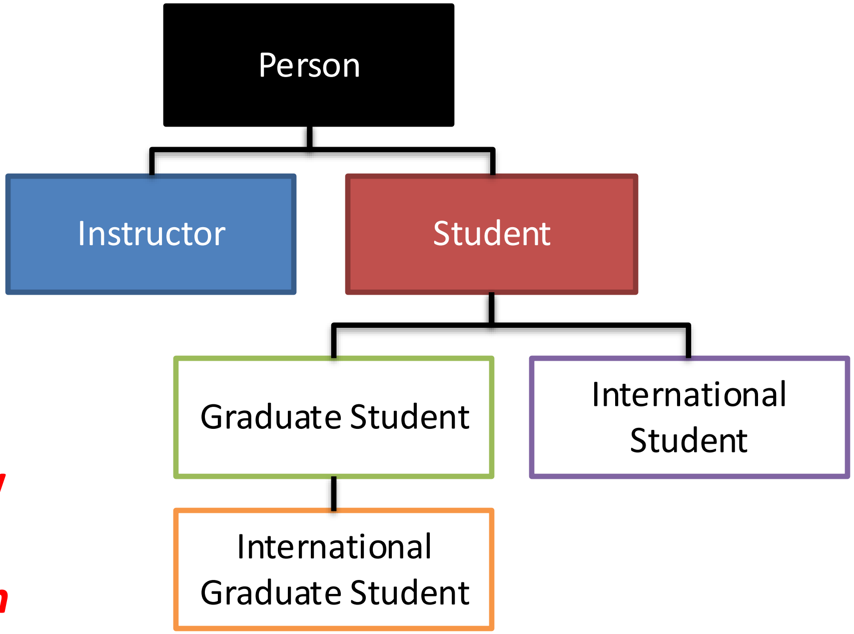
CS 10:
Problem solving via Object Oriented Programming

Abstraction

A note about inheritance: you can declare base class and instantiate as subclass




```java
Person bob = new Instructor("Bob", "f00000");
Person carol = new Student("Carol", "f11111");
```

***Bob* and *Carol* are declared as *Person* objects, but instantiated as *Instructor* and *Student* respectively**

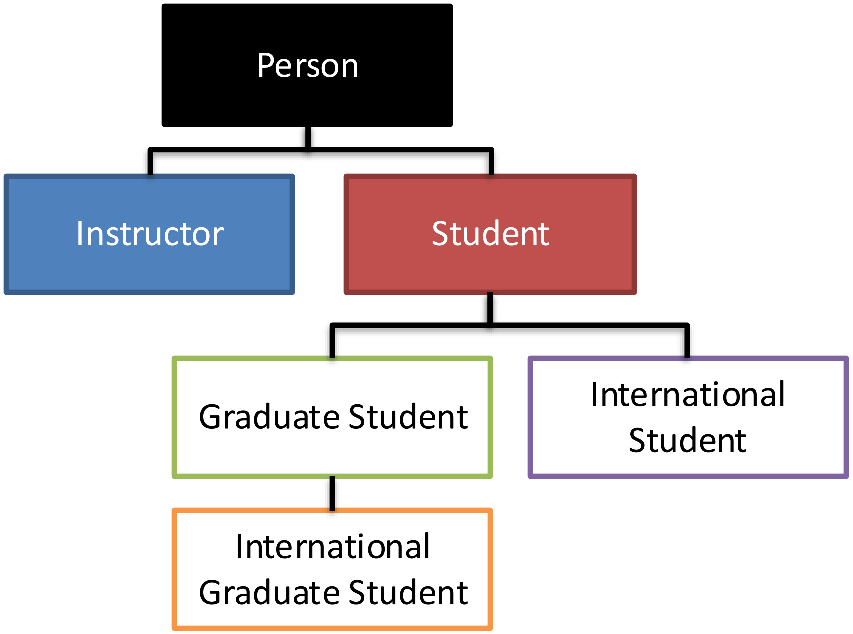
An *Instructor* "is a" *Person*, a *Student* "is a" *Person* so Java allows this declaration

Why would we ever do such insanity?!?!?

So we can store items in an array, or as we'll see today, in a List

We can declare a List of *Person* objects that will allow us to hold *Instructors* and *Students*

You can *cast* to the instantiated type

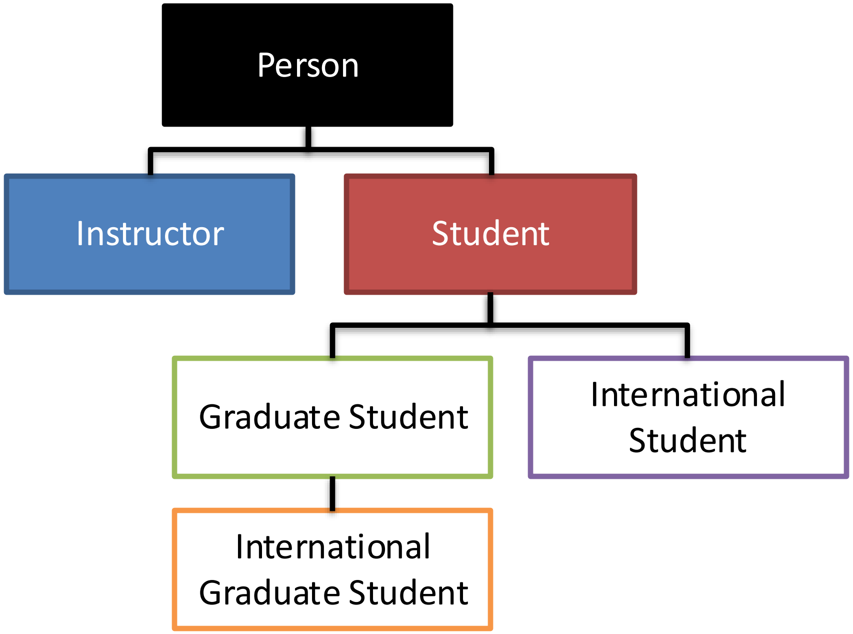



```
Person bob = new Instructor("Bob", "f00000");
Person carol = new Student("Carol", "f11111");

((Instructor) bob).tenured = true;
((Student) carol).graduationYear = 2027;
```

To use subclass specific functionality, we must *cast* to the subclass

Cast *bob* as an *Instructor* to access *tenured* (*Person* doesn’t have *tenured*)
Cast *carol* as a *Student* to access *graduationYear* (*Person* doesn’t have *graduationYear*)

Cannot cast to a type outside the inheritance chain

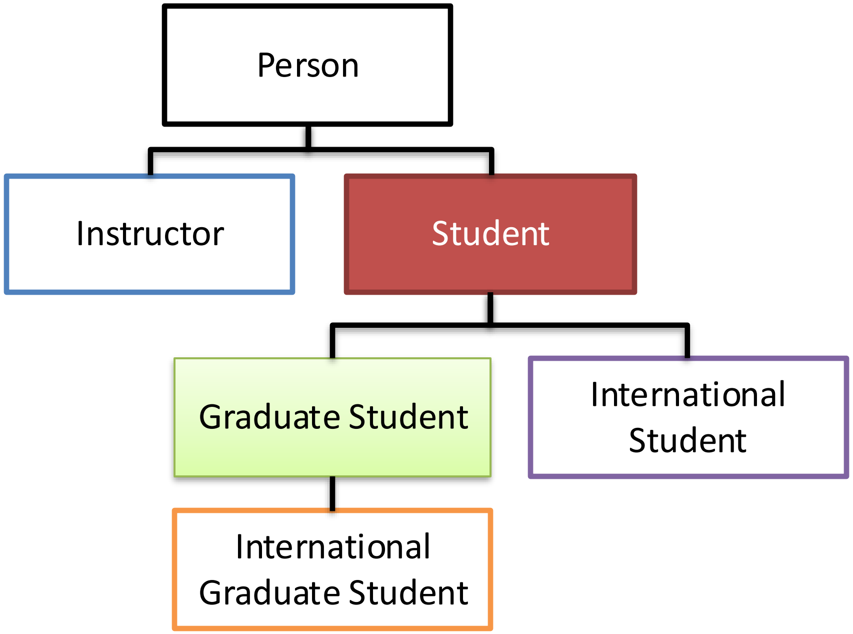



```
Person bob = new Instructor("Bob", "f00000");
Person carol = new Student("Carol", "f11111");

((Instructor) bob).tenured = true;
((Student) carol).graduationYear = 2027;

((Student) bob).graduationYear = 2028;
```

Output:
Exception: class Instructor cannot be cast to class Student

Can't cast an object to a subclass outside its inheritance chain
bob* is instantiated as an *Instructor*, can't cast as a *Student

Cannot cast down the inheritance chain



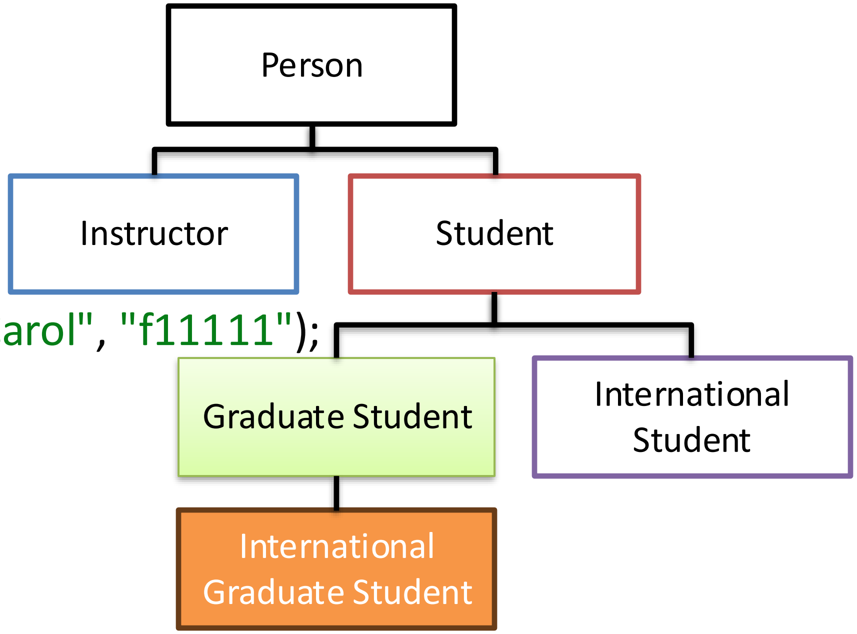

```
Person bob = new Instructor("Bob", "f00000");
Person carol = new Student("Carol", "f11111");

((Instructor) bob).tenured = true;
((Student) carol).graduationYear = 2027;

((GraduateStudent) carol).graduationYear = 2028;
```

Output:
class Student cannot be cast to class GraduateStudent

Cannot cast down the inheritance chain

A *Student* is not necessarily a *GraduateStudent*
(but a *GraduateStudent* is a *Student!*)

Can cast up the inheritance chain




```
Person bob = new Instructor("Bob", "f00000");
Person carol = new InternationGraduateStudent("Carol", "f11111");

((Instructor) bob).tenured = true;
((Student) carol).graduationYear = 2027;

((GraduateStudent) carol).graduationYear = 2028;
```

Can cast up the inheritance chain

If *carol* where an *InternationalGraduateStudent*, could be cast to a *GraduateStudent*

An *InternationalGraduateStudent* is a *GraduateStudent*

Agenda

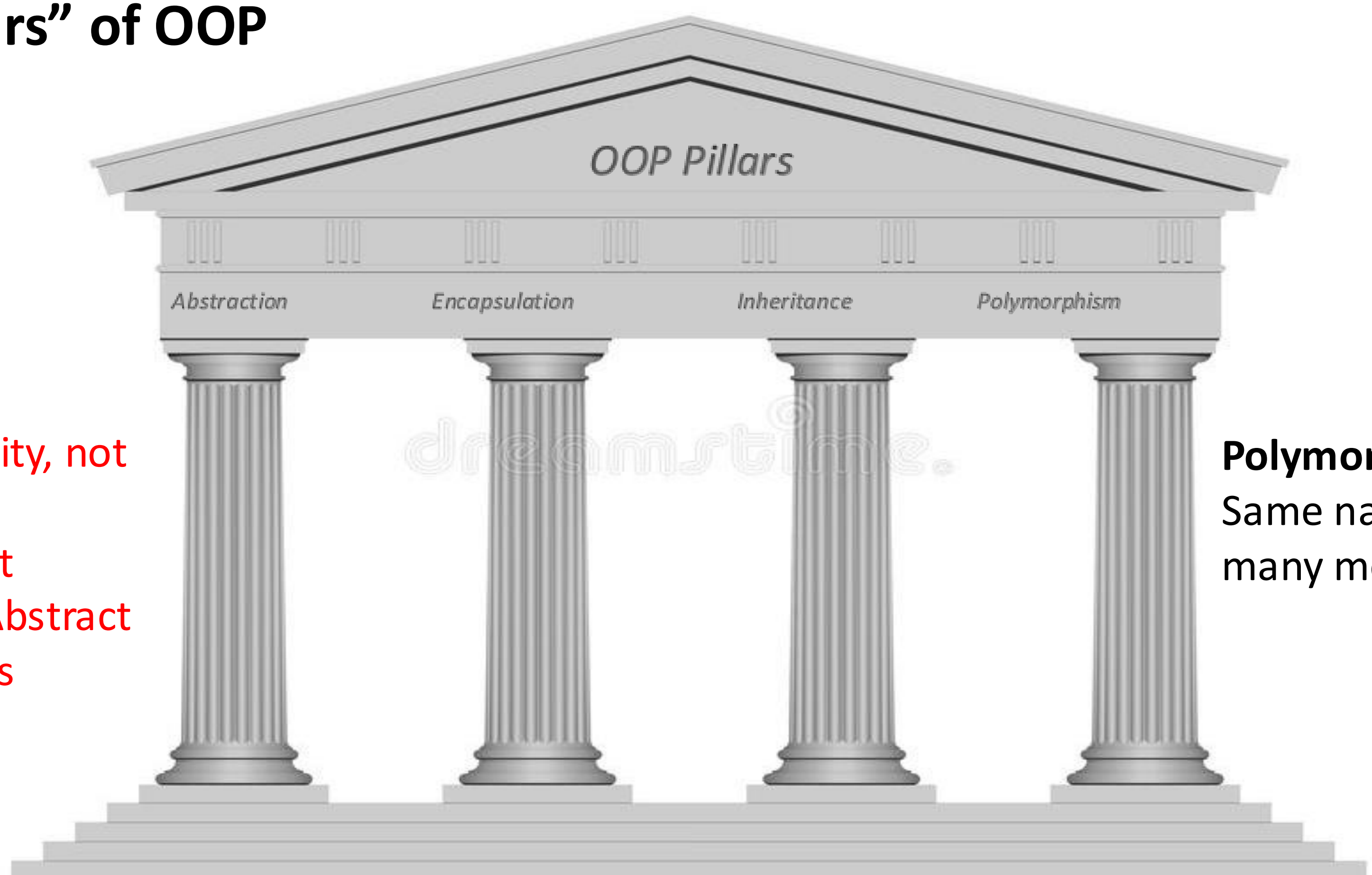
1. ADTs
2. Generics
3. Java provided List implementation
4. Run-time complexity
5. Asymptotic notation

Key points:

1. **ADTs say *what* needs to be done, but not *how* do implement it**
2. **An *interface* describes methods must be implemented for an ADT**

OOP relies on four main pillars to create robust, adaptable, and reusable code

Four “pillars” of OOP



Abstraction

- Name functionality, not how to implement
- Leads to Abstract Data Types (ADTs)

Polymorphism

Same name, many meanings

Encapsulation

- Bind code and data into one thing called an object
- Code called methods in OOP (not functions)

Inheritance

- Create specialty versions that “inherit” functionality of parent
- Reduces code

Abstract Data Types specify operations on a data set that defines overall behavior

Abstract Data Types (ADTs)

- ADTs specify a set of ***operations*** (e.g., *get, set, add*, …) that define how the ADT behaves ***on a collection*** of data elements

- At the ADT level we don't know (and don't really care) what data structure is used to store elements (e.g., linked list or array or something else, it doesn't matter at an abstract level)

- Also do not care about what kind of data the ADT holds (e.g., Strings, integers, Objects) – the ADT works the same way regardless of what type of data it holds

- **Big idea: hide the way data is represented and manipulated while allowing others to work with the data in a consistent manner**

The List ADT defines required operations, but not how to implement them

List ADT

Operation	Description
`size()`	Return number of items in List
`isEmpty()`	True if no items in List, otherwise false
`get(i)`	Return the item at index *i*
`set(i,e)`	Replace the item at index *i* with item *e*
`add(e)`	Add item *e* to end of the list
`add(i,e)`	Insert item *e* at index *i*, moving all subsequent items one index larger
`remove(i)`	Remove and return item at index *i*, move all subsequent items one index smaller

Big idea: List works the same regardless of what data structure it uses to store data or what type of data it holds

These operations MUST be implemented to complete the ADT

Free to implement other methods, but must have these

We never say how many items the list can hold; it grows as needed

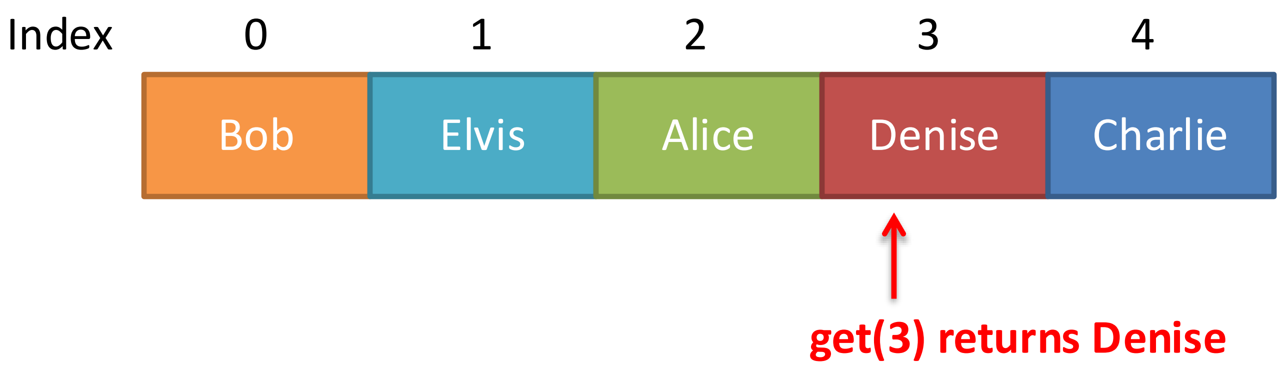
Example: List ADT defines a set of operations

List holds multiple elements (items) referenced by position in List

Index	0	1	2	3	4
	Bob	Elvis	Alice	Denise	Charlie

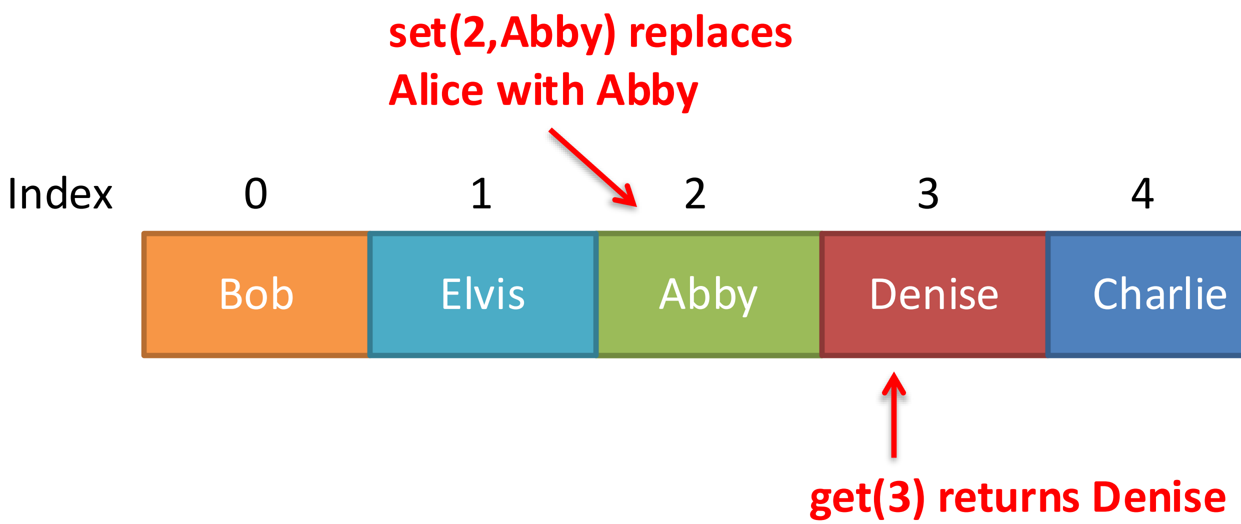
Example: List ADT defines a set of operations

List holds multiple elements (items) referenced by position in List



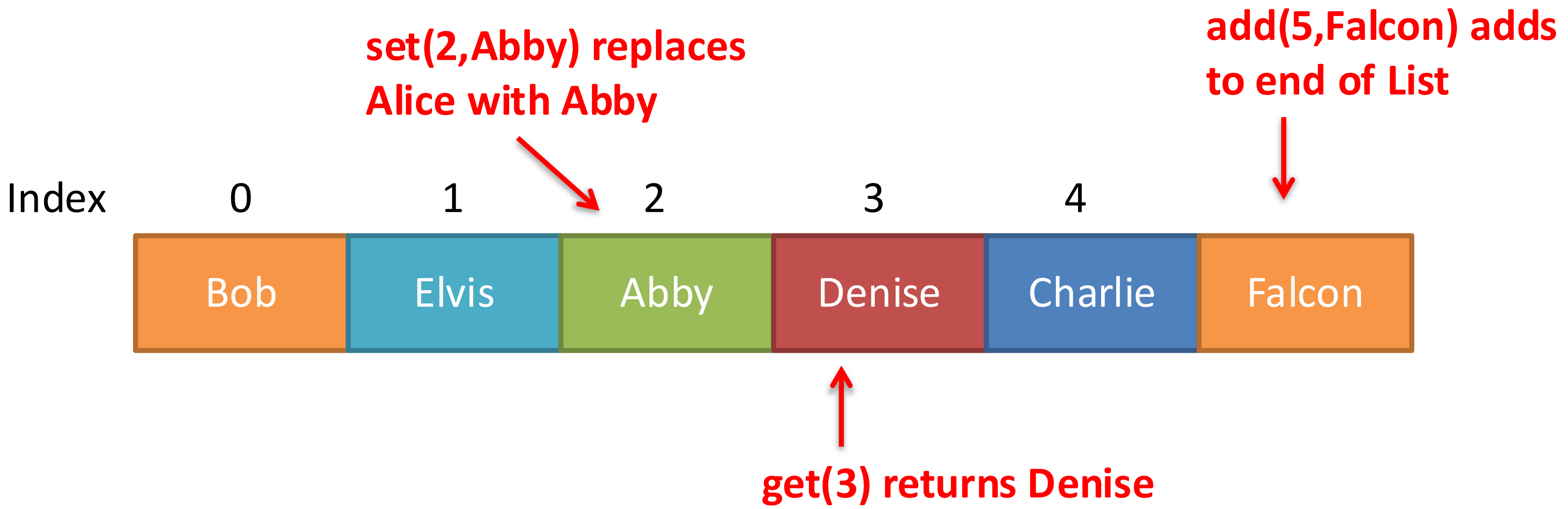
Example: List ADT defines a set of operations

List holds multiple elements (items) referenced by position in List



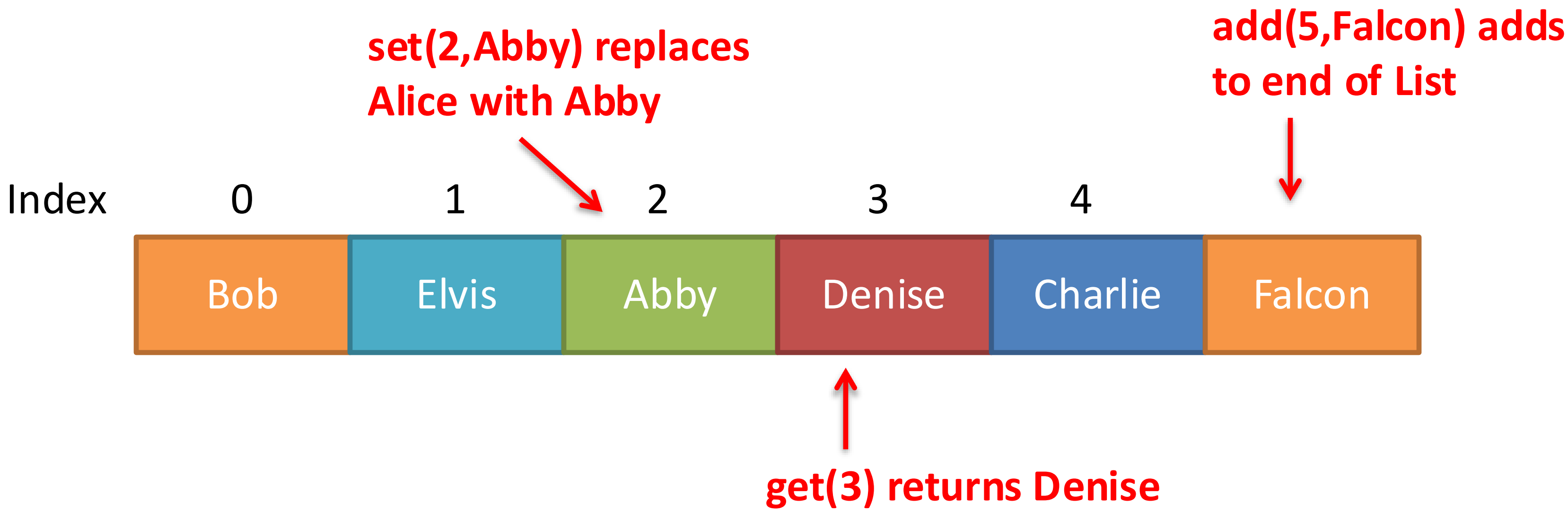
Example: List ADT defines a set of operations

List holds multiple elements (items) referenced by position in List



Example: List ADT defines a set of operations

List holds multiple elements (items) referenced by position in List

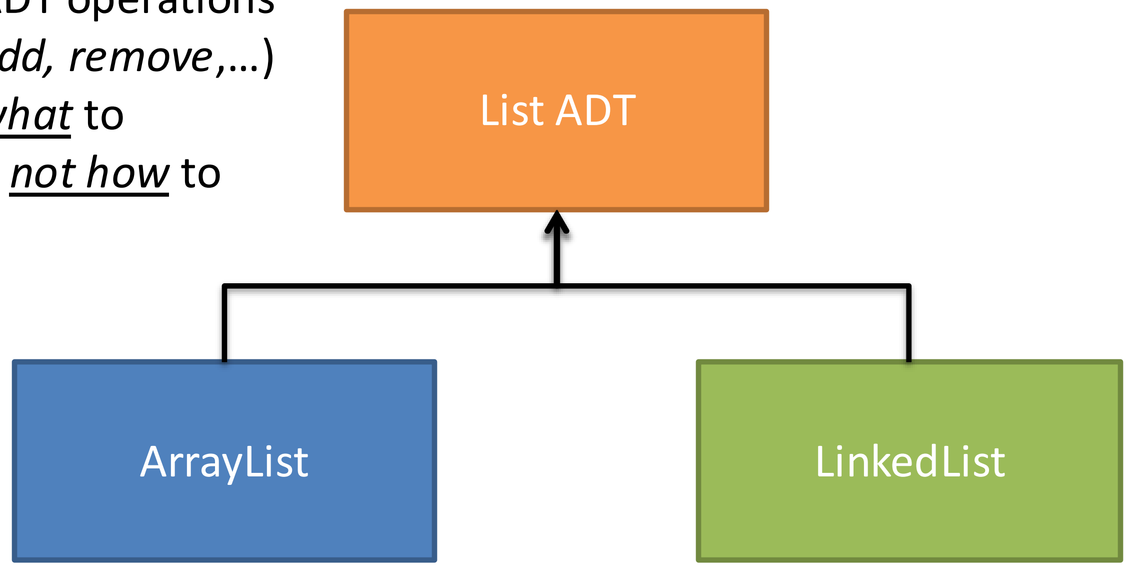


- **ADT defines these operations (and others)**
- **What data structure does it use? Array? Linked List?**
  - We don't know and don't care at the abstract level, we just care that the operations (get, set, add, remove, size, isEmpty) work as expected
- **What type of elements are these? Strings, Integers, Student Objects?**
  - See answer above – we don't care
  - The type of element does not affect how the ADT works!

ADTs can be implemented differently, but must provide common functionality

Java Interface:

- Defines set of ADT operations (e.g., *get, set, add, remove*,…)
- Interface says *what* to implement, but *not how* to implement



Elements stored in an Array

Elements stored in a Linked List

Implementation:

- Code to implement operations that are defined by interface
- Can be written using different data structures
- MUST implement all functionality defined by interface
- But you can include other functionality

Java has both ArrayList and LinkedList implementations of List
Both implementations provide the same functionality as required by interface, but store data differently

We will implement the List interface using both approaches

Interfaces go in one file, implementations go in another file(s)

Interface file
Specifies required operations
`SimpleList.java`

Uses keyword `interface`

Linked list implementation

OR

Array implementation

Implementation file
Actually implements required operations using a specific data structure

Same interface <u>*could*</u> be implemented in different ways (e.g., linked list *or* array)

Use keyword `implements` to implement an `interface`

SimpleList.java is an interface that specifies what operations MUST be implemented

```java
public interface SimpleList<T> {
  /**
   * Returns # elements in the List (they are indexed 0..size-1)
   */
  public int size();

  /**
   * Returns true if there are no elements in the List, false otherwise
   * @return true or false
   */
  public boolean isEmpty();

  /**
   * Adds the item at the index, which must be between 0 and size
   */
  public void add(int idx, T item) throws Exception;

  /**
   * Add item at end of List
   */
  public void add(T item) throws Exception;

  /**
   * Removes and returns the item at the index, which must be between 0 and size-1
   */
  public T remove(int idx) throws Exception;

  /**
   * Returns the item at the index, which must be between 0 and size-1
   */
  public T get(int idx) throws Exception;

  /**
   * Replaces the item at the index, which must be between 0 and size-1
   */
  public void set(int idx, T item) throws Exception;
```

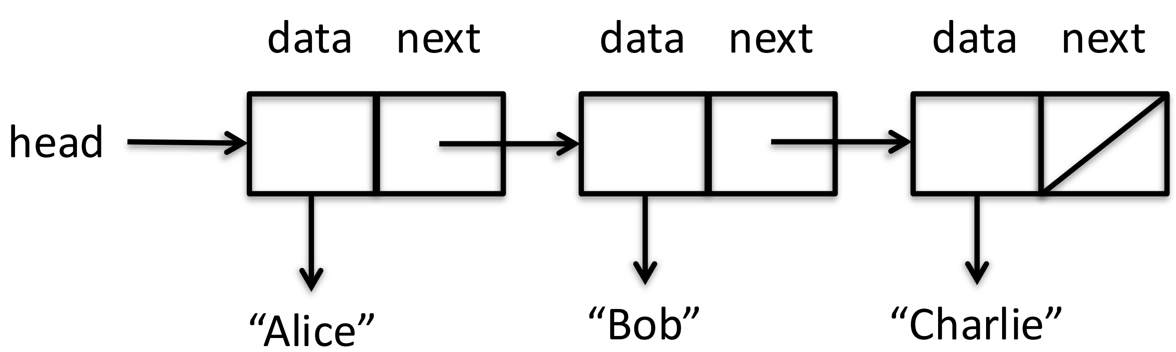
Interface keyword tells Java this is an interface

- **Standby for what "T" and "throws Exception" mean**
- **Methods defined to include parameters and return types (called a "signature"), no implementation code! here**
- **If you are going to implement SimpleList, then you MUST implement these methods**
- **How you implement (use array, linked list, …) is your business**
- **Java's List interface has a few more methods, ours simplifies things a little**
- **Why bother with an interface?**

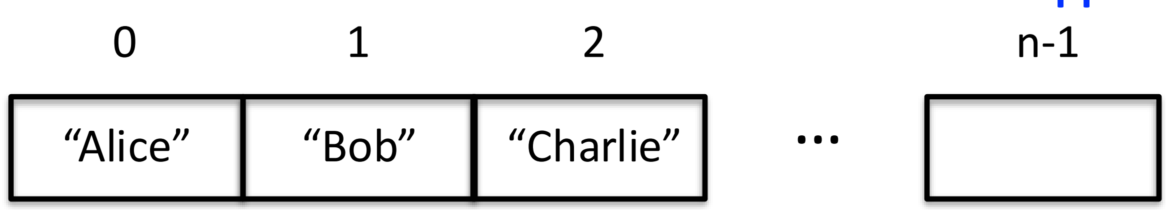
The List ADT could be *implemented* with a singly linked list *OR* an array; either works

Examples of List implementation

Singly linked list



Array



- **We will implement List ADT both ways**
- **Each implementation has pros and cons**
- **Java has built-in version of the List ADT – ArrayList and LinkedList**
- **We will create our own two versions to contrast approaches**

Agenda

1. ADTs
2. Generics
3. Java provided List implementation
4. Run-time complexity
5. Asymptotic notation

Key points:

1. **Generics allow us to write an ADT one time, irrespective of the data types involved**

Generics allow a variable to stand in for a Java type

```
public interface SimpleList<T> {
   public T get(int idx) throws Exception;
   public void add(int idx, T item) throws Exception;
```

- T stands for whatever object *type* we instantiate
- With SimpleList<Student> list = new ArrayList<Student>(); then T always stands for Student
- SimpleList<Point> then T always stands for Point
- Allows us to write *one* implementation that works regardless of what kind of object we store in our data set
- Must use autobox version of primitives (Integer, Double, etc)
- By convention we name type of variables with a single uppercase letter, often T for “type”, later we’ll use K for key and V for value

Agenda

1. ADTs
2. Generics
3. Java provided List implementation
4. Run-time complexity
5. Asymptotic notation

Key points:

1. **Java provides two implementations of the List ADT, an ArrayList and a Linked List**
2. **Each implementation provides the same ADT operations, but work differently**
3. **We will soon implement both ourselves to see how they work**

ArrayListDemo.java: ArrayLists can hold multiple objects; provide useful methods

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<Integer>();
      list.add(1);
      list.add(2);
      list.add(1,3);
      System.out.println(list);
      Integer b = list.get(1);
      System.out.println(b);
      list.remove(1);
      System.out.println(list);
      list.set(1,4);
      System.out.println(list);
      System.out.println(list.size());
   }
}
```

Java provides the ArrayList
We will write our own version of the *List* ADT using:

- **Array**
- **Linked list**

- **Declare object of type *List* on left hand side**
- **On right hand side, *new* instantiates an object of type *ArrayList***
- **Later if we decide a *LinkedList* implementation of the *List* ADT would be better, we simply change from *ArrayList* to *LinkedList***
- **In following code, we just call methods defined by the *List* ADT**
- **Here Java will use the *ArrayList* implementation**
- **If we changed *ArrayList* to *LinkedList*, Java would use the *LinkedList* implementation, but the result would be the same**

ArrayListDemo.java: ArrayLists can hold multiple objects; provide useful methods

Must import *Arraylist* (code is not in our project)

- **IntelliJ Settings/Preferences**
- **Select Editor->General->Auto Import**
- **Check the "Add unambiguous imports on the fly"**

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(1,3);
        System.out.println(list);
        Integer b = list.get(1);
        System.out.println(b);
        list.remove(1);
        System.out.println(list);
        list.set(1,4);
        System.out.println(list);
        System.out.println(list.size());
    }
}
```

ArrayListDemo.java: ArrayLists can hold multiple objects; provide useful methods

```java
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(1,3);
        System.out.println(list);
        Integer b = list.get(1);
        System.out.println(b);
        list.remove(1);
        System.out.println(list);
        list.set(1,4);
        System.out.println(list);
        System.out.println(list.size());
    }
}
```

- **Provide type of objects ArrayList will hold in <> brackets (can't be primitive)**
- **Integer is the object version of int**
- **Lists can hold only *one* type of object (unlike Python)**
- **Lists are called generic containers because they can hold any type of object (Integers, Doubles, Strings, Students)**
- **Don't need to specify length of List, it can grow as need (unlike an array)**

ArrayListDemo.java: ArrayLists can hold multiple objects, provide useful methods

```java
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(1,3);
        System.out.println(list);
        Integer b = list.get(1);
        System.out.println(b);
        list.remove(1);
        System.out.println(list);
        list.set(1,4);
        System.out.println(list);
        System.out.println(list.size());
    }
}
```

- ***add(E elmt)* appends item to end of *List***
  - ***E* = type (Integer here)**
  - ***elmt* = object (element) to add to the end of the *List***
- **Note: this call does not specify an index to the new item, so add at the end**

ArrayListDemo.java: ArrayLists can hold multiple objects, provide useful methods

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(1,3);
        System.out.println(list);
        Integer b = list.get(1);
        System.out.println(b);
        list.remove(1);
        System.out.println(list);
        list.set(1,4);
        System.out.println(list);
        System.out.println(list.size());
    }
}
```

- ***add(E elmt)*** **appends item to end of** ***List***
  - ***E*** **= type (Integer here)**
  - ***elmt*** **= object (element) to add to the end of the** ***List***
- **Note: this call does not specify an index to the new item, so add at the end**

ArrayList list

1

ArrayListDemo.java: ArrayLists can hold multiple objects, provide useful methods

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(1,3);
        System.out.println(list);
        Integer b = list.get(1);
        System.out.println(b);
        list.remove(1);
        System.out.println(list);
        list.set(1,4);
        System.out.println(list);
        System.out.println(list.size());
    }
}
```

- ***add(E elmt)* appends item to end of *List***
  - ***E* = type (Integer here)**
  - ***elmt* = object (element) to add to the end of the *List***
- **Note: this call does not specify an index to the new item, so add at the end**

ArrayList list

1	2

ArrayListDemo.java: ArrayLists can hold multiple objects, provide useful methods

```java
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<Integer>();
      list.add(1);
      list.add(2);
      list.add(1,3);
      System.out.println(list);
      Integer b = list.get(1);
      System.out.println(b);
      list.remove(1);
      System.out.println(list);
      list.set(1,4);
      System.out.println(list);
      System.out.println(list.size());
   }
}
```

- ***add(int i, E elmt)* adds item at index *i***
- ***Lists* are zero indexed (start at index 0, unlike Matlab)**
- **Items slide right to make room**

ArrayList list

1	2

ArrayListDemo.java: ArrayLists can hold multiple objects, provide useful methods

```java
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
  public static void main(String[] args) {
    List<Integer> list = new ArrayList<Integer>();
    list.add(1);
    list.add(2);
    list.add(1,3);
    System.out.println(list);
    Integer b = list.get(1);
    System.out.println(b);
    list.remove(1);
    System.out.println(list);
    list.set(1,4);
    System.out.println(list);
    System.out.println(list.size());
  }
}
```

- ***add(int i, E elmt)* adds item at index *i***
- ***Lists* are zero indexed (start at index 0, unlike Matlab)**
- **Items slide right to make room**
- ***add* method is overloaded (two versions that take different parameters)**

ArrayList list

1	3	2

ArrayListDemo.java: ArrayLists can hold multiple objects, provide useful methods

```java
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(1,3);
        System.out.println(list);
        Integer b = list.get(1);
        System.out.println(b);
        list.remove(1);
        System.out.println(list);
        list.set(1,4);
        System.out.println(list);
        System.out.println(list.size());
    }
}
```

Printing a List calls *toString* behind the scenes
The designers of Java have already written this method for the ArrayList class

Output

[1, 3, 2]

ArrayList list

1	3	2

ArrayListDemo.java: ArrayLists can hold multiple objects, provide useful methods

```java
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<Integer>();
      list.add(1);
      list.add(2);
      list.add(1,3);
      System.out.println(list);
      Integer b = list.get(1);
      System.out.println(b);
      list.remove(1);
      System.out.println(list);
      list.set(1,4);
      System.out.println(list);
      System.out.println(list.size());
   }
}
```

- **ArrayLists provide random access (can get item from anywhere)**
- ***get(int i)*** **returns item at index *i***
- **Remember zero-based indexing!**

Output

[1, 3, 2]

ArrayList list

1	3	2

ArrayListDemo.java: ArrayLists can hold multiple objects, provide useful methods

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<Integer>();
      list.add(1);
      list.add(2);
      list.add(1,3);
      System.out.println(list);
      Integer b = list.get(1);
      System.out.println(b);
      list.remove(1);
      System.out.println(list);
      list.set(1,4);
      System.out.println(list);
      System.out.println(list.size());
   }
}
```

- **ArrayLists provide random access (can get item from anywhere)**
- ***get(int i)* returns item at index *i***
- **Remember zero-based indexing!**

Output

[1, 3, 2]
3

ArrayList list

1	3	2

ArrayListDemo.java: ArrayLists can hold multiple objects, provide useful methods

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(1,3);
        System.out.println(list);
        Integer b = list.get(1);
        System.out.println(b);
        list.remove(1);
        System.out.println(list);
        list.set(1,4);
        System.out.println(list);
        System.out.println(list.size());
    }
}
```

- **Can remove item from anywhere in List**
- ***remove(int i)*** **removes item at index *i* and "pushes" remaining items left**

Output

[1, 3, 2]
3

ArrayList list

1	3	2

ArrayListDemo.java: ArrayLists can hold multiple objects, provide useful methods

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(1,3);
        System.out.println(list);
        Integer b = list.get(1);
        System.out.println(b);
        list.remove(1);
        System.out.println(list);
        list.set(1,4);
        System.out.println(list);
        System.out.println(list.size());
    }
}
```

- **Can remove item from anywhere in List**
- ***remove(int i)* removes item at index *i* and "pushes" remaining items left**

Output

```
[1, 3, 2]
3
[1, 2]
```

ArrayList list

1	2

ArrayListDemo.java: ArrayLists can hold multiple objects, provide useful methods

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(1,3);
        System.out.println(list);
        Integer b = list.get(1);
        System.out.println(b);
        list.remove(1);
        System.out.println(list);
        list.set(1,4);
        System.out.println(list);
        System.out.println(list.size());
    }
}
```

set(int i, E elmt)* sets the item at index *i* to *elmt

Overwrites value at index *i*

Output

```
[1, 3, 2]
3
[1, 2]
```

ArrayList list

1	2

ArrayListDemo.java: ArrayLists can hold multiple objects, provide useful methods

```java
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
  public static void main(String[] args) {
    List<Integer> list = new ArrayList<Integer>();
    list.add(1);
    list.add(2);
    list.add(1,3);
    System.out.println(list);
    Integer b = list.get(1);
    System.out.println(b);
    list.remove(1);
    System.out.println(list);
    list.set(1,4);
    System.out.println(list);
    System.out.println(list.size());
  }
}
```

set(int i, E elmt)* sets the item at index *i* to *elmt

Overwrites value at index *i*

Output

[1, 3, 2]
3
[1, 2]

ArrayList list

1	4

ArrayListDemo.java: ArrayLists can hold multiple objects, provide useful methods

```java
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(1,3);
        System.out.println(list);
        Integer b = list.get(1);
        System.out.println(b);
        list.remove(1);
        System.out.println(list);
        list.set(1,4);
        System.out.println(list);
        System.out.println(list.size());
    }
}
```

Output

[1, 3, 2]
3
[1, 2]
[1, 4]

ArrayList list

1	4

ArrayListDemo.java: ArrayLists can hold multiple objects, provide useful methods

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(1,3);
        System.out.println(list);
        Integer b = list.get(1);
        System.out.println(b);
        list.remove(1);
        System.out.println(list);
        list.set(1,4);
        System.out.println(list);
        System.out.println(list.size());
    }
}
```

- ***size()* returns the number of items stored in the *List***
- **What index does the last item have?**
- ***size()-1* due to zero-based indexing**

Output

[1, 3, 2]
3
[1, 2]
[1, 4]
2

ArrayList list

1	4

Lists can hold any kind of object, not just autoboxed versions of primitive data types

```
public class StudentTrackerAppList {
   public static void main(String[] args) {
      List<Student> students = new ArrayList<Student>();
      students.add(new Student("Alice", "f00xyz"));
      students.add(new GraduateStudent("Bob", "f00123"));
      students.add(new InternationalStudent("Charlie", "f00abc"));
```

- **List to hold multiple Student objects**
- **Add Student to List with *add* method**
- **Remember because a GraduateStudent is a Student, this List can also hold GraduateStudents and any other subclasses of Student**

Lists do not declare a maximum size unlike arrays

```
public class StudentTrackerAppList {
   public static void main(String[] args) {
      List<Student> students = new ArrayList<Student>();
      students.add(new Student("Alice", "f00xyz"));
      students.add(new GraduateStudent("Bob", "f00123"));
      students.add(new InternationalStudent("Charlie", "f00abc"));
```

```
public class StudentTrackerApp {
   public static void main(String[] args) {
      int numberOfStudents = 3;
      Student[] students = new Student[numberOfStudents];
      students[0] = new Student("Alice", "f00xyz");
      students[1] = new GraduateStudent("Bob", "f00123"));
      students[2] = new InternationalStudent("Charlie", "f00abc"));
```

- **Example from prior class that stored Student objects in an array**
- **Using arrays we had to declare the maximum number of Students the array could hold**
- **With a List there is no maximum number (as long as there is memory available)**

For-each loops are available for Lists, like they are with arrays

```java
public class StudentTrackerAppList {
  public static void main(String[] args) {
    List<Student> students = new ArrayList<Student>();
    students.add(new Student("Alice", "f00xyz"));
    students.add(new GraduateStudent("Bob", "f00123"));
    students.add(new InternationalStudent("Charlie", "f00abc"));


    //print students using for-each loop
    System.out.println("Before studying");
    for (Student student : students) {
      System.out.println(student);
    }
```

```java
public class StudentTrackerApp {
  public static void main(String[] args) {
    int numberOfStudents = 3;
    Student[] students = new Student[numberOfStudents];
    students[0] = new Student("Alice", "f00xyz");
    students[1] = new GraduateStudent("Bob", "f00123"));
    students[2] = new InternationalStudent("Charlie", "f00abc"));

    //print students using for-each loop
    System.out.println("Before studying");
    for (Student student : students) {
      System.out.println(student);
    }
```

For-each loop available for arrays and Lists

Use *size* to get the number of items in a List

```java
public class StudentTrackerAppList {
   public static void main(String[] args) {
      List<Student> students = new ArrayList<Student>();
      students.add(new Student("Alice", "f00xyz"));
      students.add(new GraduateStudent("Bob", "f00123"));
      students.add(new InternationalStudent("Charlie", "f00abc"));


      //print students using for-each loop
      System.out.println("Before studying");
      for (Student student : students) {
         System.out.println(student);
      }

      //randomly select students to study to simulate an actual application
      for (int i = 0; i < 10; i++) {
         //pick random student
         int index = (int) (Math.random() * students.size());

         //add random studying time between 0 and 5 hours
         double time = Math.random() * 5;
         students.get(index).study(time);
      }
```

```java
public class StudentTrackerApp {
   public static void main(String[] args) {
      int numberOfStudents = 3;
      Student[] students = new Student[numberOfStudents];
      students[0] = new Student("Alice", "f00xyz");
      students[1] = new GraduateStudent("Bob", "f00123"));
      students[2] = new InternationalStudent("Charlie", "f00abc"));

      //print students using for-each loop
      System.out.println("Before studying");
      for (Student student : students) {
         System.out.println(student);
      }

      //randomly select students to study to simulate an actual application
      for (int i = 0; i < 10; i++) {
         //pick random student
         int index = (int)(Math.random() * numberOfStudents);

         //add random studying time between 0 and 5 hours
         double time = Math.random() * 5;
         students[index].study(time);
      }
```

Use *size* to get number of items in List (vs. predefined number with array)

Note the cast between double and int

Also note where the parenthesis are! Don't cast Math.random or you'll always get 0!

Why?

Math.random gives number exclusive of 1, so casting drops decimal part

Use *get* to retrieve an item at a given index

```java
public class StudentTrackerAppList {
  public static void main(String[] args) {
    List<Student> students = new ArrayList<Student>();
    students.add(new Student("Alice", "f00xyz"));
    students.add(new GraduateStudent("Bob", "f00123"));
    students.add(new InternationalStudent("Charlie", "f00abc"));


    //print students using for-each loop
    System.out.println("Before studying");
    for (Student student : students) {
      System.out.println(student);
    }

    //randomly select students to study to simulate an actual application
    for (int i = 0; i < 10; i++) {
      //pick random student
      int index = (int) (Math.random() * students.size());

      //add random studying time between 0 and 5 hours
      double time = Math.random() * 5;
      students.get(index).study(time);
    }
```

```java
public class StudentTrackerApp {
  public static void main(String[] args) {
    int numberOfStudents = 3;
    Student[] students = new Student[numberOfStudents];
    students[0] = new Student("Alice", "f00xyz");
    students[1] = new GraduateStudent("Bob", "f00123"));
    students[2] = new InternationalStudent("Charlie", "f00abc"));

    //print students using for-each loop
    System.out.println("Before studying");
    for (Student student : students) {
      System.out.println(student);
    }

    //randomly select students to study to simulate an actual application
    for (int i = 0; i < 10; i++) {
      //pick random student
      int index = (int)(Math.random() * numberOfStudents);

      //add random studying time between 0 and 5 hours
      double time = Math.random() * 5;
      students[index].study(time);
    }
```

Use *get* to *get* an item in List (vs. square brackets with array)

C-style for loops are also available, use *get* with them

```java
public class StudentTrackerAppList {
  public static void main(String[] args) {
    List<Student> students = new ArrayList<Student>();
    students.add(new Student("Alice", "f00xyz"));
    students.add(new GraduateStudent("Bob", "f00123"));
    students.add(new InternationalStudent("Charlie", "f00abc"));


    //print students using for-each loop
    System.out.println("Before studying");
    for (Student student : students) {
      System.out.println(student);
    }

    //randomly select students to study to simulate an actual application
    for (int i = 0; i < 10; i++) {
      //pick random student
      int index = (int) (Math.random() * students.size());

      //add random studying time between 0 and 5 hours
      double time = Math.random() * 5;
      students.get(index).study(time);
    }

    //print students using C-style for loop
    System.out.println("After studying");
    for (int i = 0; i < students.size(); i++) {
      System.out.println(students.get(i));
    }
```

List use *get* to retrieve item at index (vs. square brackets with array)

```java
public class StudentTrackerApp {
  public static void main(String[] args) {
    int numberOfStudents = 3;
    Student[] students = new Student[numberOfStudents];
    students[0] = new Student("Alice", "f00xyz");
    students[1] = new GraduateStudent("Bob", "f00123"));
    students[2] = new InternationalStudent("Charlie", "f00abc"));

    //print students using for-each loop
    System.out.println("Before studying");
    for (Student student : students) {
      System.out.println(student);
    }

    //randomly select students to study to simulate an actual application
    for (int i = 0; i < 10; i++) {
      //pick random student
      int index = (int)(Math.random() * numberOfStudents);

      //add random studying time between 0 and 5 hours
      double time = Math.random() * 5;
      students[index].study(time);
    }

    //print students using C-style for loop
    System.out.println("After studying");
    for (int i = 0; i < students.size(); i++) {
      System.out.println(students[i]);
    }
```

C-style for loops are also available, use *get* with them

```
public class StudentTrackerAppList {
  public static void main(String[] args) {
    List<Student> students = new ArrayList<Student>();
    students.add(new Student("Alice", "f00xyz"));
    students.add(new GraduateStudent("Bob”, "f00123"));
    students.add(new InternationalStudent("Charlie", "f00abc"));


    //print students using for-each loop
    System.out.println("Before studying");
    for (Student student : students) {
      System.out.println(student);
    }

    //randomly select students to study to simulate an actual application
    for (int i = 0; i < 10; i++) {
      //pick random student
      int index = (int) (Math.random() * students.size());

      //add random studying time between 0 and 5 hours
      double time = Math.random() * 5;
      students.get(index).study(time);
    }

    //print students using C-style for loop
    System.out.println("After studying");
    for (int i = 0; i < students.size(); i++) {
      System.out.println(students.get(i));
    }
```

Output

```
Before studying
Name: Alice (f00xyz)
        Graduation year: null
        Hours studying: 0.0
        Hours in class: 0.0
Name: Bob (f00123)
        Graduation year: null
        Hours studying: 0.0
        Hours in class: 0.0
        Hours in the lab: 0.0
        Department: null
        Advisor: null
Name: Charlie (f00abc)
        Graduation year: null
        Hours studying: 0.0
        Hours in class: 0.0
        Home country: null
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Alice. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Charlie. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Alice. I'm studying!
Hi Mom! It's Alice. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Alice. I'm studying!
After studying
Name: Alice (f00xyz)
        Graduation year: null
        Hours studying: 6.590768116487223
        Hours in class: 0.0
<snip>
```

Agenda

1. ADTs
2. Generics
3. Java provided List implementation
4. Run-time complexity
5. Asymptotic notation

Key points:

1. **We'd like a way to compare different approaches to solving the same problem in a principled manner**
2. **Considering the number of operations helps us do that**

How long does it take to find an item in a List?

Index	0	1	2	3	4		n-2	n-1
	Bob	Elvis	Abby	Denise	Charlie	…	Yancy	Zephyr

Assume there are *n* items in the List (index 0 … n-1)

Find index of “Paula” in List

What pseudo code would you use:

for *i = 0 … n-1*
    get item at index *i*
    if item is equal to search value
        return index *i*
return -1 (or otherwise indicate search term not in List)

How long to find the item? Should we time how long it takes?

Time would depend on

- Hardware
- Where Paula was located in the List

What is the best case?

What is the worst case?

What is the average case?

How long does it take to find an item in a List?

Index	0	1	2	3	4		n-2	n-1
	Bob	Elvis	Abby	Denise	Charlie	…	Yancy	Zephyr

Instead of timing execution we will ***count*** how many operations are needed in the ***worst case***

- Doesn't depend on hardware or software environment
- Could use average case, but average is hard to define sometimes because it would be based on the input's distribution
- Worst case tells us it won't take longer to execute
- Allows language-independent analysis based on number of elements

Operations to count

- Assign value to variable
- Following an object reference to heap memory
- Performing arithmetic operation (e.g., add two numbers)
- Compare two values (if statement)
- Access element in array
- Calling or returning from a method

Often run-time will depend on the number of elements an algorithm must process

Constant time – does not depend on number of items

- Returning the first element of a list takes a constant amount of time *irrespective* of the number of elements in the list
- Just return the first item
- No need to march down list to find the first element (*head*)
- Array *get()* implementation is also constant time (array *get()* is constant time everywhere, linked list only constant at *head*)

Linear time – directly depends on number of items

- Example: searching for a particular value stored in a list
- Start at first item, compare value with value trying to find
- Keep going until find item, or end up at end of list
- Could get lucky and find item right away, might not find it at all
- Worst case is we check all *n* items

Often run-time will depend on the number of elements an algorithm must process

Polynomial time – depends on a function of number of items

- Example: nested loop in image and graphic methods
- If changing all pixels in n by n image, must do a total of n^2 operations because inner and outer loops each run n times
- Normally runs slower than a constant or linear time algorithm

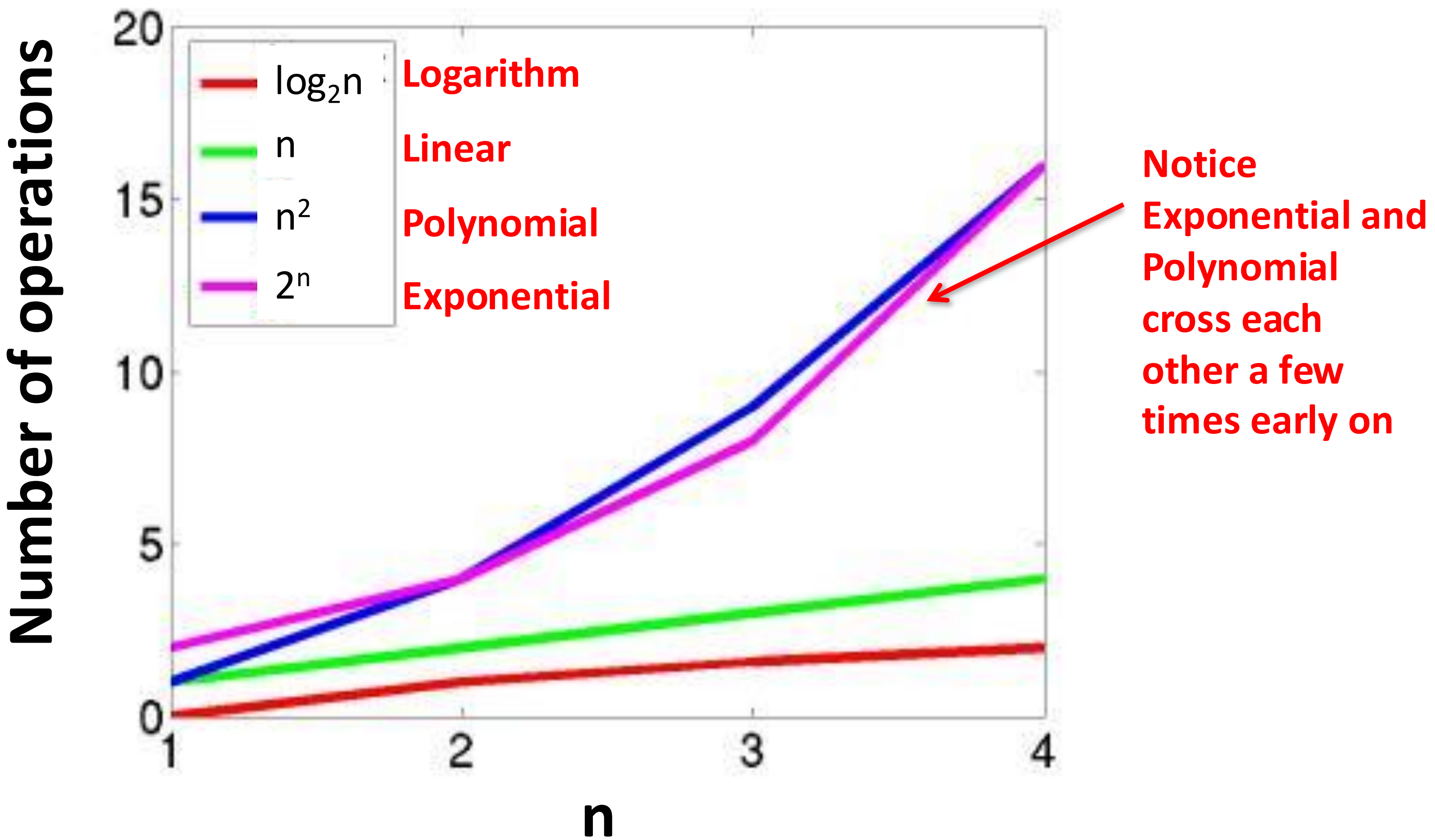
Logarithm time – avoids operations on some items

- Soon we will look at binary search
- Reduces the number of items algorithm must process (don't process all n items)
- Runs faster than linear or polynomial time (slower than constant)

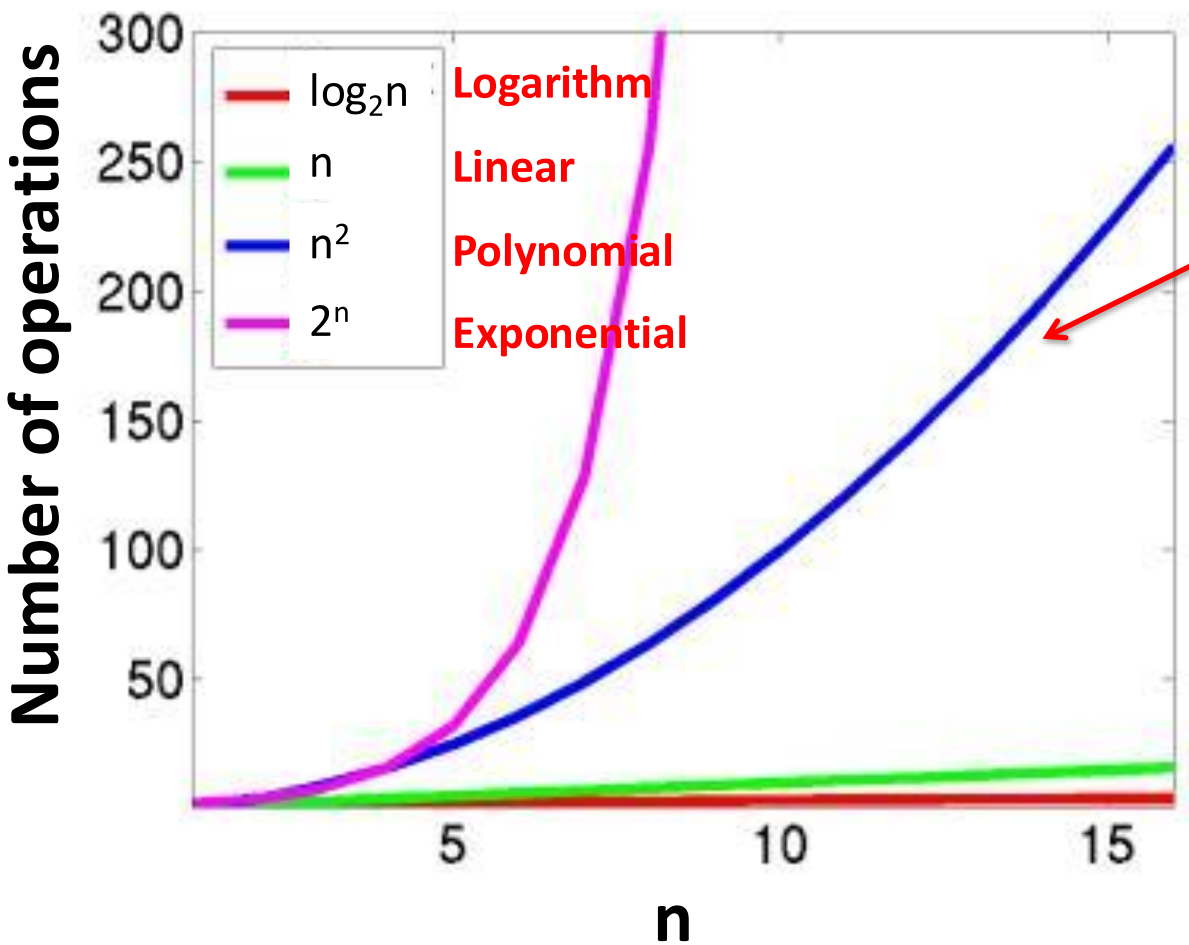
Exponential time – base raised to power

- Combination problems: all possible bit combinations in n bits = 2^n
- SLOW!

For small numbers of items, run time does not differ by much



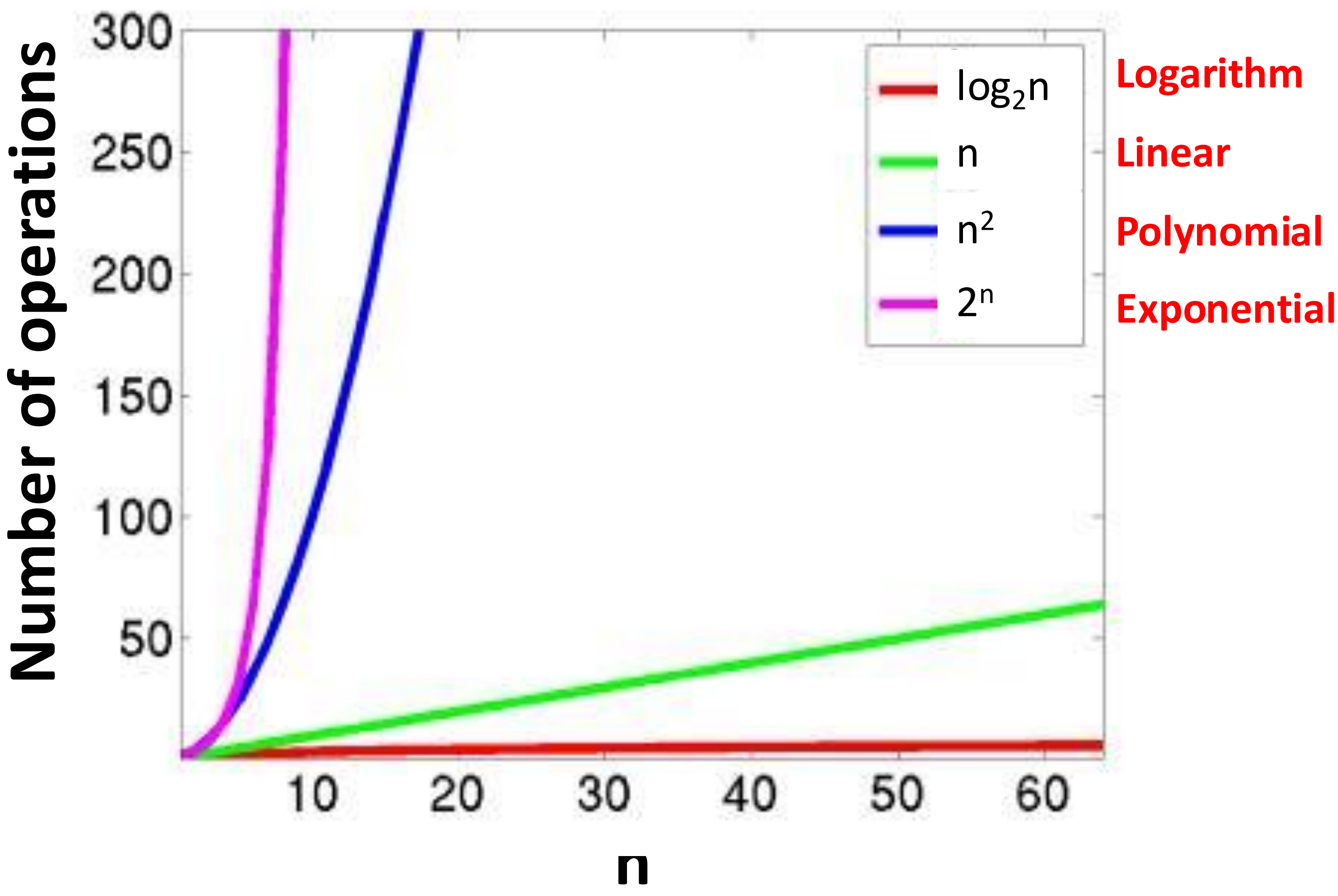
As *n* grows, number of operations between different algorithms begins to diverge



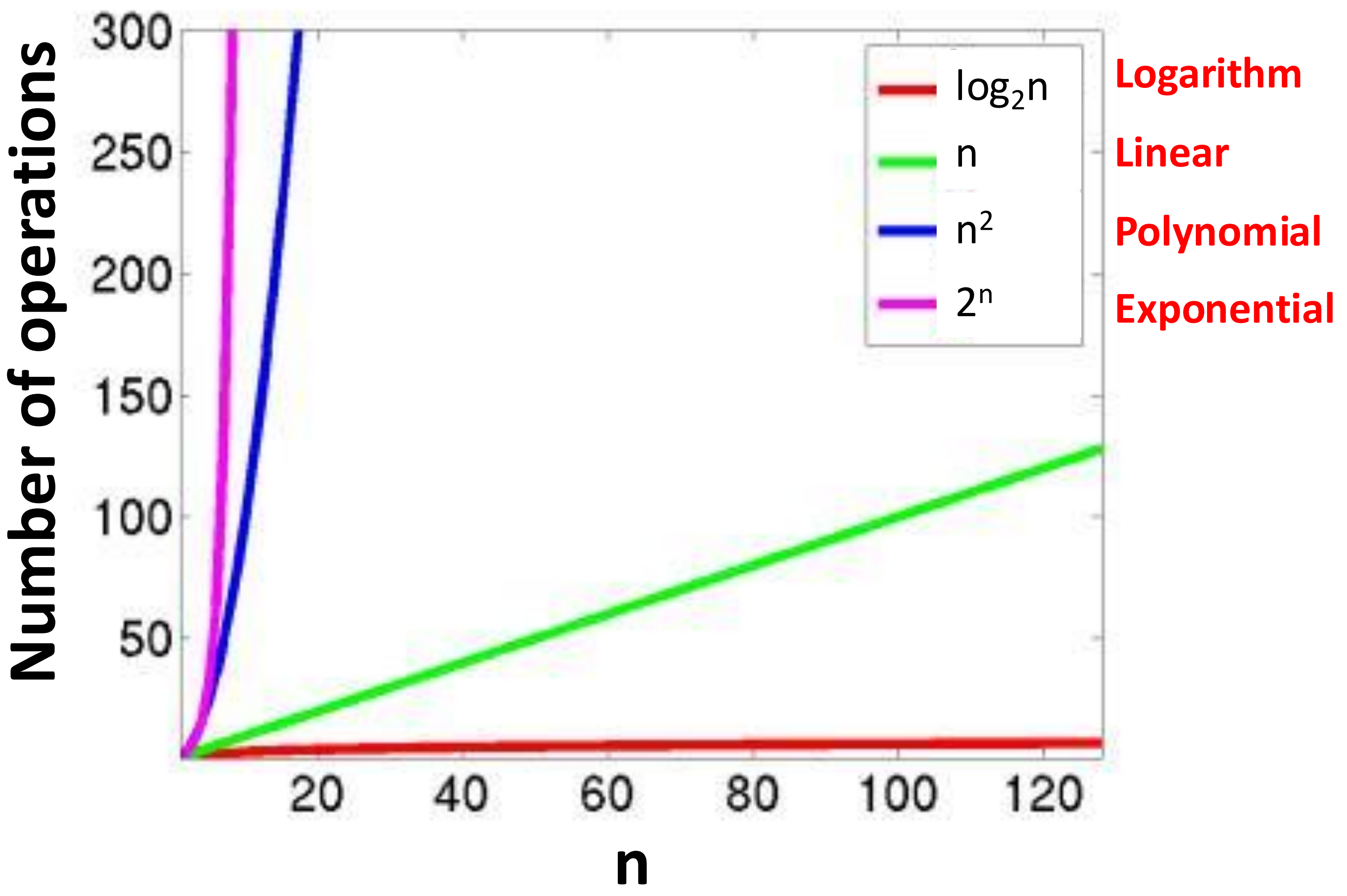
After n=4 Exponential is always greater than Polynomial

We will use that soon to define n_0 (standby for more info)

Even with only 60 items, there is a large difference in number of operations



Eventually, even with speedy computers, some algorithms become impractical



Sometimes complexity can hurt us, sometimes it can help us

Hurts us
Can't brute force chess
algorithm 2^n

Helps us
Can't crack password
algorithm 2^n

Images: thechessstore.com; studyoffice.org

Agenda

1. ADTs
2. Generics
3. Java provided List implementation
4. Run-time complexity
5. Asymptotic notation

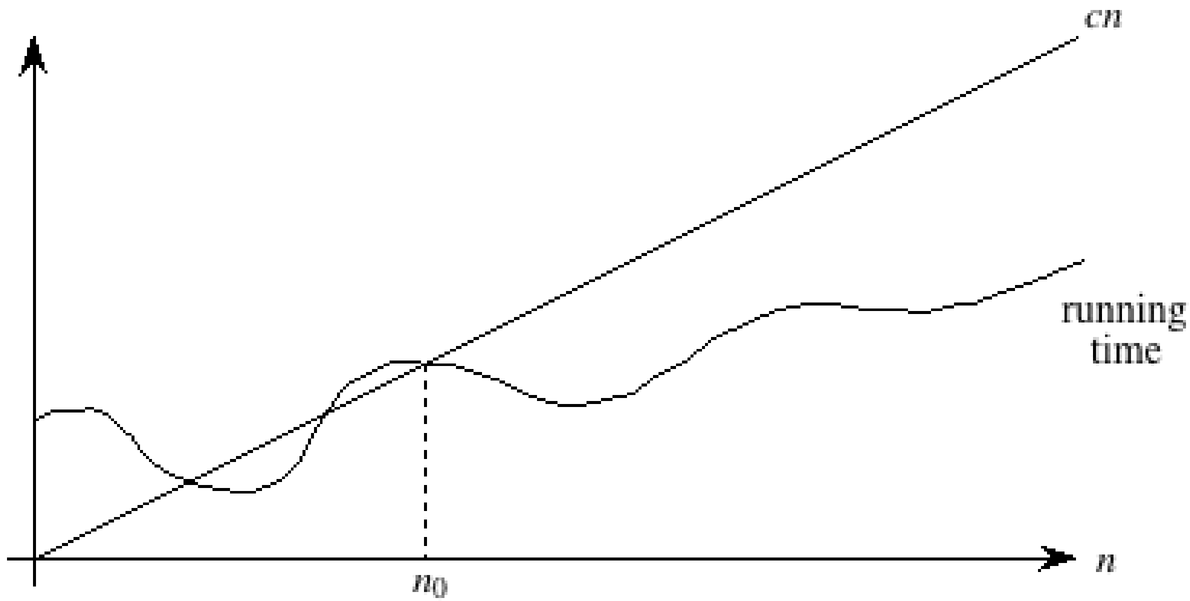
Key points:

1. **Big-Oh provides an upper bound on run-time complexity**
2. **Big-Omega provides a lower bound on run-time complexity**
3. **Big-Theta provides a tight bound on run-time complexity**

Computer scientists describe upper bounds on orders of growth with "Big Oh" notation

O gives an asymptotic upper bounds

"Big Oh of n", and "Oh of n", and "order n" all mean the same thing!



Example: find *specific* item in a list

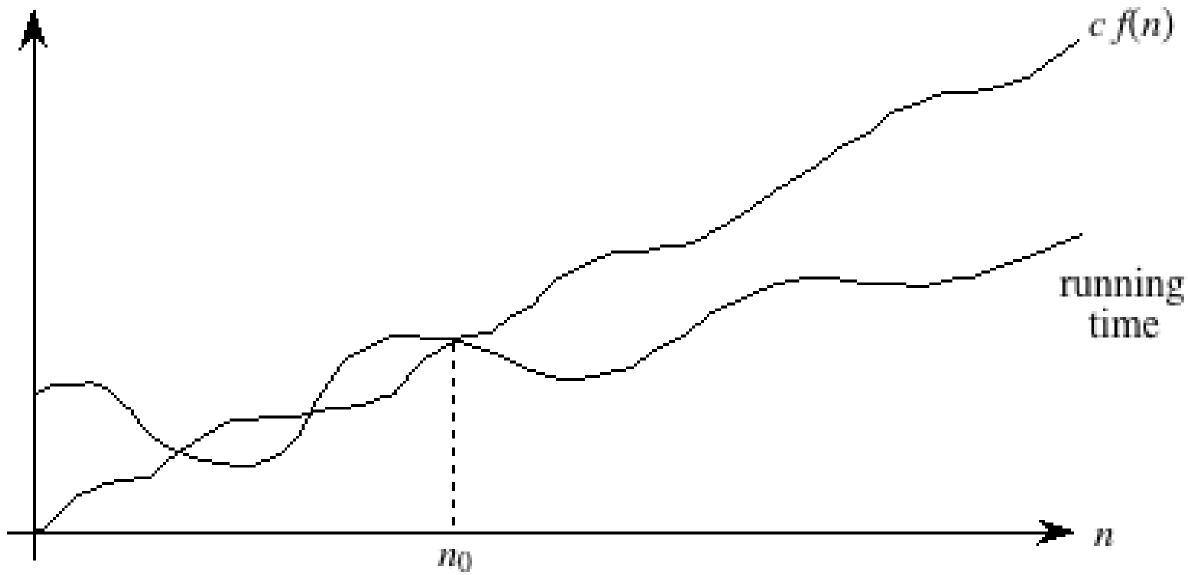
- **Might find item on first try**
- **Might not find it at all (must check all *n* items in list)**
- **Worst case (upper bound) is O(*n*)**

Run-time complexity is O(n) if there exists constants n_0 and c such that:

- $\forall n \geq n_0$
- run time of size n is at *most* cn, upper bound
- O(n) is the *worst* case performance for large n, but actual performance could be better
- O(n) is said to be "linear" time
- O(1) means constant time

We can extend Big Oh to any, not necessarily linear, function

O gives an asymptotic upper bounds

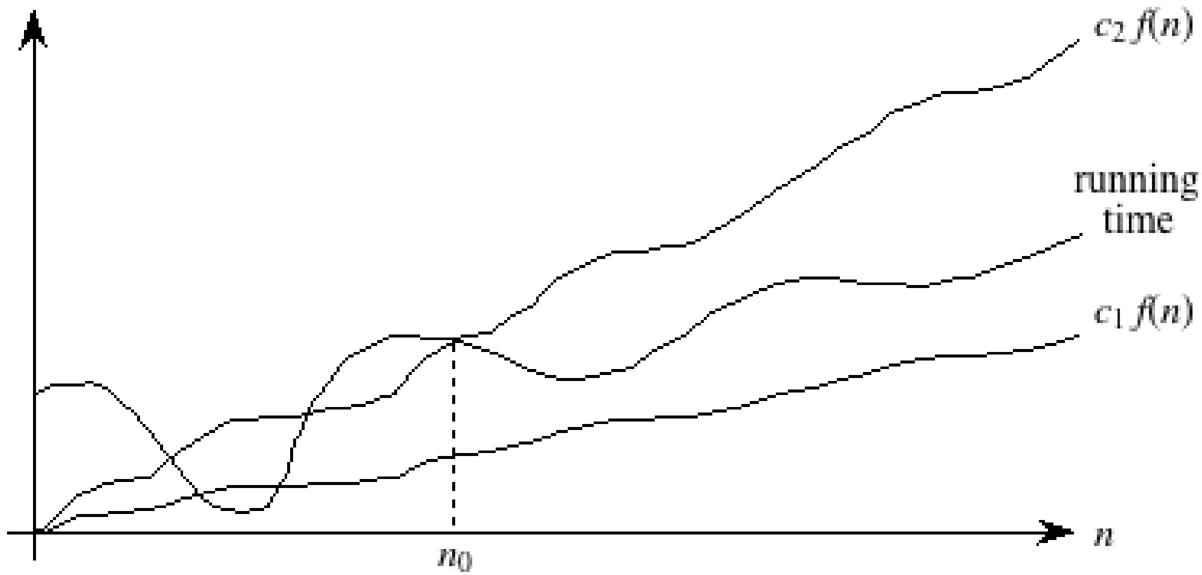


Run-time complexity is O(*f(n)*) if there exists constants n_0 and *c* such that:

- $\forall n \geq n_0$
- run time of size *n* is at ***most*** *cf(n)*, upper bound
- O(*f(n)*) is the ***worst*** case performance for large *n*, but actual performance could be better
- *f(n)* can be a non-linear function such as n^2 or *log(n)*
- In that case O(n^2) *or* O(*log n)*

Run time can also be Ω (Big Omega), where run time grows at least as fast

Ω gives an asymptotic <u>lower</u> bounds



Run-time complexity is $\Omega(f(n))$ if there exists constants n_0 and c_1 such that:

- $\forall n \geq n_0$
- run time of size *n* is at <u>*least*</u> $c_1 f(n)$, lower bound
- $\Omega(n)$ is the <u>*best*</u> case performance for large *n*, but actual performance can be worse

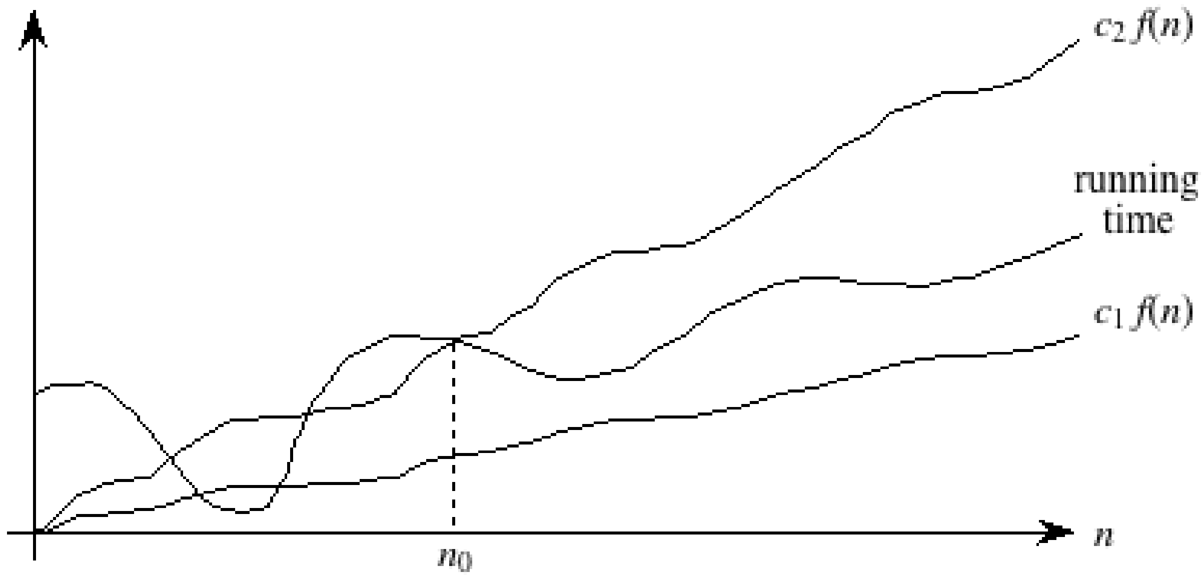
Example: find <u>*largest*</u> item in a list

- **Must check each *n* items**
- **Largest item could be at end of list, can't stop early**
- **Can't do better than Ω (*n*)**

We use Θ (Big Theta) for tight bounds when we can define O and Ω

Θ gives an asymptotic <u>tight</u> bounds

We can also apply these concepts to how much memory an algorithm uses (not just run-time complexity)



Example: find *<u>largest</u>* item in a list

- **Best case: already seen it is Ω(n)**
- **Worst case: must check each item, so O(n)**
- **Because Ω(n) *<u>and</u>* O(n) we say it is Θ(n)**

Run-time complexity is Θ(*f(n)*) if there exists constants n_0 and c_1 and c_2 such that:

- $\forall n \geq n_0$
- run time of size *n* is at <u>*least*</u> $c_1 f(n)$ and at <u>most</u> $c_2 f(n)$
- Θ(*n*) gives a tight bound, which means run time will be within a constant factor
- Generally we will use either O or Θ
- **O, Ω, Θ called asymptotic notation**

We ignore constants and low-order terms in asymptotic notation

Constants don't matter, just adjust c_1 and c_2

- Constant multiplicative factors are absorbed into c_1 (and c_2)
- Example: $1000n^2$ is $O(n^2)$ because we can choose c_1 to be 1000 (remember bounded by $c_1 n$)
- Do care in practice – if an operation takes a constant time, O(1), but more than 24 hours to complete, can't run it everyday

Low order terms don't matter either

- If $n^2+1000n$, then choose $c_1 = 1$, so now $n^2 +1000n \geq c_1 n^2$
- Now must find c_2 such that $n^2 +1000n \leq c_2 n^2$
- Subtract n^2 from both sides and get $1000n \leq c_2 n^2 - n^2 = (c_2-1)n^2$
- Divide both sides by $(c_2-1)n$ gives $1000/(c_2-1) \leq n$
- Pick $c_2 = 2$ and $n_0 = 1000$, then $\forall n \geq n_0$, $1000 \leq n$
- So, $n^2 +1000n \leq c_2 n^2$, try with $n=1000$ get $n^2 + 1000^2 = 2*n^2$
- **In practice, we simply ignore constants and low order terms**

Pierson's field guide to spotting run-time complexity

Constant time	Linear time	Polynomial time
$O(1)$	$O(n)$	$O(n^2)$

Key points

1. ADTs say *what* needs to be done, but not *how* do implement it
2. An *interface* describes methods must be implemented for an ADT
3. Generics allow us to write an ADT one time, irrespective of the data types involved
4. Java provides two implementations of the List ADT, an ArrayList and a LinkedList
5. Each implementation provides the same ADT operations, but work differently
6. We will soon implement both ourselves to see how they work
7. We'd like a way to compare different approaches to solving the same problem in a principled manner
8. Considering the number of operations helps us do that
9. Big-Oh provides an upper bound on run-time complexity
10. Big-Omega provides a lower bound on run-time complexity
11. Big-Theta provides a tight bound on run-time complexity