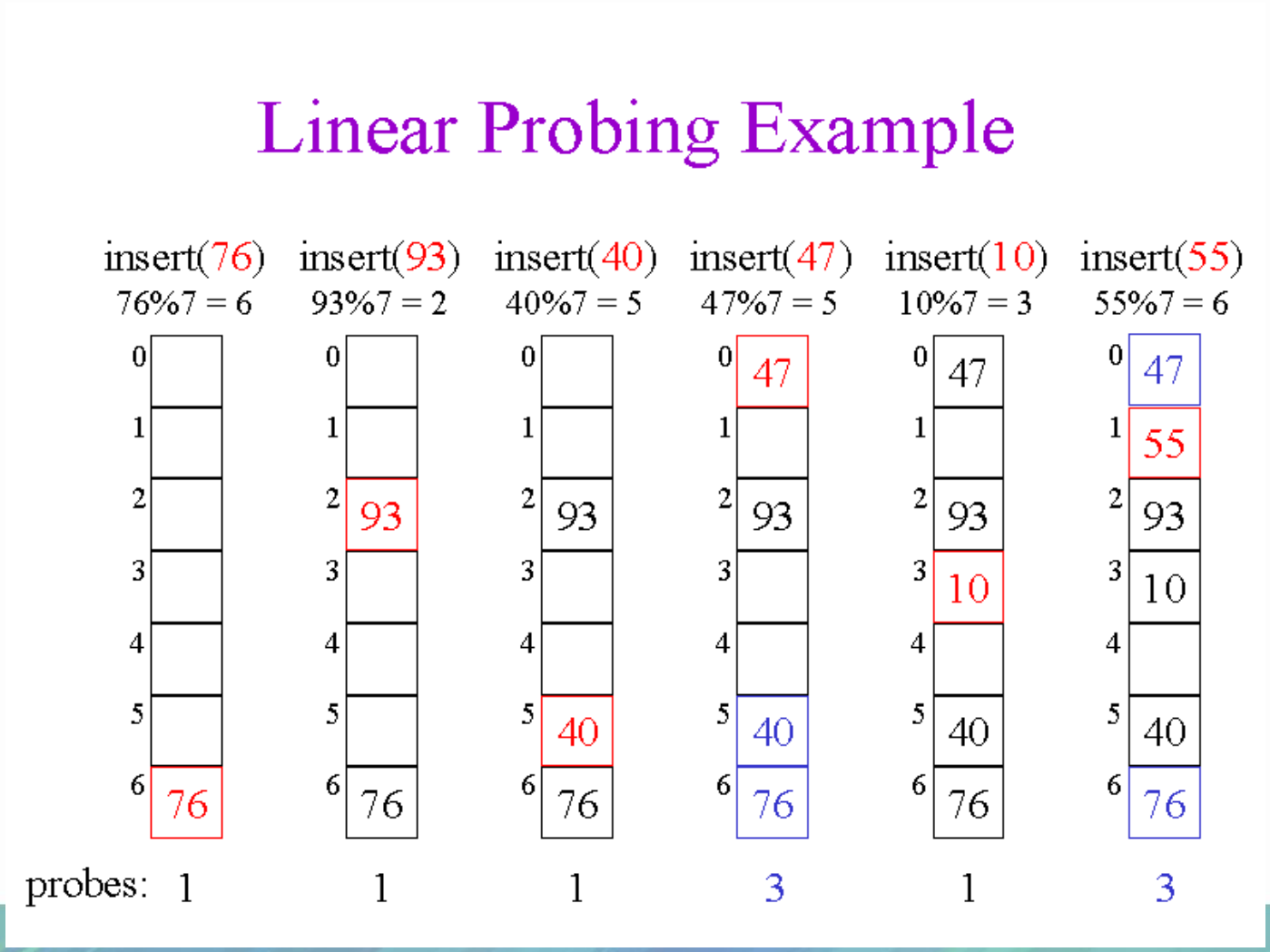
Example : Linear Probing

- If the item 52 hashes to index 2, there is a collision
- To find a spot for it, probe linearly (starting at the index at which it originally hashed to) until you find an empty spot ("E" for empty)
- Search: start at hashed index, if not found linearly prob until get to empty spot
- What if we are searching for 52 (now in index 7) but prior to that, 14 was removed?



Another Example

- [From cs.washington.edu]




Step by step you will achieve your goals!
Good Luck!


YOU GOT THIS
quickmeme.com

You Got This! ☺