Northeastern University

CS 5010: Programming Design Paradigms

Fall 2022

Lecture 7: Data Structures and Algorithms

Acknowledgement: lecture notes inspired by course material prepared by UW faculty members Z. Fung and H. Perkins.
In addition, this slide deck was prepared by Tamara Bonaci, faculty member at Northeastern University.

Brian Cross
b.cross@northeastern.edu

Administrivia

- **Lab 4** due Friday @ 11:59pm
- **Codewalk 3** questions out Monday, **October 24th**
  - Due **Tuesday, October 25th @ 11:59pm**
  - Show UML ~2 minutes, then walk through code.
- **No Lab Monday**
  - Use time to create and publish your codewalk

Administrivia

- **Assignment 3** due Monday @ 11:59pm
  - To be done individually
  - Helping your peers in general concepts 👍
  - Copying code 👎
  - Need help? Please ask and ask early!
    - Office Hours
    - Piazza
- Integrity
- Tips

Administrivia

- Homeworks #4 - #6 coming up
- Form Teams of 2 for group assignments (#4 - #6).
  - Fill out survey with group information before **next class**
  - Partners both in evening section
  - New group repos created after next class

Upcoming Grades

- Homework 2
  - ETA: By end of week
- Homework 1 Refactor
  - ETA: By next week
- HW #3 Predesign
  - Feedback has been sent
  - Consult the UML Predesign Instructions
    - Ensure you have ONLY the UML in your PR
    - Ensure you have a meaningful title of your PR

Agenda – Algorithms and Data Structures 1

- Working as a Team
- Quick Review
- Streams
- Data collections
  - Iterating over data collections
    - Interface Iterable
    - Iterator
  - Ordering of objects
    - Interface Comparable
    - Interface Comparator
- List ADT
  - Doubly-linked List
  - Inner and nested classes
  - Algorithm: Recursion
  - Recursive data collections

Working as a team

Learn about your partner

Do they have schedule restrictions?

What are their interests in regard to engineering and problem solving?

Use each other’s strengths

Help with each other’s developing skills

Define the parts of the spec

Break the problem down

- What are the different components?
- How do they relate?
- UML can help

How will your components interact?

- Identify the behaviors
- What interfaces will you define?

Tests will help you

Write tests to help validate contracts

Use tests to help with ambiguity

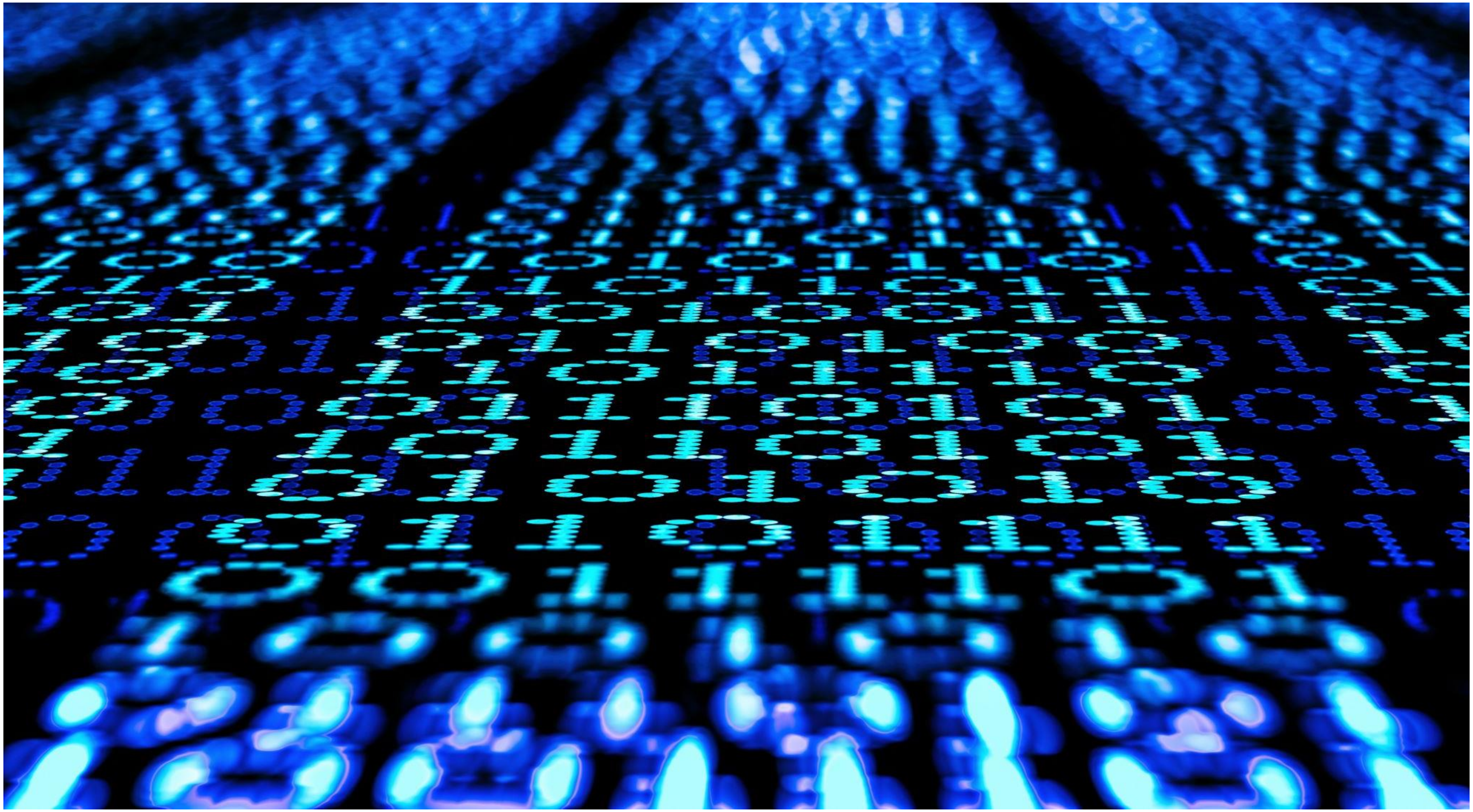
Write them early so that you can use them

You have to write them anyway, get as much use as possible out of them.

Integration

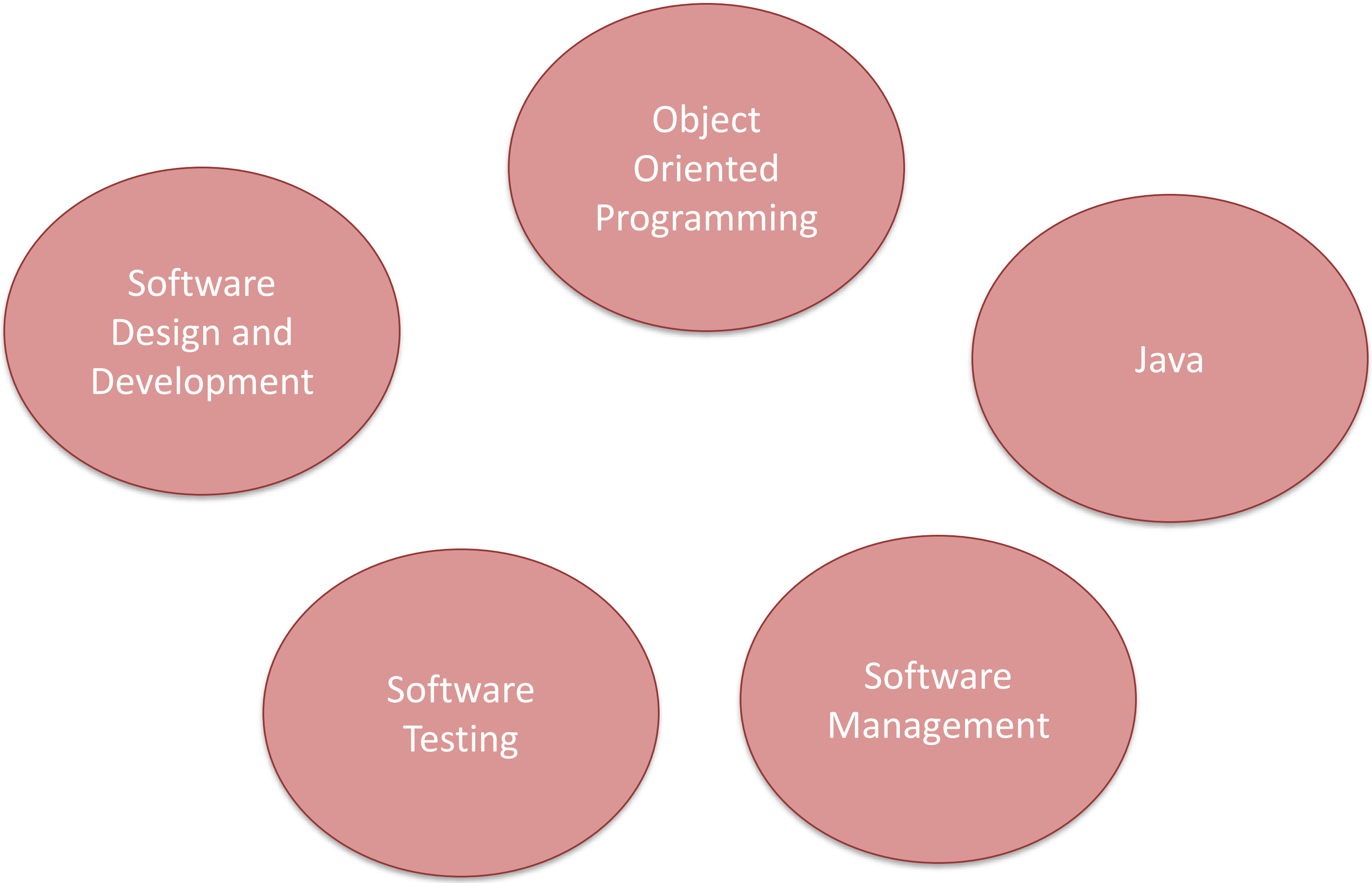
- Don’t wait until the last minute for integrating your components
  - You can integrate multiple times.
- Integration time is when miscommunications are found out.
- Utilize tests!
  - Having good tests can help prepare for integration
  - Will help ensure things stay working.

Working together in git



A Quick Review

Course so Far…



Object Oriented Programming

Software Design and Development

Java

Software Testing

Software Management

Objects and Classes

- **Object** – an entity consisting of states and behavior
  - States stored in variables/fields
  - Behavior represented through methods

- **Class** – template/blueprint describing the states and the behavior that an object of that type supports

Review

INTERFACE

ABSTRACT CLASS

CONCRETE CLASS

Abstract Class vs. Abstract Data Type?

Abstract class

- contains one or more abstract methods

Abstract Data Type

- a high-level model of a data type, where the data type is defined from a point of view of the user (focus on operations (behavior), not on implementation)

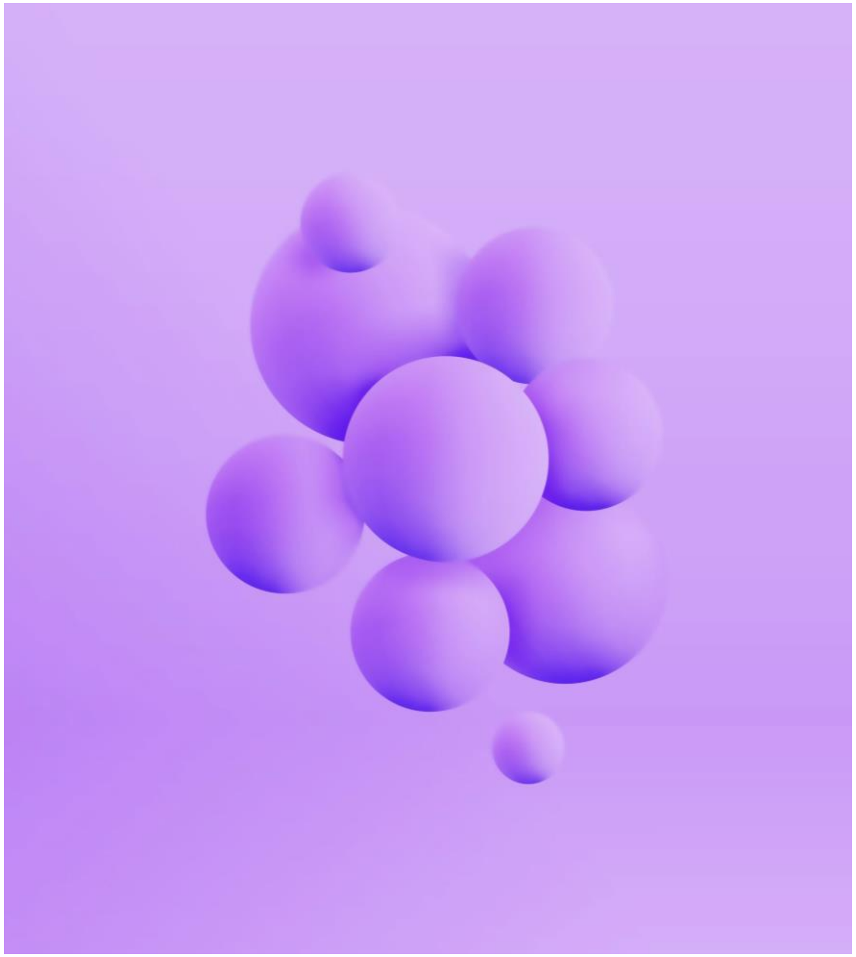
Review: Abstract Data Type

- Abstract Data Type (ADT) - model that describes data by specifying the operations that we can perform on them
- Clients care about the ADT → we need to capture the client's expectations in terms of the operations on the ADT
- For each operation, we need to describe:
  - The expected inputs, and any conditions that need to hold for our inputs and/or our ADT
  - The expected outputs and any conditions that need to hold for our output and/or our ADT
  - Invariants about our ADT

Object Oriented Design Principles

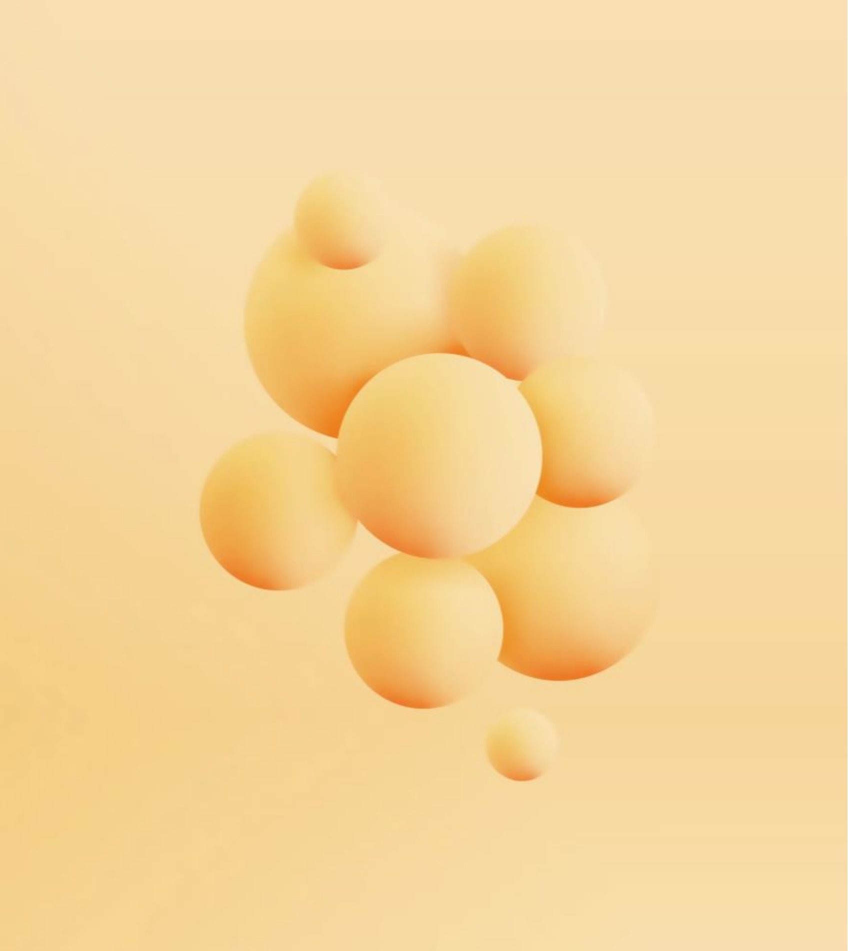
- Encapsulation
- Abstraction
- Information hiding
- Polymorphism
- Inheritance

Review: Polymorphism



The ability of one instance to be viewed/used as different types (the ability to take many shapes/forms/views)

Three Types of Polymorphism



1. Subtype polymorphism
2. Ad hoc polymorphism
3. Parametric polymorphism

Review: Subtype Polymorphism

- Subtypes are substitutable for supertypes
  - Instance of subtypes won’t surprise a client by failing to satisfy the supertype’s specification
  - Instance of subtype won’t surprise a client by having more expectations than the supertypes’ specifications

The subtype IS A supertype and thus should act like it.

Review: Compile Time and Run Time Types

```
Person emily = new Person();
Singer adele = new Singer();
Person flora = new Singer();
```

- Static (compile time) type – the declared type of a reference variable. Used by a compiler to check syntax

- Dynamic (run time) type – the type of an object that the reference variable currently refers to (it can change as the program execution progresses)

Review: Ad Hoc Polymorphism

- Overloading allows us to create methods that share the same method name but differ in their signature

- Ad hoc polymorphism – another name for function and operator overloading

- Ad hoc polymorphism – a type of polymorphism where a polymorphic functions can be applied to arguments of different types
  - Polymorphic (overloaded) function can denote a number of distinct and potentially heterogeneous implementations, depending on the type of argument(s) to which it is applied

Parametric Polymorphism

- Parametric polymorphism - ability for a function or type to be written in such a way that it handles values identically without depending on knowledge of their types
  - Such a function or type is called a generic function or data type

- Motivation - parametric polymorphism allows us to write flexible, general code without sacrificing type safety
  - Most commonly used in Java with collections

Algorithms and Data Structures 1

ASIDE: BASIC JAVA I/O

I/O Streams

- Concept of an I/O stream – a communication channel ("pipe") between a source and a destination that allows us to create a flow of data

- I/O Stream has:
  - An input source (a file, another program, device)
  - An output destination (file, another program, device)
  - The kind of data streamed (bytes, ints, objects)

How to Do I/O

```
import java.io.*;
```

- Open the stream
- Use the stream (read, write, or both)
- Close the stream

Opening a Stream

- Problem:
  - There exists some external data that we want to get, or
  - We want to put data somewhere outside your program
- Solution: open a steam
  - When we open a stream, we are making a connection to that external place
  - Once the connection is made, we can forget about the external place, and just use the stream

open
use
close

Example of Opening a Stream

- `FileReader` - used to connect to a file that will be used for input:

```
FileReader fileReader = new FileReader(fileName);
```

  - Filename - specifies where to find the (external) file (Note: after instantiating `FileReader` object, we never use `fileName` again; instead, we use `fileReader` object

open
use
close

Using a Stream

- Using a stream means doing input from it, or output to it
- Some streams can be used only for input, others only for output, still others for both
- But it is not usually that simple - we need to manipulate the data in some way as it comes in, or goes out

open
use
close

Example of Using a Stream

```
int ch;
ch = fileReader.read( );
```

- The `fileReader.read()` method reads one character, and returns it as an integer, or -1 if there are no more characters to read
- The meaning of the integer depends on the file encoding (ASCII, Unicode, other)

Manipulating the Input Data

- Reading characters as integers is not usually what we want to do
- A `BufferedReader` will convert integers to characters; it can also read whole lines
- The constructor for `BufferedReader` takes a `FileReader` parameter:

```
BufferedReader bufferedReader = new
BufferedReader(fileReader);
```

open
use
close

Reading Lines

```
String s;
s = bufferedReader.readLine( );
```

- A `BufferedReader` will return null if there is nothing more to read

BufferedReader

- Reads text from a character-input stream, buffering characters to provide efficient reading of characters, arrays, and lines
- Buffer size may be specified, or the default size may be used (the default is large enough for most purposes)

- Why use BufferReader?
  - In general, each read request causes a corresponding read request to be made of the underlying character or byte stream
  - These operations may be costly → each invocation of `read()` or `readLine()` in a `FileReader` or `InputFileReader` causes bytes to be read from the file, converted into characters, and then returned
  - `BufferReader` increases efficiency by buffering the input from the specified file

open
use
close

Closing a Stream

- A stream is an expensive resource!
- There can be a limit on the number of streams that you can have open at one time
- You should not have more than one stream open on the same file
- You should close a stream before you can open it again
- Always close your streams!

Algorithms and Data Structures 1

DATA COLLECTIONS

Data Collections?

Collection of chewed gums

Collection of pens

Collection of cassette tapes

Collection of old radios

What is a data collection?

Shoes collection

Star wars collection

Cars collection

[Pictures credit: http://www.smosh.com/smosh-pit/articles/19-epic-collections-strange-things]

Data Collections?

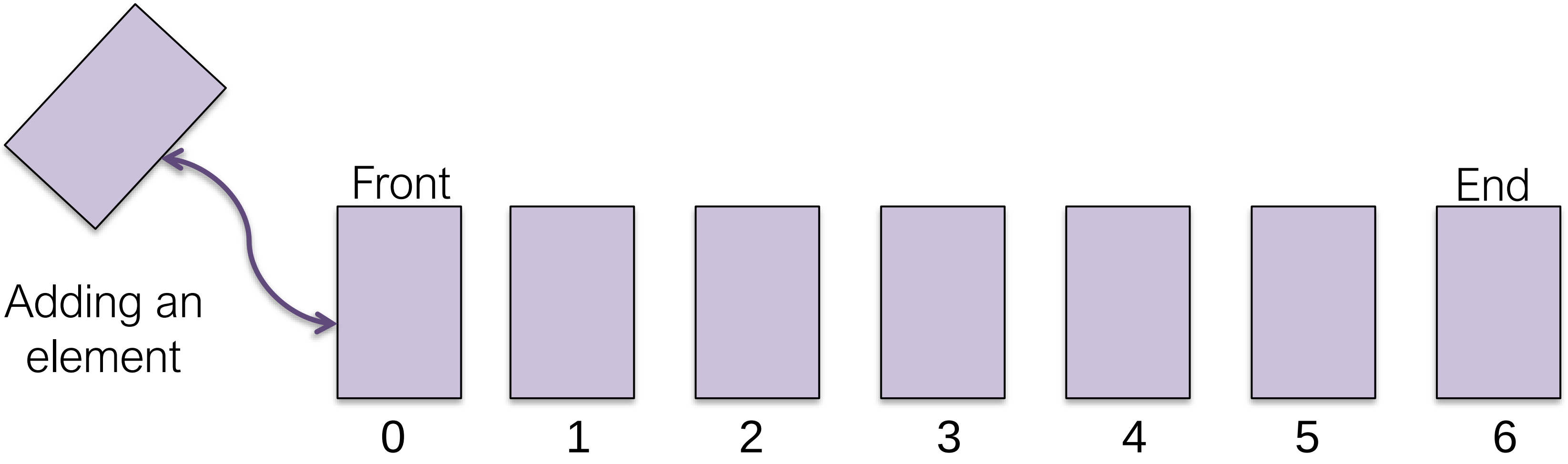
- Data collection - an object used to store data (think *data structures*)
  - Stored objects called elements
  - Some typical operations:
    - `add()`
    - `remove()`
    - `clear()`
    - `size()`
    - `contains()`
- Some examples: ArrayList, LinkedList, Stack, Queue, Maps, Sets, Trees

Why do we need different data collections?

Example: ArrayList vs. LinkedList

- **List** - a collection of elements with 0-based **indexes**
  - Elements can be added to the front, back, or in the middle
  - Can be implemented as an ArrayList or as a LinkedList

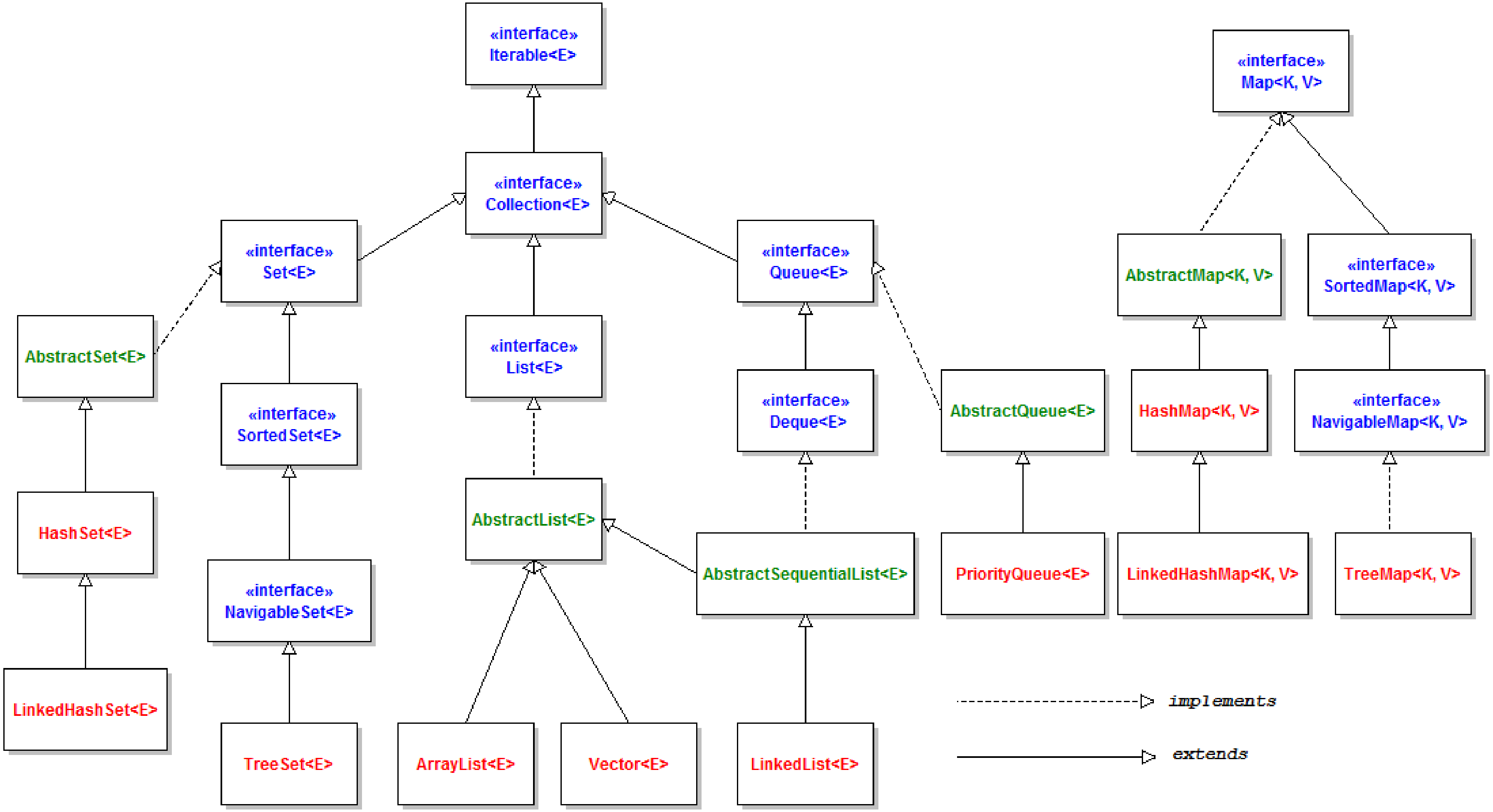
  - What is the complexity of adding an element to the front of an:
    - ArrayList?
    - LinkedList?



Algorithms and Data Structures 1

JAVA COLLECTIONS FRAMEWORK

Java Collections API



[Pictures credit: http://www.codejava.net]

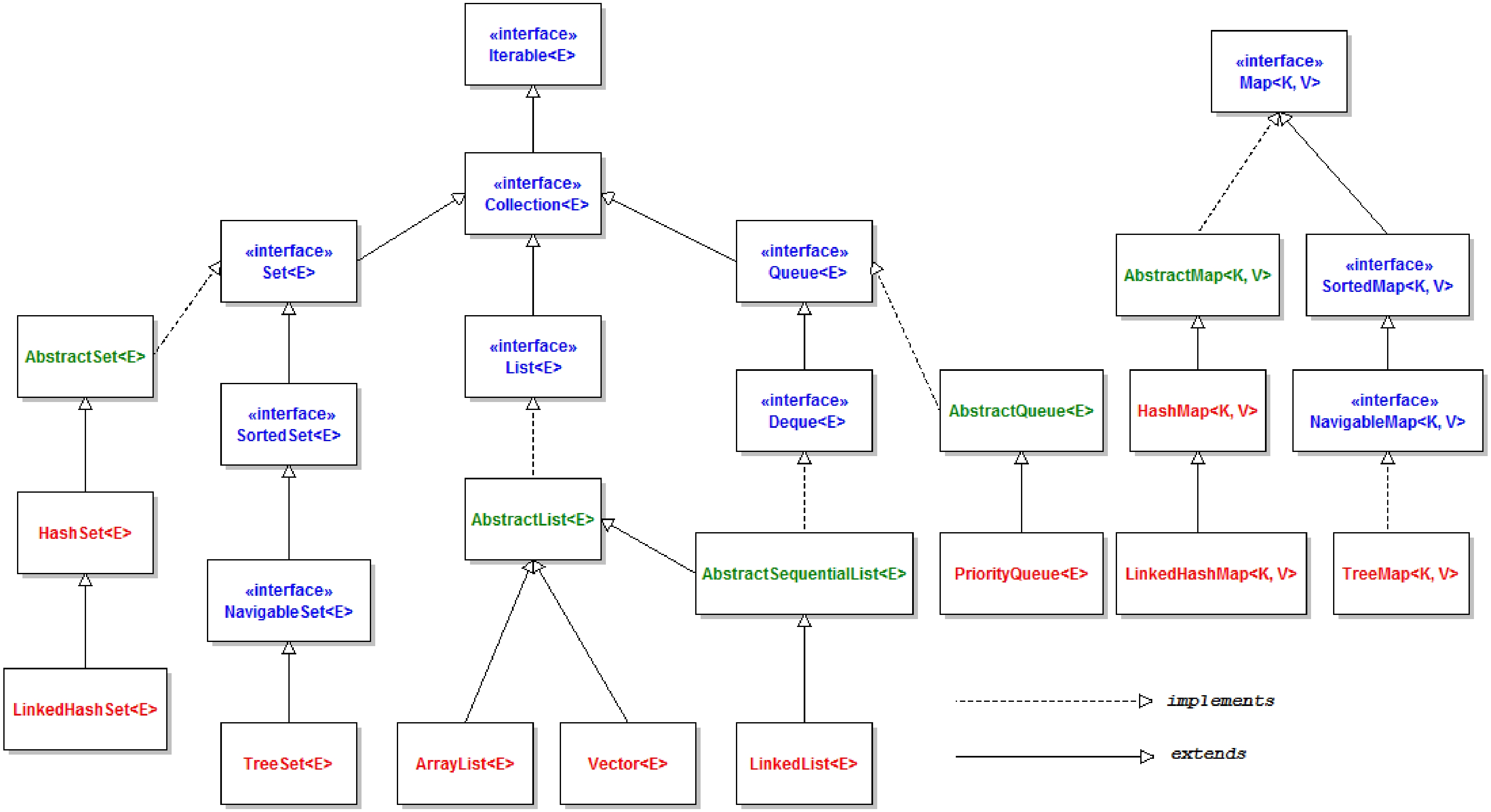
Java Collections Framework

- Under the Java Collections Framework, we have:
  - Interfaces that define the behavior of various data collections
  - Abstract classes that implement the interface(s) of the collection framework, that can then be extended to created a specialized data collection
  - Concrete classes that provide a general-purpose implementation of the interface(s)

Java Collections Framework

- Goals of the Java Collections Framework:
  1. Reduce programming effort by providing the most common data structures
  2. Provide a set of types that are easy to use and extend
  3. Provide flexibility through defining a standard set of interfaces for collections to implement
  4. Improve program quality through the use and reuse of tested implementations of common data structures

Java Collections API



[Pictures credit: http://www.codejava.net]

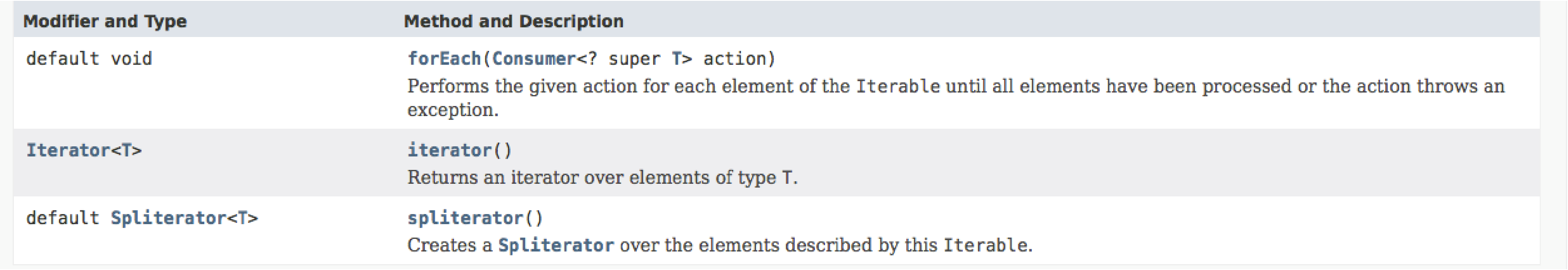
Java Collections Framework

- Part of the `java.util` package

- Interface `Collection<E>:`
  - Root interface in the collection hierarchy
  - Extended by four interfaces:
    - `List<E>`
    - `Set<E>`
    - `Queue<E>`
    - `Map<K,V>`
  - Extends interface `Iterable<T>`

Interfaces Iterable<T>

- Super-interface for interface `Collection<T>`
- Implementing interface `Iterable<T>` allows an object to be traversed using the for each loop
- Every object that implements `Iterable<T>` must provide a method `Iterator iterator()`



Modifier and Type	Method and Description
`default void`	`forEach(Consumer<? super T> action)` Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception.
`Iterator<T>`	`iterator()` Returns an iterator over elements of type `T`.
`default Spliterator<T>`	`spliterator()` Creates a `Spliterator` over the elements described by this `Iterable`.

`Iterable<T>` Interface and forEach Loop

- Super-interface for interface `Collection<T>`
- Implementing interface `Iterable<T>` allows an object to be traversed using the for each loop
- Every object that implements `Iterable<T>` must provide a method `Iterator iterator()`

- `ForEach loop:`

  default void `forEach(Consumer<? super T> action)`

- Performs the action for each element of the interface `Iterable`, until:
  - All elements have been processed or
  - The action throws an exception
- Actions are performed in the order of iteration (unless otherwise specified by the implementing class)
- Exceptions thrown by the action are relayed to the caller

Interface: Iterator<E>

- Interface Iterator – an iterator over a collection

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove():
}
```

- Iterator remove() method – removes the last item returned by method next()

- Careful when an iterator is used directly (not via a for each loop)→ if you make any structural changes to a collection being iterated (add, remove, clear), the iterator is no longer valid (ConcurrentModificationException thrown)

Algorithms and Data Structures 1

NATURAL ORDERING OF OBJECTS

Comparing Objects

- Problem: How do we compare `String` in some list of `Strings`?
  - Operators like < and > do not work with `String` objects
  - But we do think of `Strings` as having an alphabetical ordering
- Natural ordering: Rules governing the relative placement of all values of a given type
- Comparison function: Code that, when given two values A and B of a given type, decides their relative ordering:
  - A < B,   A == B,   A > B

The `compareTo` method

- A standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method
  - Example: in the `String` class, there is a method:

```
public int compareTo(String other)
```

- A call of A.`compareTo(B)` will return:

a value < 0 if A comes "before" B in the ordering,

a value > 0 if A comes "after" B in the ordering,

0 if A and B are considered "equal" in the ordering.

Using compareTo

- `compareTo` can be used as a test in an if statement

```
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) {  // true
    ...
}
```

Primitives	Objects
`if (a < b) { ...`	`if (a.compareTo(b) < 0) { ...`
`if (a <= b) { ...`	`if (a.compareTo(b) <= 0) { ...`
`if (a == b) { ...`	`if (a.compareTo(b) == 0) { ...`
`if (a != b) { ...`	`if (a.compareTo(b) != 0) { ...`
`if (a >= b) { ...`	`if (a.compareTo(b) >= 0) { ...`
`if (a > b) { ...`	`if (a.compareTo(b) > 0) { ...`

`compareTo()` and Java Collections

- We can use an array or list of `Strings` with Java's included binary search method because it calls `compareTo` internally

```
String[] a = {"al", "bob", "cari", "dan", "mike"};
int index = Arrays.binarySearch(a, "dan");  // 3
```

- Java's `TreeSet/Map` use compareTo internally for ordering

```
Set<String> set = new TreeSet<String>();
for (String s : a) {
    set.add(s);
}
System.out.println(s);
// [al, bob, cari, dan, mike]
```

Ordering Our Own Types

- Problem: we cannot binary search or make a `TreeSet/Map` of arbitrary types, because Java doesn't know how to order the elements
- Example: the program compiles but crashes when we run it

```
Set<HtmlTag> tags = new TreeSet<HtmlTag>();
tags.add(new HtmlTag("body", true));
tags.add(new HtmlTag("b", false));
...
Exception in thread "main"
  java.lang.ClassCastException
          at
  java.util.TreeSet.add(TreeSet.java:238)
```

Comparable Template

```
public class name implements Comparable<name> {

    ...

    public int compareTo(name other) {
        ...
    }
}
```

Comparable Example

```
public class Point implements Comparable<Point> {
    private int x;
    private int y;
    ...


    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;   // same x, smaller y
        } else if (y > other.y) {
            return 1;    // same x, larger y
        } else {
            return 0;    // same x and same y
        }
    }
}
```

Interface Comparator

- Problem: we may want to be able to order instance of some classes by more then one property (for example, by first name and by last name). What can we do?
- Solution: use interface `Comparator<T>` - a comparison function, which imposes a total ordering on some collection of objects
    - Can be passed to a sort method, to allow precise control over the sort order
    - Can also be used to control the order of certain data structures (tree sets and tree maps)
    - Can also be used provide an ordering for collections of objects that don't have a natural ordering

Method `compare(T o1, T o2)`

- `int compare(T o1, T o2)` - compares its two arguments for order and returns:
  - A negative integer if the first argument is less than the second
  - Zero, if the first argument is equal to the second
  - A positive integer if the first argument is greater than the second
- The implementor must also ensure that the relation is transitive:

  ```
  ((compare(x, y) > 0) && (compare(y, z) >
  0)) implies compare(x, z) > 0
  ```

- Not strictly required, but good rule to follows: `compare()` method should be consistent with `equals()` method:

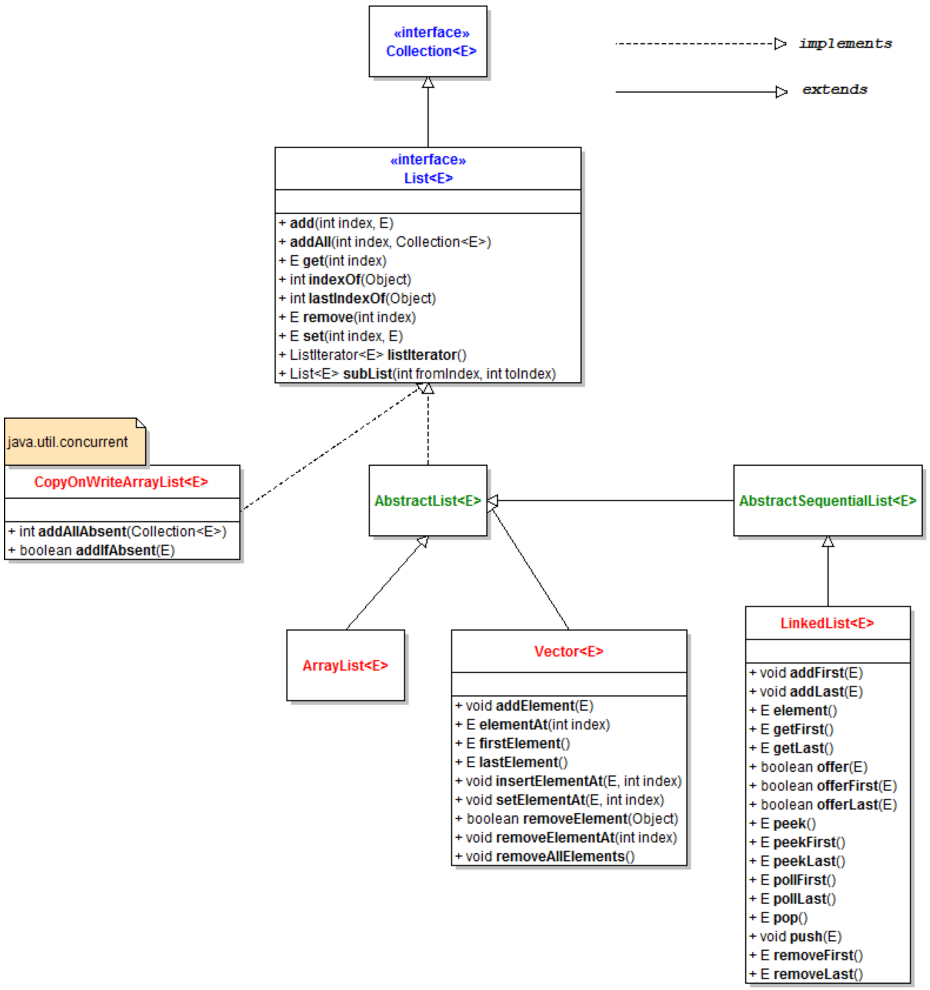
  ```
  (compare(x, y)==0) == (x.equals(y))
  ```

  (Any comparator that violates this condition should clearly indicate this fact. The recommended language is "`Note: this comparator imposes orderings that are inconsistent with equals.`") (don't do this!)

Algorithms and Data Structures 1

LIST ADT

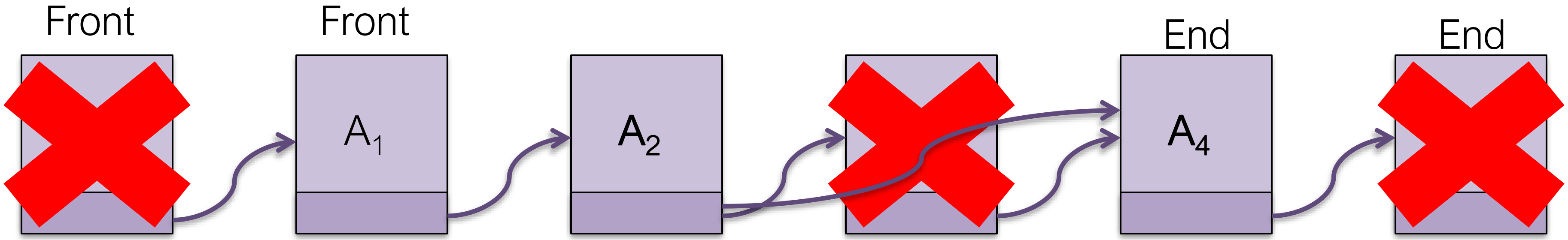
Java List API



[Pictures credit: http://www.codejava.net]

- `List<E>` - the base interface
- Abstract subclasses:
  - `AbstractList<E>`
  - `AbstractSequentialList<E>`
- Concrete classes:
  - `ArrayList<E>`
  - `LinkedList<E>`
  - `Vector<E>` (legacy collection)
  - `CopyOnWriteArrayList<E>` (class under java.util.concurrent package)
- Main methods:
  - `E get(int index);`
  - `E set(int index, E newValue);`
  - `Void add(int index, E x);`
  - `Void remove(int index);`
  - `ListIterator<E> listIterator();`

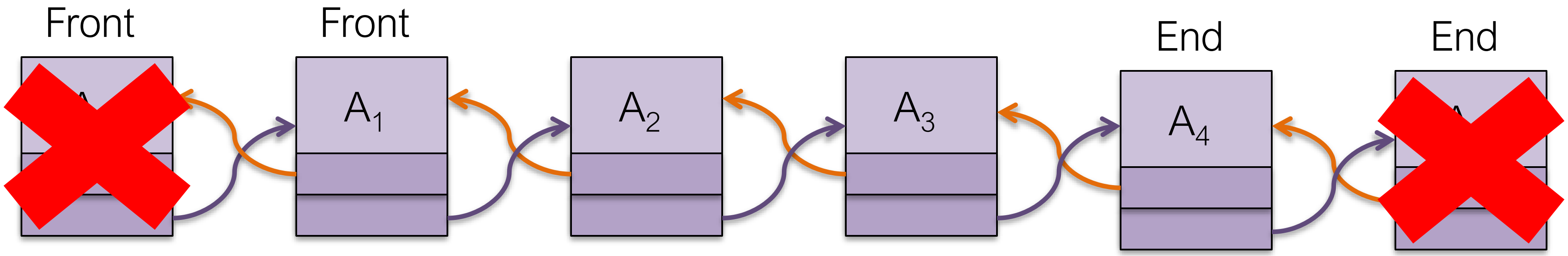
Example: Removing Elements from a LinkedList



- Remove the first element of the list
- Remove element A_3 from the list
- Remove the last element in the list

- What's the tricky part about removing elements A_3 and A_5?
- Having to find their predecessors (elements A_2 and A_4) and updating their link to last node
- Idea: every node maintains the link to its previous and its next node → doubly linked list

Doubly LinkedList



- Removing the first element of the list
- Removing the last element of the list

Algorithms and Data Structures

NESTED CLASES

Nested Classes

- Nested class – a class defined within some other class

- Some nested class features:
  - The scope of a nested class is bounded by the scope of its outer class (i.e., the inner class does not exist independently of its outer class)
    - Class example: node and tree
  - A nested class is a member of its outer class
    - Can be declared private, public, protected
    - Can have access to all members (including private) of its outer class
    - Outer class does not have access to the members of the nested class

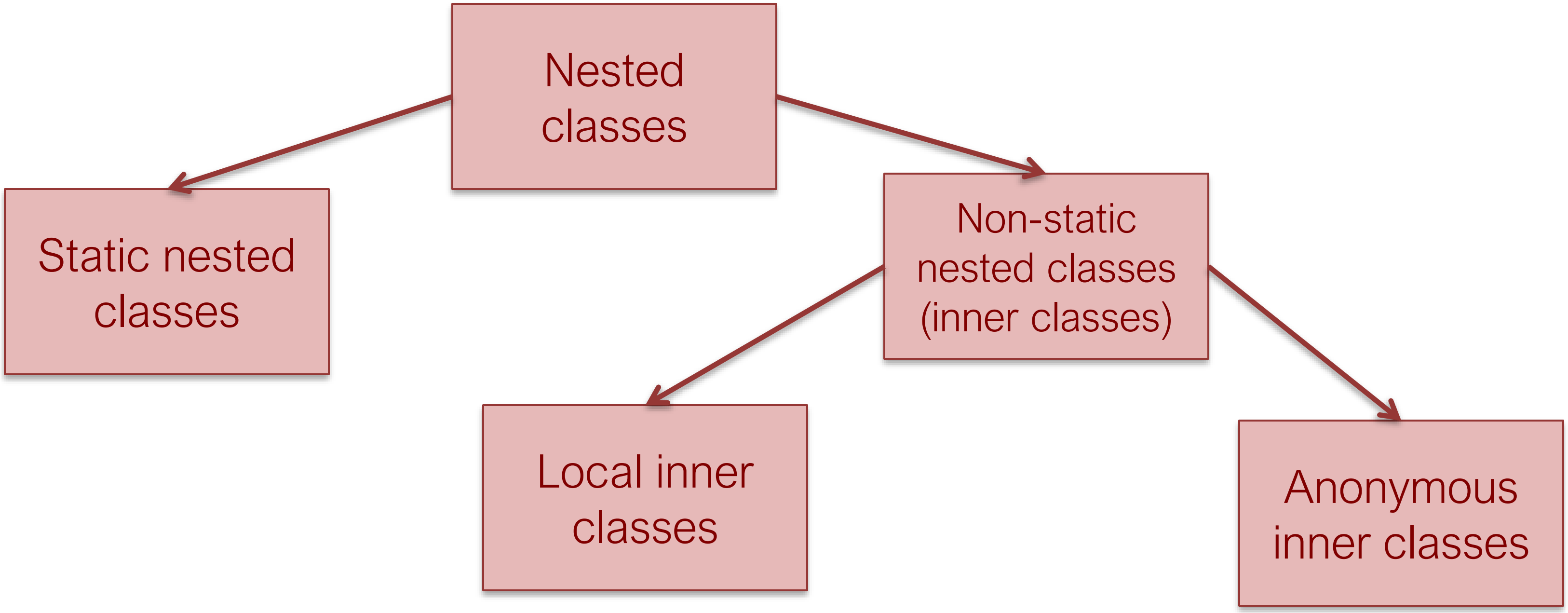
Why Nested Classes?

- Logical grouping of classes
  - Nested class makes sense only within the context of the outer class
- Helps with encapsulation
  - Nested class can access outer class members, even if private.
- Can help with readability of code
  - Better readability == easier maintenance.

Nested Classes

- Nested class – a class defined within some other class

- Nested classes can be:
  - Static classes
  - Non-static (inner classes)



Static Nested Classes

- Behave the same way as top-level classes

```
class A {
//code for A
    static class B {
    //code for B
    }
}
```

- To access a static nested class, we need to use the name of the outer class:

```
    A.B b = new A.B()
```

Inner Classes

- Object of inner classes exist within an instance of the outer class

```
class X {
//code for X
  Class Y {
  //code for Y
  }
}
```

- To access an inner nested class, we need to do so through an instance of the outer class:

```
X x = new X();
  X.Y y = x.new Y();
```

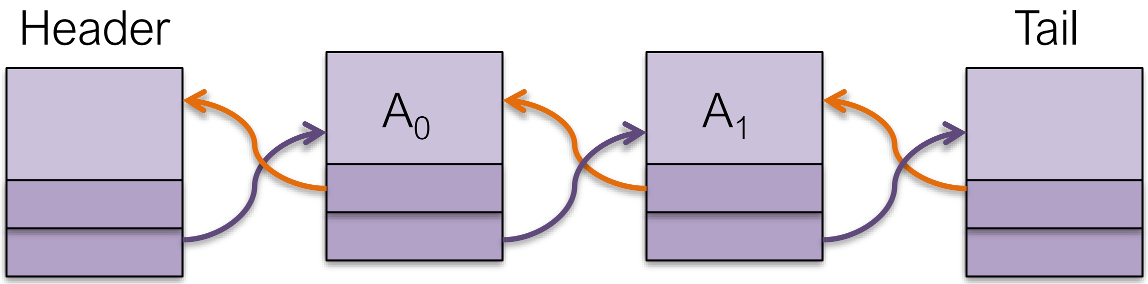
Difference Between Static and Inner Classes

- Static nested classes do not have a direct access to the non-static members of the outer class (non-static variables and methods)
  - Static class must access the non-static members of its enclosing class through an object
- Inner classes have access to all members (static and non-static, including private) of its outer class, and may refer to them directly
  - Used more frequently

Implementation of a Doubly LinkedList

- Doubly linked list – need to provide and maintain links to both ends of the list
- Implemented classes:
  - Class `MyLinkedList`
  - Class `Node` – private nested class, contains the data and links to previous and next nodes
  - Class `LinkedListIterator` – private inner class, implementing the interface `Iterator`

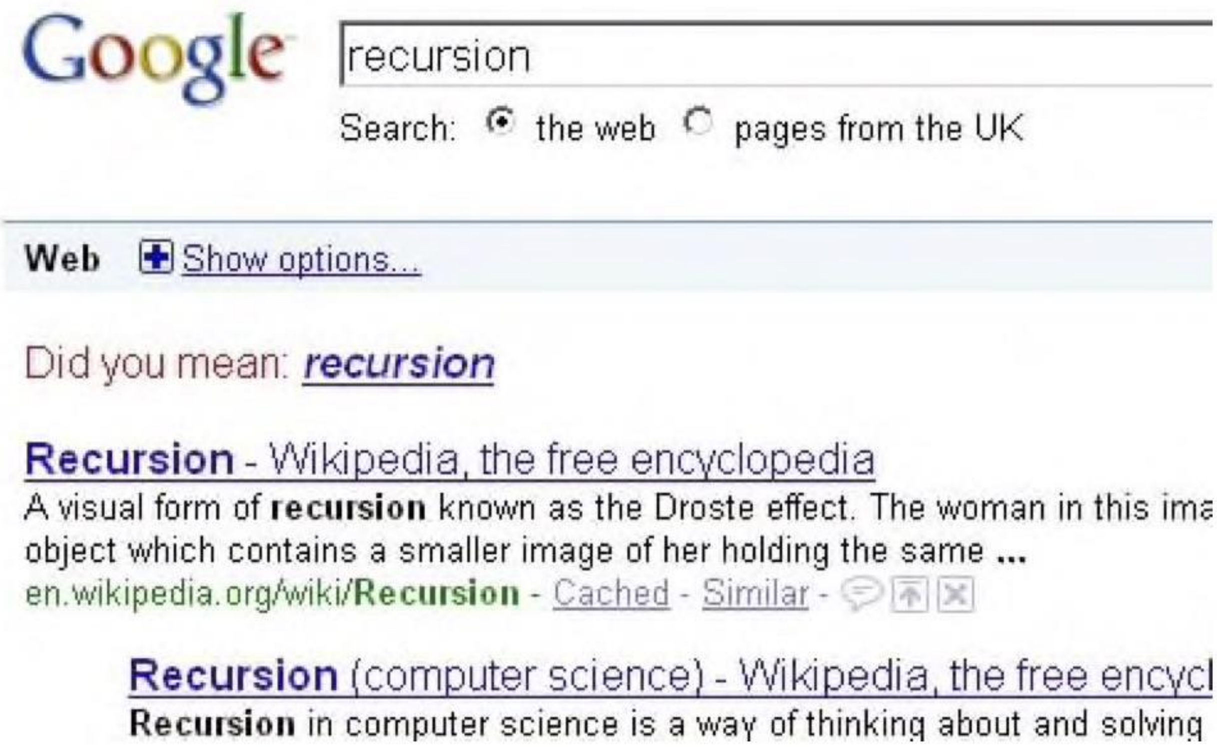
- Sentinel nodes:
  - Header and tail nodes, used to logically represent the beginning and the end markers



Algorithms and Data Structures 1

LIST ADT AND RECURSIVE DATA COLLECTIONS

Recursion



recursion

Search: ● the web ○ pages from the UK

Web ⊞ Show options...

Did you mean: ***recursion***

Recursion - Wikipedia, the free encyclopedia

A visual form of **recursion** known as the Droste effect. The woman in this ima object which contains a smaller image of her holding the same **...**

en.wikipedia.org/wiki/**Recursion** - Cached - Similar -

Recursion (computer science) - Wikipedia, the free encycl

Recursion in computer science is a way of thinking about and solving

[Pictures credit: http://www.telegraph.co.uk/technology/google/6201814/Google-easter-eggs-15-best-hidden-jokes.html]

Recursion

- Recursion – an operation defined in terms of itself
  - Solving a problem recursively means solving smaller occurrences of the same problem

- Recursive programming – an object consist of methods that call themselves to solve some problem

- Can you think of some examples of recursions and recursive programs?

Recursive Algorithm

- Every recursive algorithm consists of:
  - **Base case – at least one** simple occurrence of the problem that can be answered directly
  - **Recursive case** - more complex occurrence that cannot be directly answered, but can be described in terms of smaller occurrences of the same problem

- A crucial part of recursive programming is identifying these cases

Example: Factorials

```
n! = n * (n – 1) * (n – 2) * … * 2 * 1
```

```
public static long factorial(int n) {
    if (n == 1) return 1;
    return n * factorial(n-1);
}
```

Recursive Data Structures

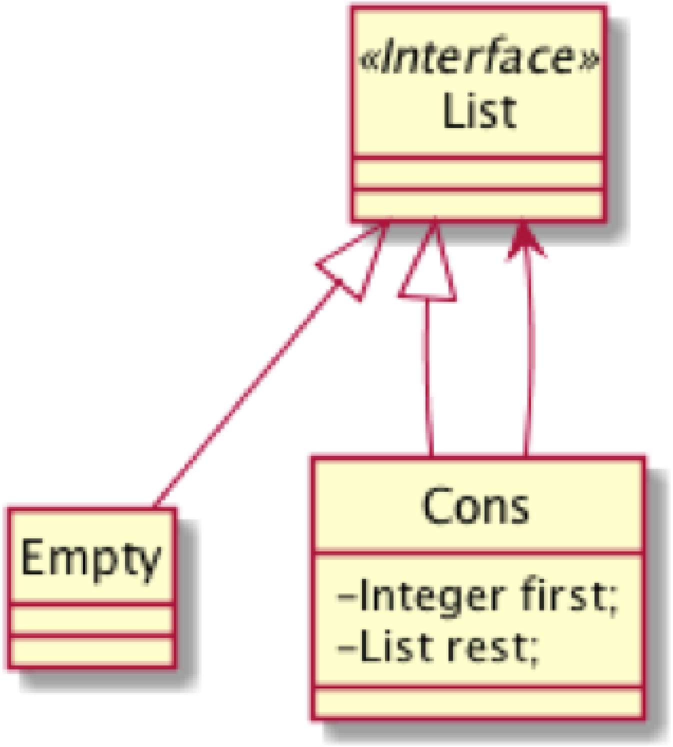
- **Recursive data structure** - a data structure partially composed of smaller or simpler instances of the same data structure

- Is linked list a recursive data structure?
- Let’s see - a linked list is either
  - Null (base case)
  - A node whose next field references a list

Lists as a Recursive Data Collection

- List – an ordered collection (also known as a sequence)

- A linked list is either:
  - Null (base case)
  - A node whose next field references a list



Questions?

References and Reading Material

- Mark Allen Weiss, Data Structures and Algorithm Analysis in Java, chapters 1 through 4
- Oracle, java.util Class Collections, [Online] http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html
- Oracle, Java Tutorials Collections, [Online] https://docs.oracle.com/javase/tutorial/collections/
- Vogella, Java Collections – Tutorial, [Online] http://www.vogella.com/tutorials/JavaCollections/article.html
- Oracle, Java Tutorials, Nester Classes, [Online] https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
- Princetion, Introduction to Programming in Java, Recursion, [Online] http://introcs.cs.princeton.edu/java/23recursion/
- Jeff Ericson, Backtracking, [Online] http://introcs.cs.princeton.edu/java/23recursion/
- Wikibooks, Algorithms/Backtracking, [Online], https://en.wikibooks.org/wiki/Algorithms/Backtracking