RESOURCES - there are MANY

- Recordings
- Lecture Worksheets
- Discussion Slides + Code
- Review Quizzes
- Go over midterm 1 & midterm 2
- Look over PAs
- Create your own practice examples!

Final Review

REMEMBER: These will be very helpful BUT you still need to do additional studying on your own!!!

OUTLINE: click on a link to jump to a section

Java

- ➔ Java Review
- ➔ Interfaces
- ➔ Generics

Time Complexity

- ➔ Runtime

Data Structures

- ➔ ArrayLists
- ➔ Circular ArrayLists
- ➔ LinkedLists
- ➔ Doubly LinkedLists
- ➔ Stacks & Queues
- ➔ Maps & HashTables
- ➔ BSTs
- ➔ Heaps

Algorithms

- ➔ BFS & DFS
- ➔ Sorting

- ➔ Iterators

Java

Java Review

What is wrong with the below snippet of code?

```
String a = “aaa”
String b = “bbb”

if (a.compareTo(b)) {System.out.print(“WHOO”);
```

What is wrong with the below snippet of code?

```
String a = “aaa”
String b = “bbb”

if (a.compareTo(b)) {System.out.print(“WHOO”);
```

compareTo returns an integer, NOT a boolean value

What is wrong with the below snippet of code?

```
int compareValues(String a, String b) {
    if (a.compareTo(b) > 0) { return 1; }
}
```

What is wrong with the below snippet of code?

```
int compareValues(String a, String b) {
    if (a.compareTo(b) > 0) { return 1; }
}
```

Error: missing return statement

The if statement may not be entered. There needs to be a return statement at the end of the method

FILL IN THE BLANK

```
int compareValues(String a, String b) {
    if (a.compareTo(b) > 0) { return 1; }
     return 0;
}
```

compareValues(“a block”, “a black bear”) returns ______

compareValues(“11”, “12”) returns ______

compareValues(“I love CSE”, “I love CSE 12”) returns ______

compareValues(“San Diego”, “San Francisco”) returns ______

FILL IN THE BLANK

```
int compareValues(String a, String b) {
    if (a.compareTo(b) > 0) { return 1; }
     return 0;
}
```

compareValues(“a block”, “a black bear”) returns **1**

compareValues(“11”, “12”) returns **0**

compareValues(“I love CSE”, “I love CSE 12”) returns **0**

compareValues(“San Diego”, “San Francisco”) returns **0**

FILL IN THE BLANK

Fields with the modifier final are assigned _______ in the _______

Static fields are stored on the _______

Non-Static fields are stored on the _______

Variables are stored on the _______

FILL IN THE BLANK

Fields with the modifier final are assigned **once** in the **constructor**

Static fields are stored on the **heap**

Non-Static fields are stored on the **heap**

Variables are stored on the **stack**

FILL IN THE BLANK

_______ update example: c1 = c2 (c1 and c2 are both objects)

_______ update example: c2.color = “blue”

Both field and variable updates change only _______ value

_______ updates: update ONE variable on the STACK

_______ updates: update ONE field on the HEAP

FILL IN THE BLANK

Variable update example: c1 = c2 (c1 and c2 are both objects)

Field update example: c2.color = “blue”

Both field and variable updates change only **ONE** value

Variable updates: update ONE variable on the STACK

Field updates: update ONE field on the HEAP

What does Q2 print?

A: 1
B: 0
C: 9
D: 10
E: Something else

```
public class Q2 {
  public static void f(Coord c) {
    Car car = new Car("blue", c);
    car.location.row = 10;
    car.location.col = 9;
  }
  public static int question() {
    Coord unit = new Coord(1, 1);
    Car blackCar = new Car("black", unit);
    f(unit);
    return blackCar.location.row;
  }
  public static void main(String[] args) {
    System.out.println(question());
  }
}
```

@Z	[] *An empty array for args, a detail not used in this example*

What does Q2 print?

A: 1
B: 0
C: 9
D: 10
E: Something else

```
public class Q2 {
  public static void f(Coord c) {
    Car car = new Car("blue", c);
    car.location.row = 10;
    car.location.col = 9;
  }
  public static int question() {
    Coord unit = new Coord(1, 1);
    Car blackCar = new Car("black", unit);
    f(unit);
    return blackCar.location.row;
  }
  public static void main(String[] args) {
    System.out.println(question());
  }
}
```

Q2.f(@A)	
c	@A
car	@C
returns: void	

Q2.question()	
unit	@A
blackCar	@B
returns: 10	

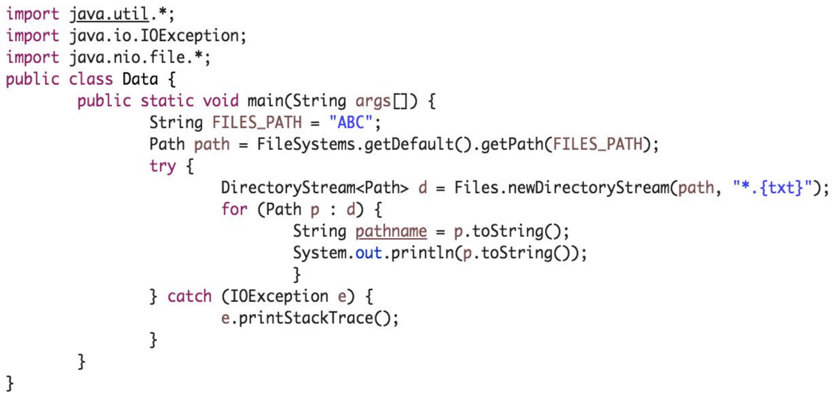
Q2.main(@Z)	
args	@Z
returns: nothing (void)	

@A	Coord row 10 col 9
@B	Car color “black” location @A
@C	Car color “blue” location @A
@Z	[] *An empty array for args, a detail not used in this example*

How long do changes persist after a method ends?

Changes on the heap last forever, never automatically undone, stays until a field update changes it again, if you want to have the behavior of changing a field and reverting it back you must implement it

What will be the output?



```java
import java.util.*;
import java.io.IOException;
import java.nio.file.*;
public class Data {
        public static void main(String args[]) {
                String FILES_PATH = "ABC";
                Path path = FileSystems.getDefault().getPath(FILES_PATH);
                try {
                        DirectoryStream<Path> d = Files.newDirectoryStream(path, "*.{txt}");
                        for (Path p : d) {
                                String pathname = p.toString();
                                System.out.println(p.toString());
                                }
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

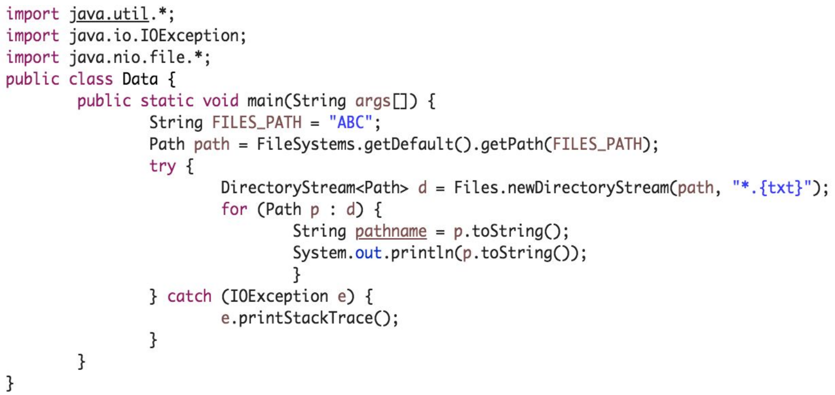
a) A : NoSuchFileException

b) B : NotDirectoryException

c) C : No exceptions will occur

d) D : It will print ABC/ABC.txt
ABC/DEF.txt

What will be the output?



```
import java.util.*;
import java.io.IOException;
import java.nio.file.*;
public class Data {
        public static void main(String args[]) {
                String FILES_PATH = "ABC";
                Path path = FileSystems.getDefault().getPath(FILES_PATH);
                try {
                        DirectoryStream<Path> d = Files.newDirectoryStream(path, "*.{txt}");
                        for (Path p : d) {
                                String pathname = p.toString();
                                System.out.println(p.toString());
                                }
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

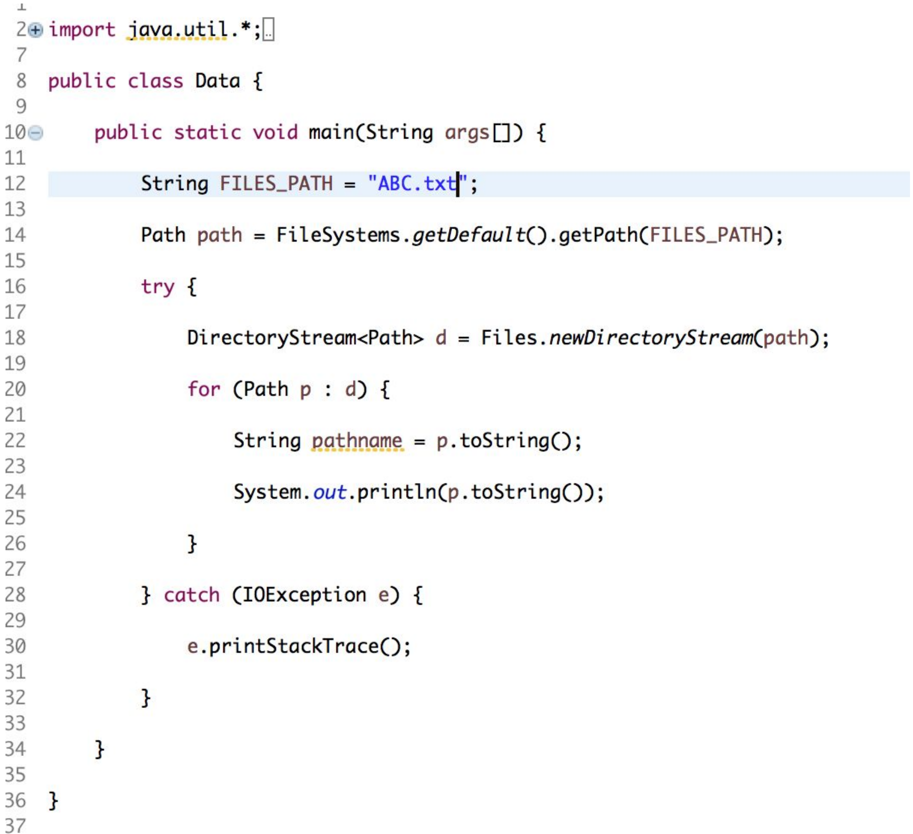
a) A : NoSuchFileException

b) B : NotDirectoryException

c) C : No exceptions will occur

d) D : It will print ABC/ABC.txt
ABC/DEF.txt

What will be the output?



```java
import java.util.*;

public class Data {

    public static void main(String args[]) {

        String FILES_PATH = "ABC.txt";

        Path path = FileSystems.getDefault().getPath(FILES_PATH);

        try {

            DirectoryStream<Path> d = Files.newDirectoryStream(path);

            for (Path p : d) {

                String pathname = p.toString();

                System.out.println(p.toString());

            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

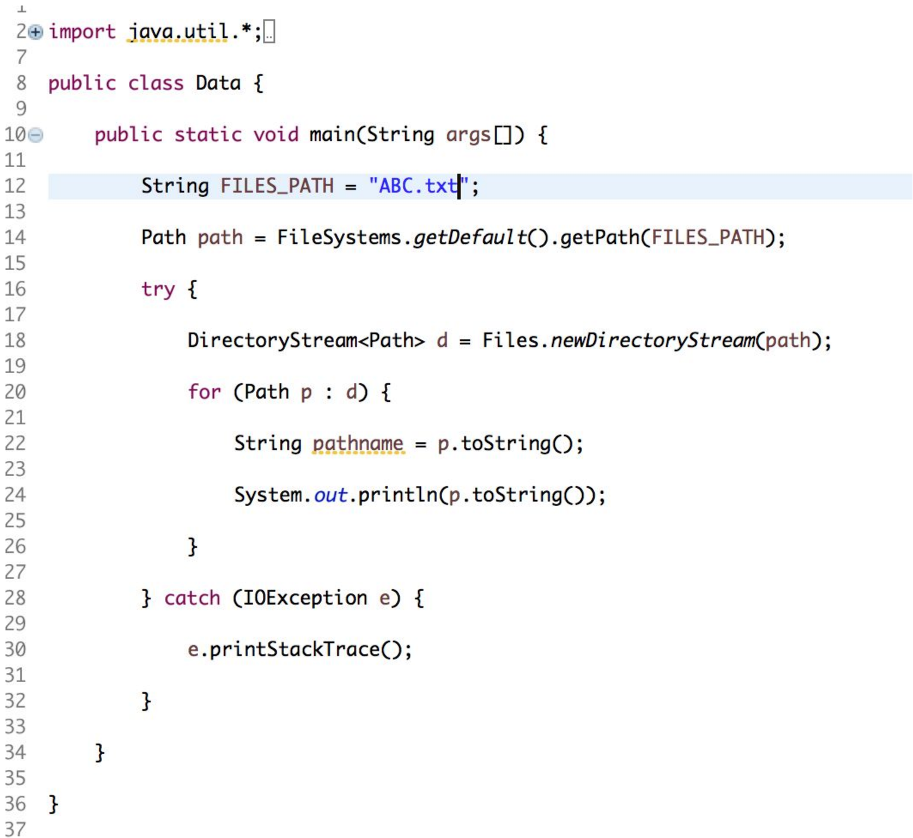
a) A : NoSuchFileException

b) B : NotDirectoryException

c) C : No exceptions will occur

d) D : It will print ABC/ABC.txt
ABC/DEF.txt

What will be the output?



```
import java.util.*;

public class Data {

    public static void main(String args[]) {

        String FILES_PATH = "ABC.txt";

        Path path = FileSystems.getDefault().getPath(FILES_PATH);

        try {

            DirectoryStream<Path> d = Files.newDirectoryStream(path);

            for (Path p : d) {

                String pathname = p.toString();

                System.out.println(p.toString());

            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

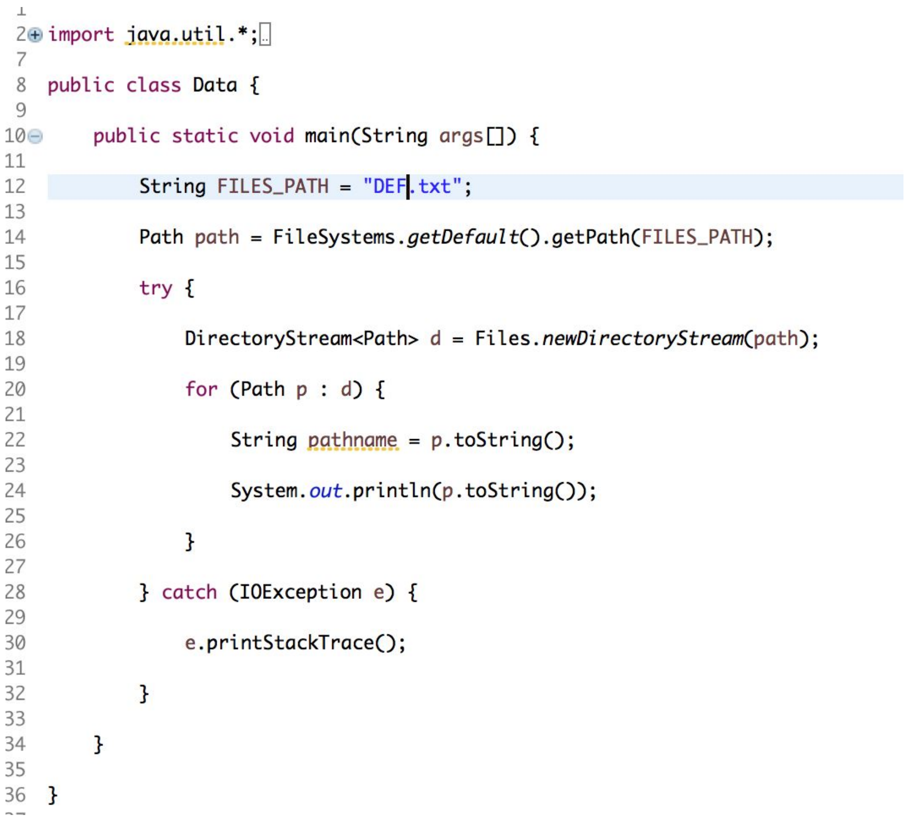
a) A : NoSuchFileException

b) B : NotDirectoryException

c) C : No exceptions will occur

d) D : It will print ABC/ABC.txt
ABC/DEF.txt

What will be the output?



```
import java.util.*;

public class Data {

    public static void main(String args[]) {

        String FILES_PATH = "DEF.txt";

        Path path = FileSystems.getDefault().getPath(FILES_PATH);

        try {

            DirectoryStream<Path> d = Files.newDirectoryStream(path);

            for (Path p : d) {

                String pathname = p.toString();

                System.out.println(p.toString());

            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

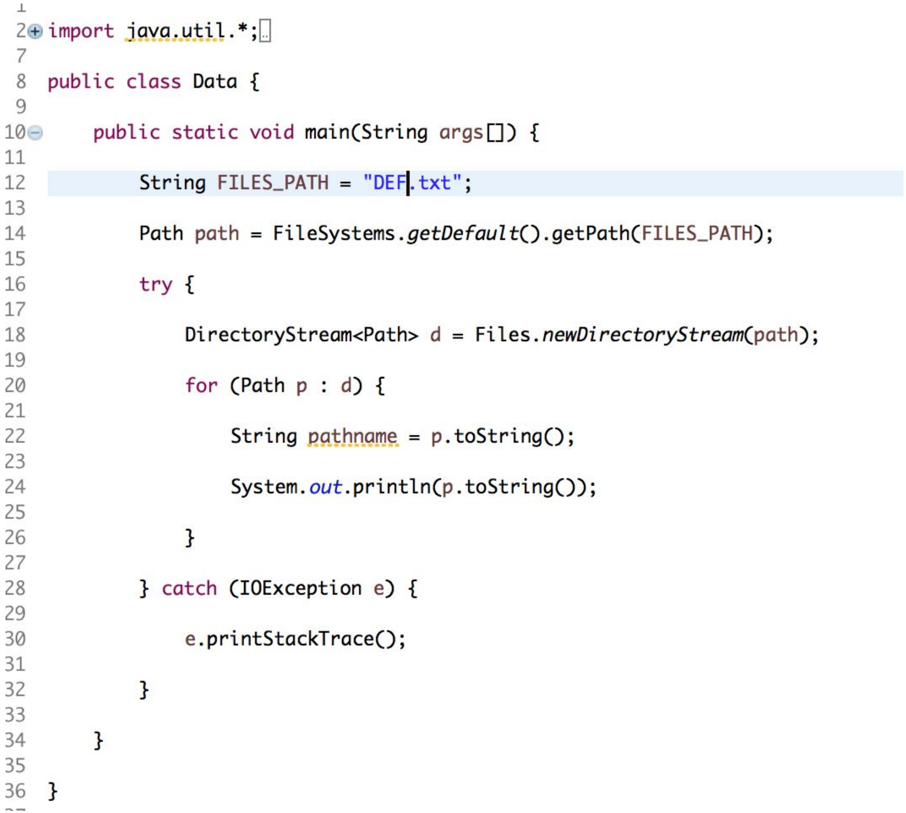
a) A : NoSuchFileException

b) B : NotDirectoryException

c) C : No exceptions will occur

d) D : It will print ABC/ABC.txt
ABC/DEF.txt

What will be the output?



```
import java.util.*;

public class Data {

    public static void main(String args[]) {

        String FILES_PATH = "DEF.txt";

        Path path = FileSystems.getDefault().getPath(FILES_PATH);

        try {

            DirectoryStream<Path> d = Files.newDirectoryStream(path);

            for (Path p : d) {

                String pathname = p.toString();

                System.out.println(p.toString());

            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

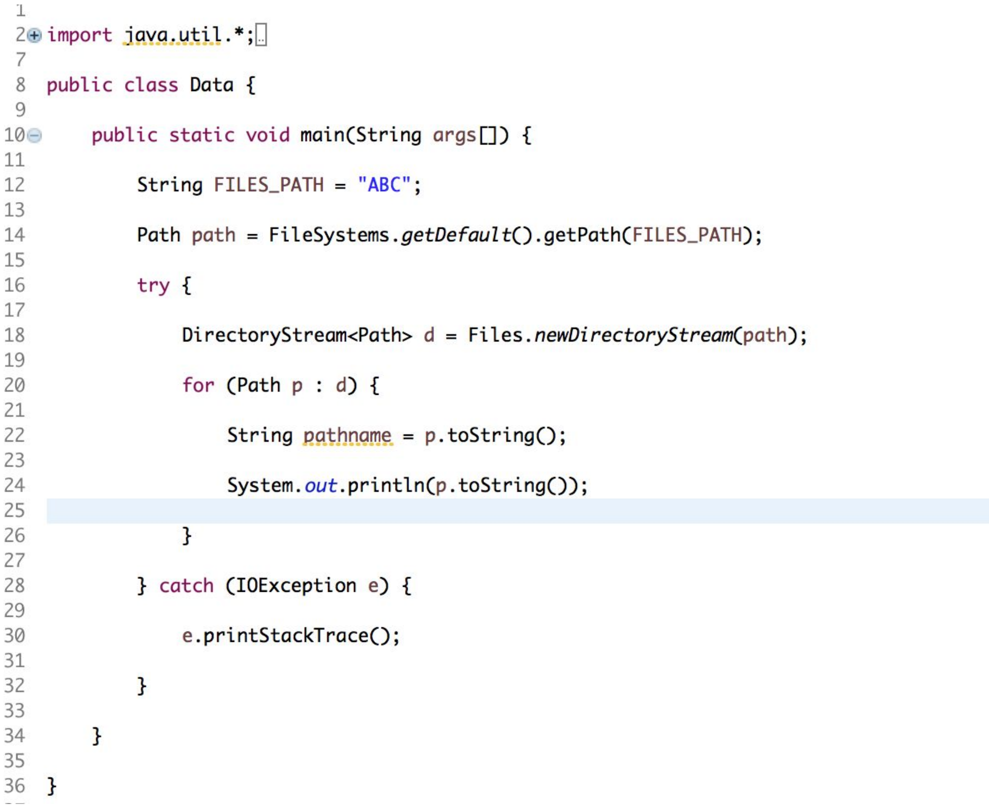
a) A : NoSuchFileException

b) B : NotDirectoryException

c) C : No exceptions will occur

d) D : It will print ABC/ABC.txt
ABC/DEF.txt

What will be the output?



```java

import java.util.*;

public class Data {

    public static void main(String args[]) {

        String FILES_PATH = "ABC";

        Path path = FileSystems.getDefault().getPath(FILES_PATH);

        try {

            DirectoryStream<Path> d = Files.newDirectoryStream(path);

            for (Path p : d) {

                String pathname = p.toString();

                System.out.println(p.toString());


            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

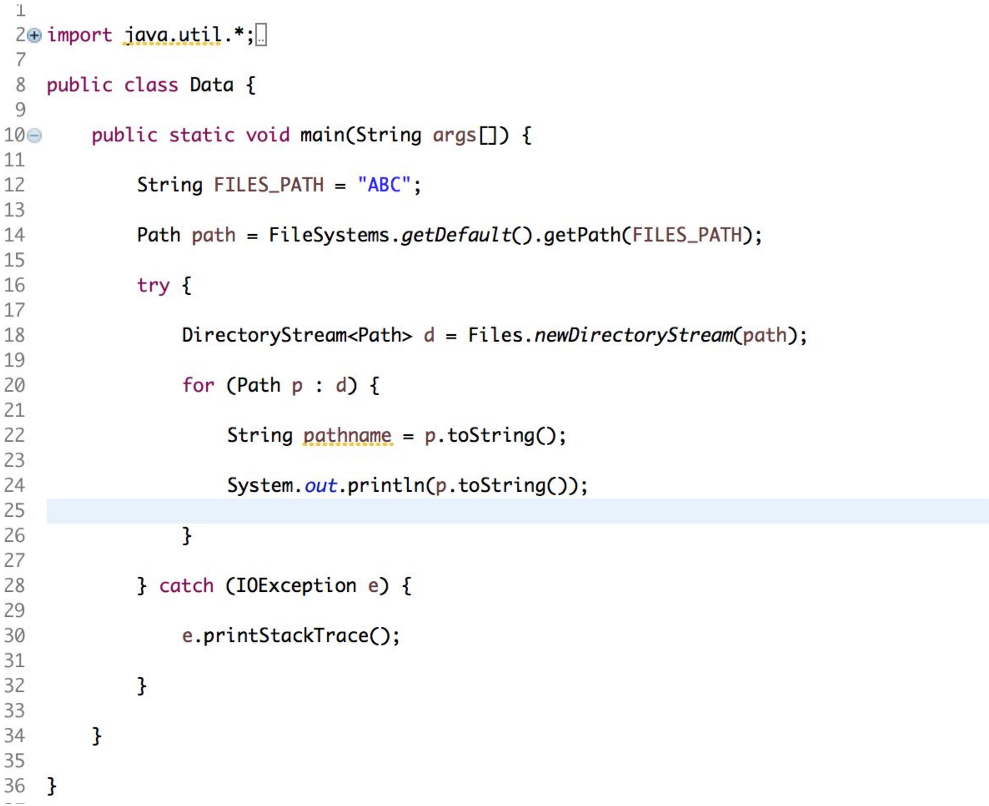
a) A : NoSuchFileException

b) B : NotDirectoryException

c) C : It will print ABC/ABC.txt
ABC/DEF.txt

d) D : It will print ABC/ABC.txt
ABC/DEF.txt
ABC/download.png

What will be the output?



```java

import java.util.*;

public class Data {

    public static void main(String args[]) {

        String FILES_PATH = "ABC";

        Path path = FileSystems.getDefault().getPath(FILES_PATH);

        try {

            DirectoryStream<Path> d = Files.newDirectoryStream(path);

            for (Path p : d) {

                String pathname = p.toString();

                System.out.println(p.toString());


            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

a) A : NoSuchFileException

b) B : NotDirectoryException

c) C : It will print ABC/ABC.txt
ABC/DEF.txt

d) D : It will print ABC/ABC.txt
ABC/DEF.txt
ABC/download.png

Interfaces

FILL IN THE BLANK

An interface...

Contains _____ and _____ _____ only. Classes implementing the interface must override and define these methods.

To define, use keyword "_____".

To implement, use keyword "_____".

Allows code reuse by having multiple classes that share a type (subtype _____).

FILL IN THE BLANK

An interface...

Contains **fields** and **method signatures** only. Classes implementing the interface must override and define these methods.

To define, use keyword “**interface**”.

To implement, use keyword “**implements**”.

Allows code reuse by having multiple classes that share a type (subtype **polymorphism**).

Which one of the lines below will result in a compile error?

```
Student s1 = new Student(“Andrew”)

User u1 = new Student(“Jane”);

Student s2 = new User(“Rachel”);
```

Which one of the lines below will result in a compile error?

```
Student s1 = new Student(“Andrew”)

User u1 = new Student(“Jane”);

Student s2 = new User(“Rachel”);
```

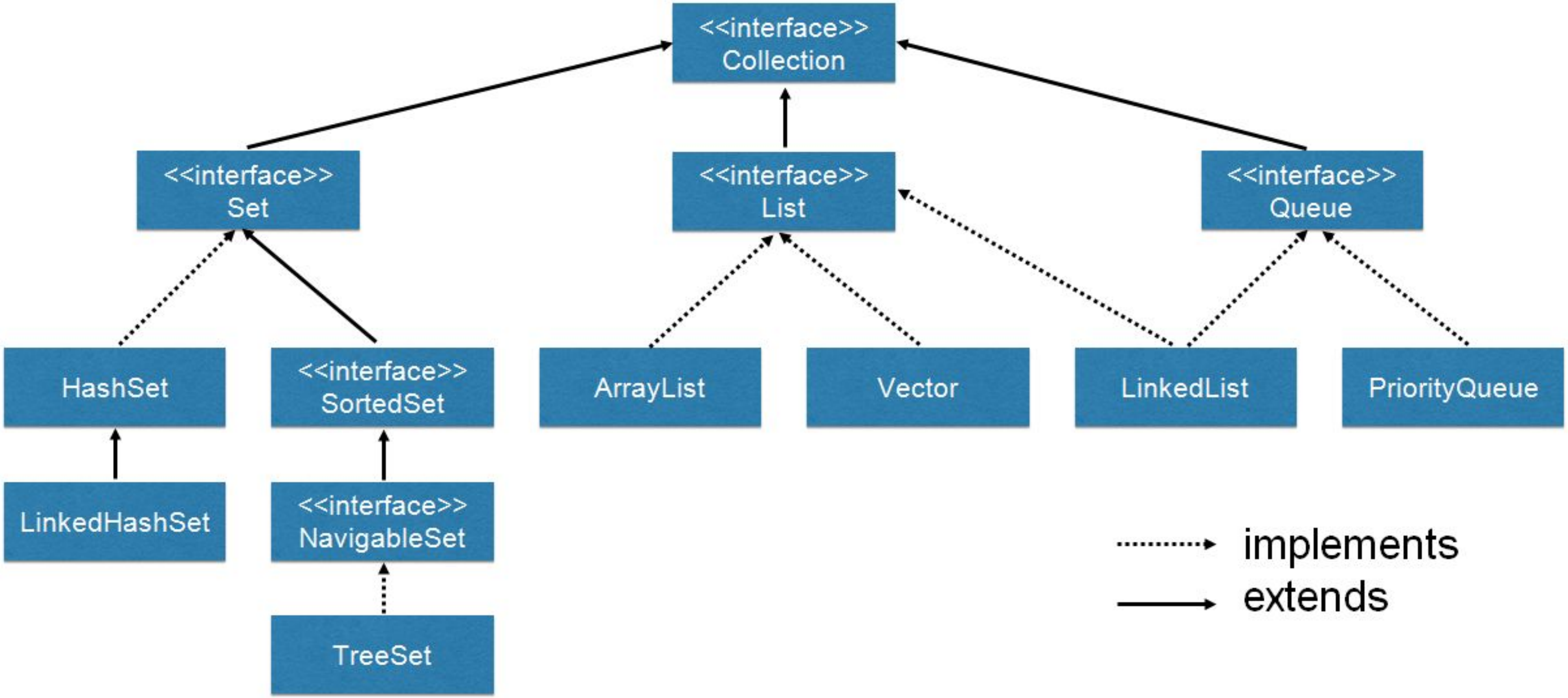
An interface cannot be initialized on its own. Interfaces are utilized by implementing them through other concrete classes. So you cannot initialize User since it is an interface.

Collection Interface

- The Collection interface is the foundation upon which the collections framework is built.

- It declares the core methods that all collections will have.

Collection Interface



What all can we do with Collection?

There are a few basic operations you'll normally use with collections:

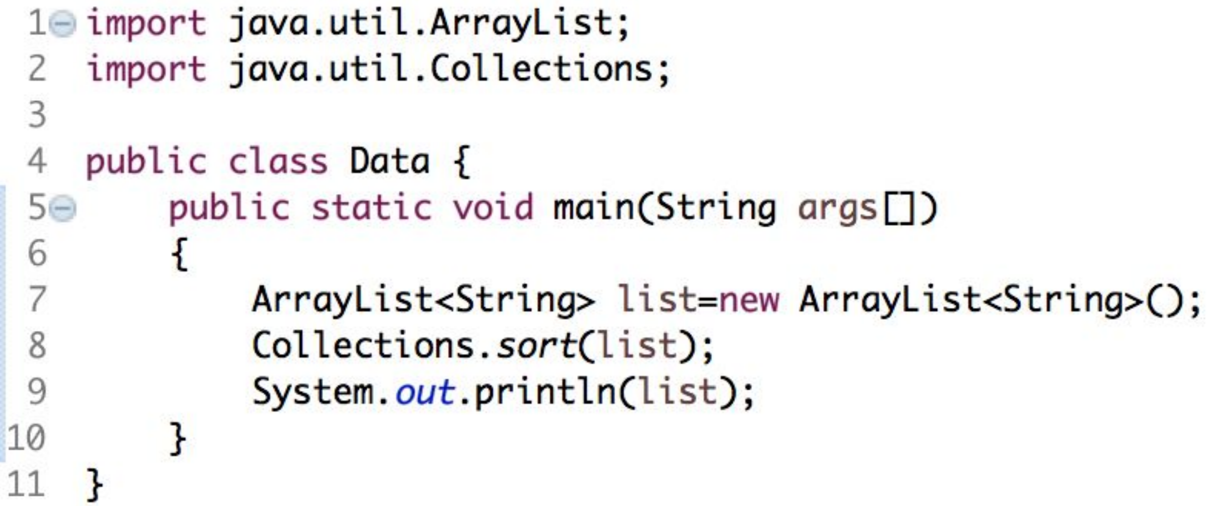
- Add objects to the collection
- Remove objects from the collection
- Find out if an object (or group of objects) is in the collection
- Retrieve an object from the collection (without removing it)
- Iterate through the collection, looking at each element (object) one after another

Collections Class

- This class consists exclusively of static methods that operate on or return collections.

- Java Collection class throws a **NullPointerException** if the collections or class objects provided to them are null.

What will be the output?



```
1  import java.util.ArrayList;
2  import java.util.Collections;
3
4  public class Data {
5      public static void main(String args[])
6      {
7          ArrayList<String> list=new ArrayList<String>();
8          Collections.sort(list);
9          System.out.println(list);
10     }
11 }
```

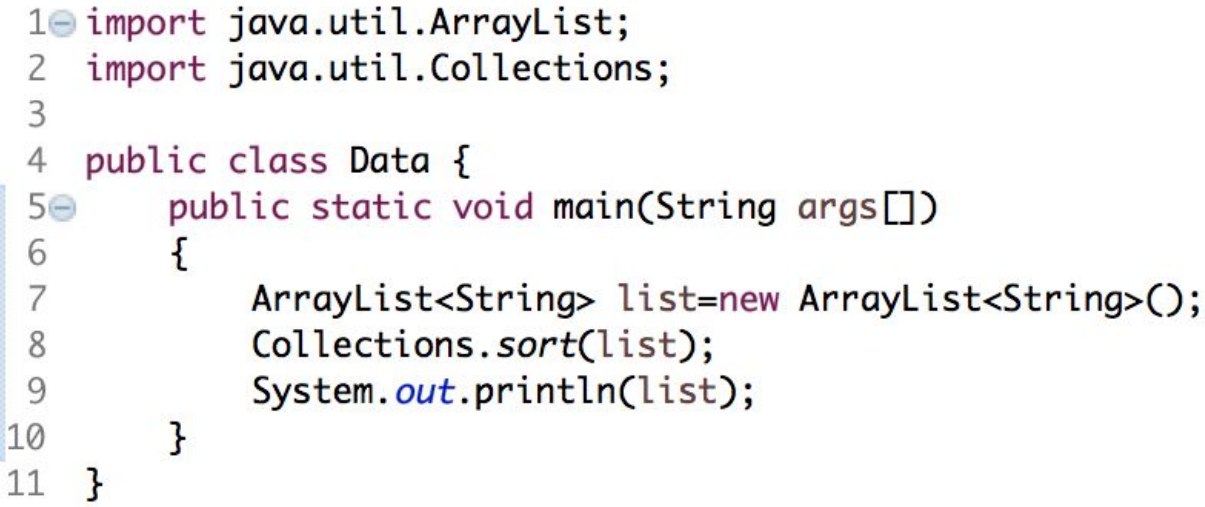
A) NullPointerException at line 8

B) NullPointerException at line 9

C) []

D) [null]

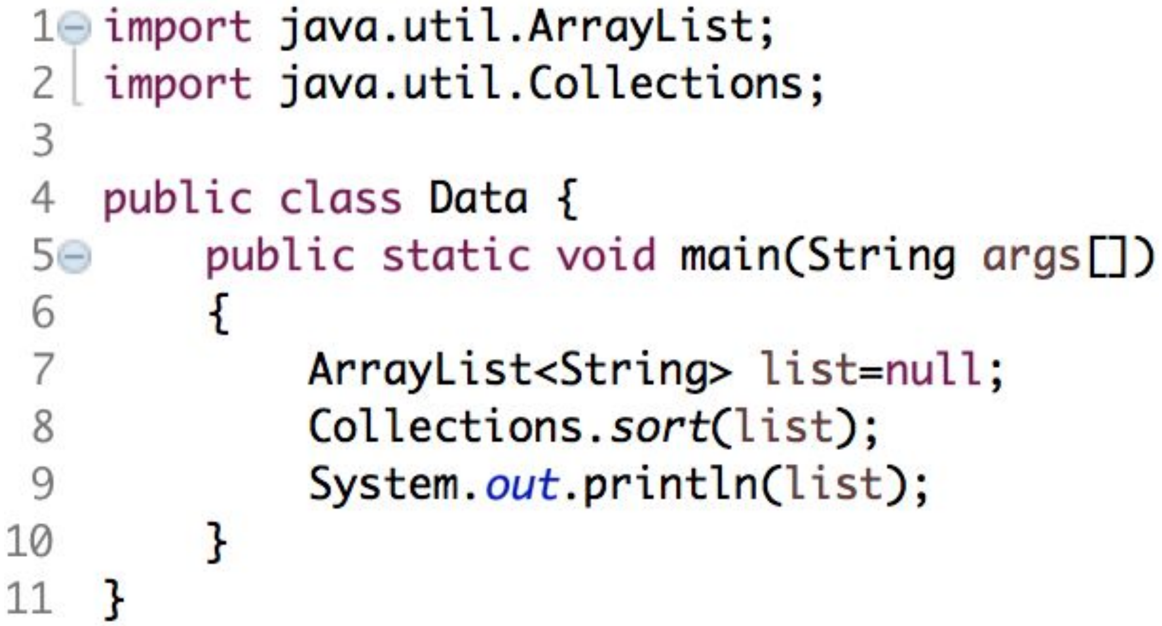
What will be the output?



```
1  import java.util.ArrayList;
2  import java.util.Collections;
3
4  public class Data {
5      public static void main(String args[])
6      {
7          ArrayList<String> list=new ArrayList<String>();
8          Collections.sort(list);
9          System.out.println(list);
10     }
11 }
```

A) NullPointerException at line 8
B) NullPointerException at line 9
C) []
D) [null]

What will be the output?



```
1  import java.util.ArrayList;
2  import java.util.Collections;
3
4  public class Data {
5          public static void main(String args[])
6          {
7                  ArrayList<String> list=null;
8                  Collections.sort(list);
9                  System.out.println(list);
10         }
11 }
```

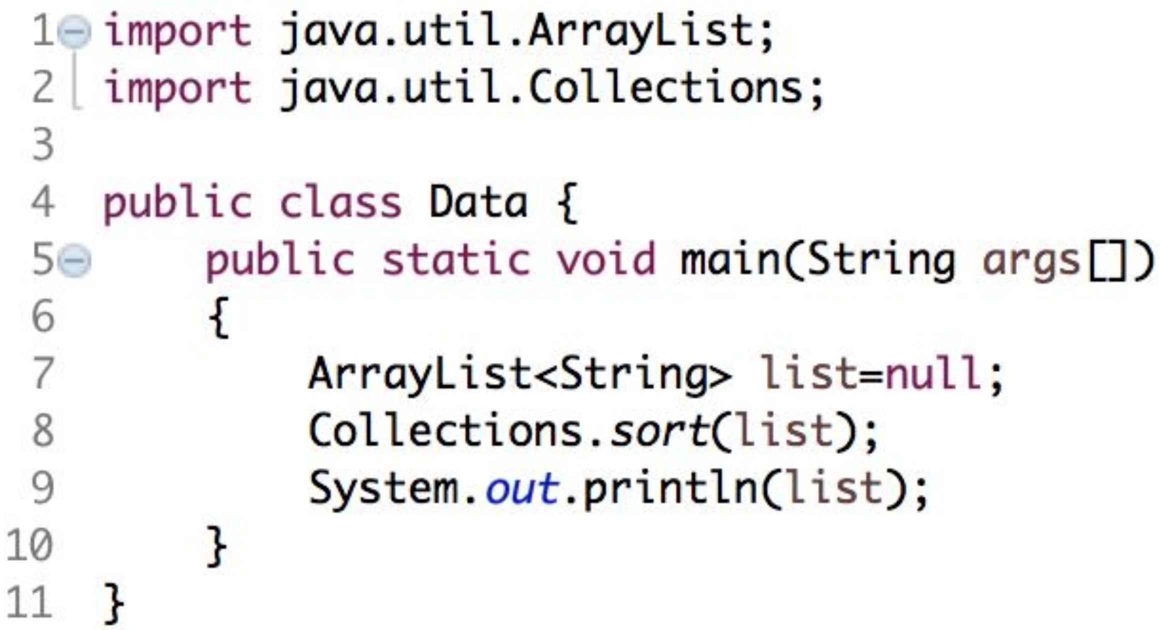
A) NullPointerException at line 8

B) NullPointerException at line 9

C) []

D) [null]

What will be the output?



```
1  import java.util.ArrayList;
2  import java.util.Collections;
3
4  public class Data {
5      public static void main(String args[])
6      {
7          ArrayList<String> list=null;
8          Collections.sort(list);
9          System.out.println(list);
10     }
11 }
```

A) NullPointerException at line 8

B) NullPointerException at line 9

C) []

D) [null]

Collections Class v/s Collection Interface

- Collections is a class, with static utility methods

- Collection is an interface with declarations of the methods common to most collections including add(), remove(), size() and iterator().

Collections.sort()

Collections.sort() in Java

- **java.util.Collections.sort()** method is present in java.util.Collections class.

- It is used to sort the elements present in the specified list of Collection in ascending order.

- Structure:
  public static void sort(List myList)
  myList : A List type object we want to sort.

- **This method doesn't return anything**

What will be the output?

```
 import java.util.ArrayList;
import java.util.Collections;
public class Data {
public static void main(String args[])
{
ArrayList<String> list=new ArrayList<String>();
ArrayList<String> newList=new ArrayList<String>();
list.add("Kevin");
list.add("Emily");
list.add("edward");
list.add("Kim");
newList = Collections.sort(list);
System.out.println(newList);
}
}
```

A) [edward, Emily, Kevin, Kim]

B) [Emily, edward, Kevin, Kim]

C) Runtime Exception

D) Compile Time Error

What will be the output?

```
 import java.util.ArrayList;
import java.util.Collections;
public class Data {
public static void main(String args[])
{
ArrayList<String> list=new ArrayList<String>();
ArrayList<String> newList=new ArrayList<String>();
list.add("Kevin");
list.add("Emily");
list.add("edward");
list.add("Kim");
newList = Collections.sort(list);
System.out.println(newList);
}
}
```

A) [edward, Emily, Kevin, Kim]
B) [Emily, edward, Kevin, Kim]
C) Runtime Exception
D) Compile Time Error

Collections.sort() in Java

- Java provides **two interfaces** to sort objects using data members of the class:

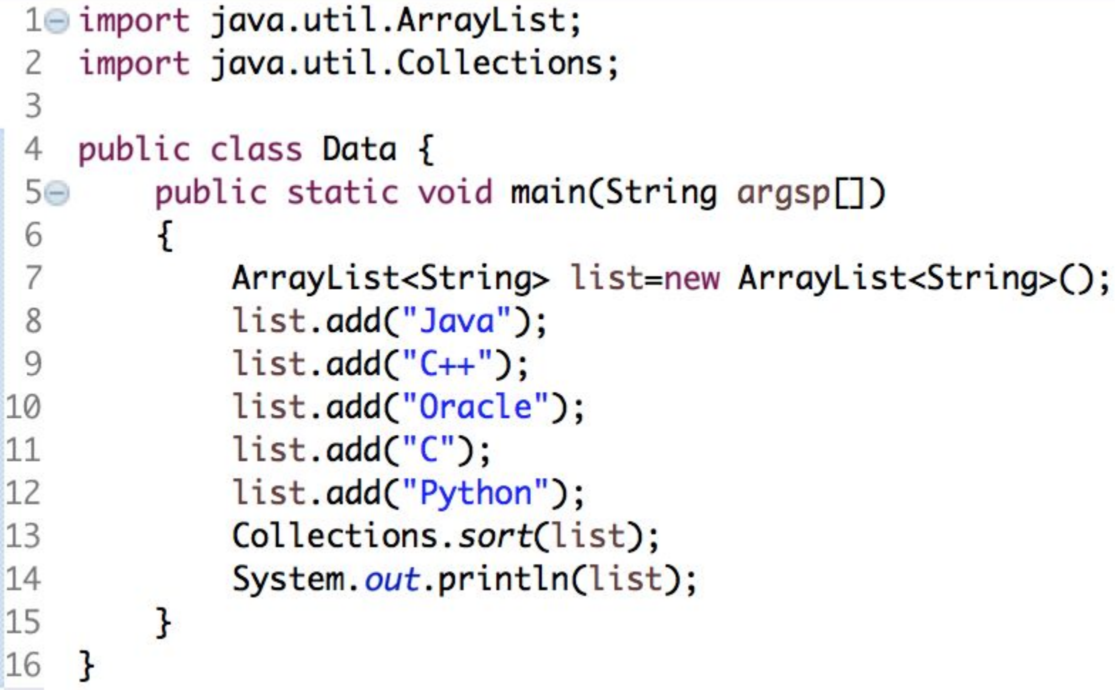
1. Comparable

1. Comparator

Using Comparable Interface

- A comparable object is capable of comparing itself with another object.
- The class itself must implements the **java.lang.Comparable** interface to compare its instances.
- Basically, a list can be sorted if only all of its elements are mutually comparable by implementing the Comparable interface. If a class implements the Comparable interface, it is considered as having natural ordering which allows objects of that class to be sorted by the **Collections.sort(list)** method.
- All basic data type wrapper classes in Java have natural ordering: String, Character, Byte, Date, Integer, Float, etc.
- Basically, a list can be sorted if only all of its elements are mutually comparable by implementing the Comparable interface.
- If a class implements the Comparable interface, it is considered as having natural ordering which allows objects of that class to be sorted by the Collections.sort(list) method.
- All basic data type wrapper classes in Java have natural ordering: String, Character, Byte, Date, Integer, Float, etc. Here are some examples:

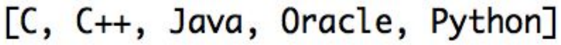
A simple example



```java
import java.util.ArrayList;
import java.util.Collections;

public class Data {
    public static void main(String argsp[])
    {
        ArrayList<String> list=new ArrayList<String>();
        list.add("Java");
        list.add("C++");
        list.add("Oracle");
        list.add("C");
        list.add("Python");
        Collections.sort(list);
        System.out.println(list);
    }
}
```

Output:



```
[C, C++, Java, Oracle, Python]
```

Using Comparator Interface

- Unlike Comparable, Comparator is external to the element type we are comparing.

- It’s a separate class.

Using Comparator Interface

We have to override the following function:

int compare(T obj1, T obj 2)

- Compares its two arguments for order.
- Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

Using Comparator Interface

- We create multiple separate classes (that implement Comparator) to compare by different members.

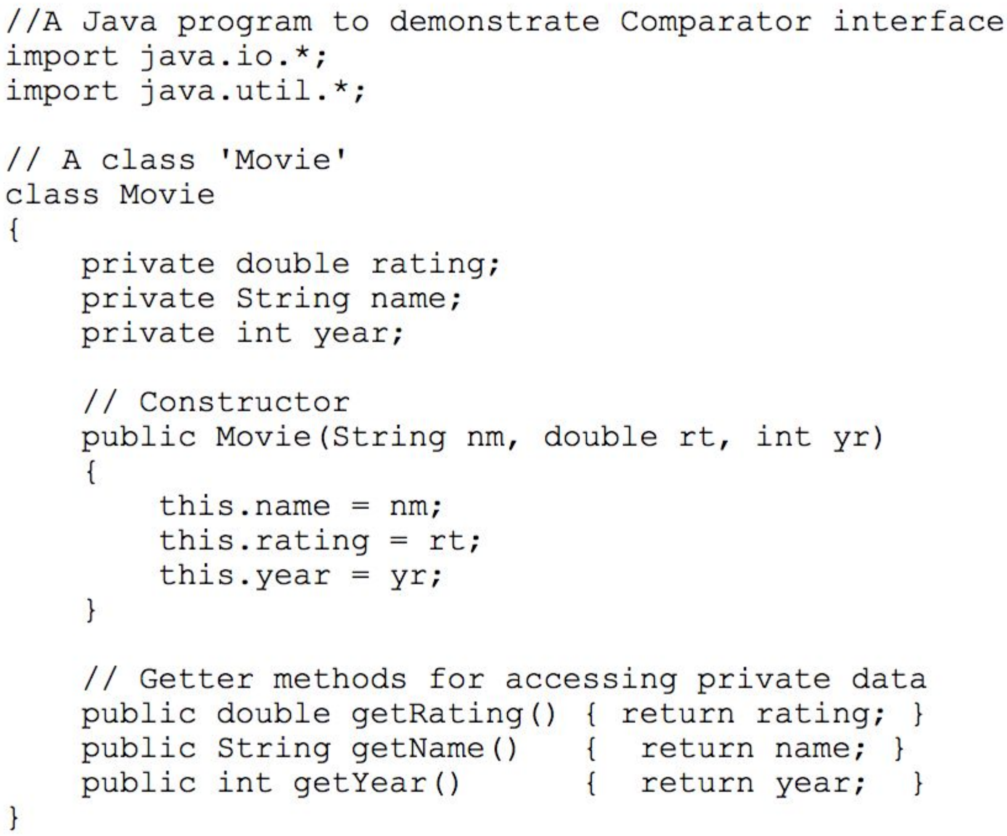
- Collections class has a sort() method that takes Comparator as an input.

- The sort() method invokes the compare() to sort objects.

Example:

- Consider a Movie class that has members like, rating, name, year.
- Suppose we wish to sort a list of Movies based on rating.
- To compare movies by Rating, we need to do 3 things :

1. Create a class that implements Comparator (and thus the compare() method)
2. Make an instance of the Comparator class.
3. Call the sort() method, giving it both the list and the instance of the class that implements Comparator.



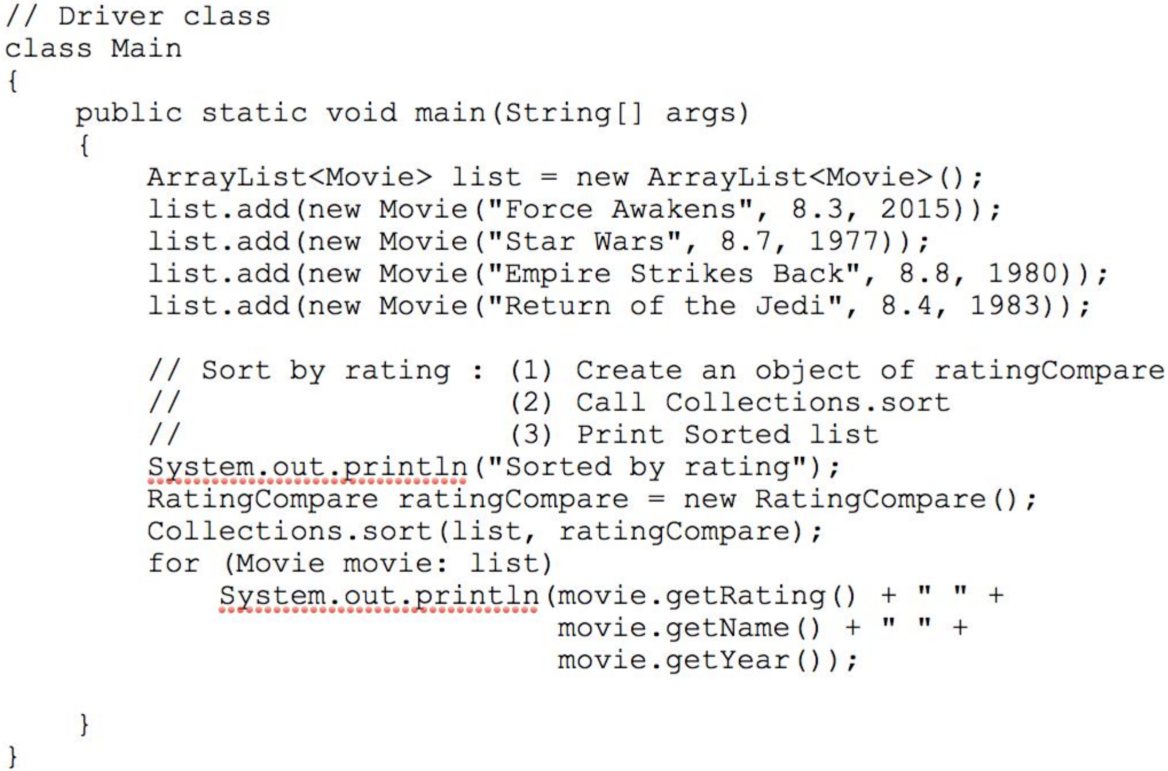
```
//A Java program to demonstrate Comparator interface
import java.io.*;
import java.util.*;

// A class 'Movie'
class Movie
{
    private double rating;
    private String name;
    private int year;

    // Constructor
    public Movie(String nm, double rt, int yr)
    {
        this.name = nm;
        this.rating = rt;
        this.year = yr;
    }

    // Getter methods for accessing private data
    public double getRating() { return rating; }
    public String getName()   {  return name; }
    public int getYear()      {  return year;   }
}
```

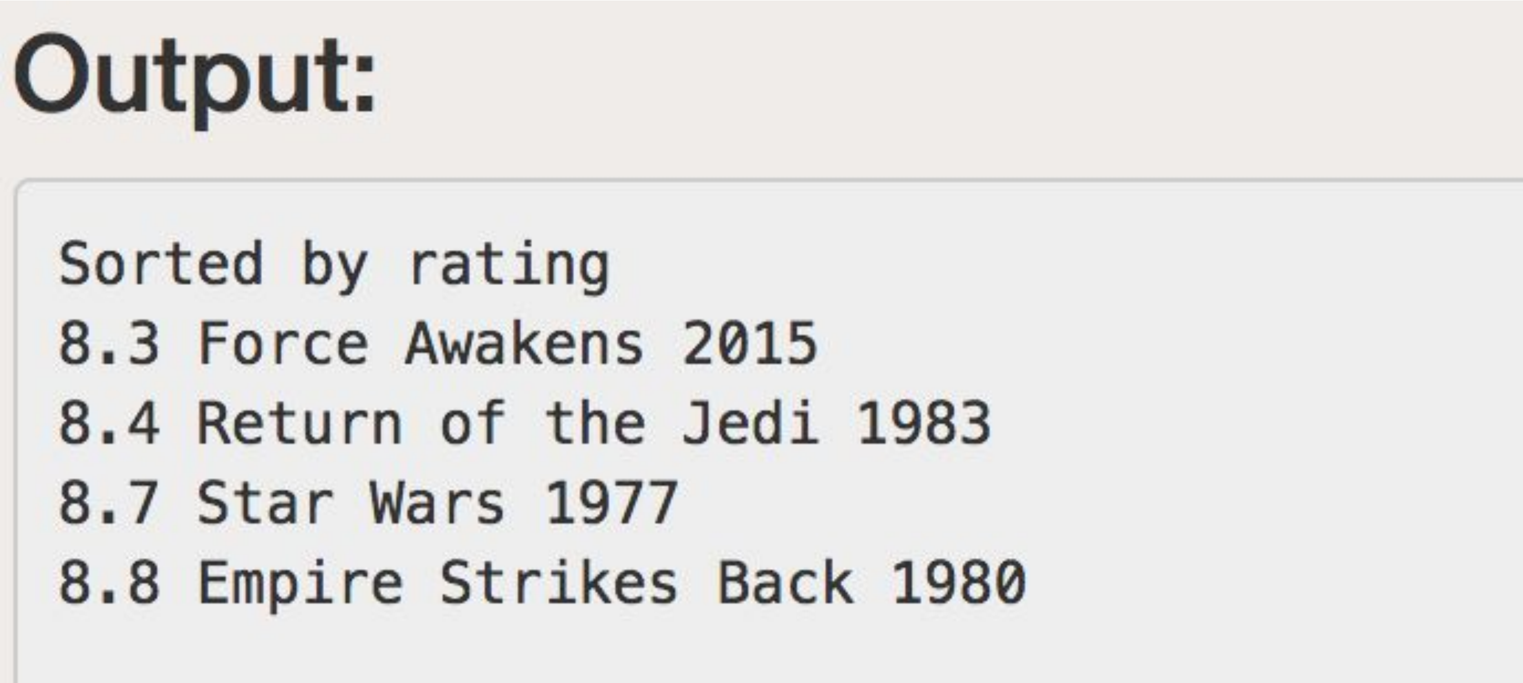
```
// Class to compare Movies by ratings
class RatingCompare implements Comparator<Movie>
{
    public int compare(Movie m1, Movie m2)
    {
        if (m1.getRating() < m2.getRating()) return -1;
        if (m1.getRating() > m2.getRating()) return 1;
        else return 0;
    }
}
```



```
// Driver class
class Main
{
    public static void main(String[] args)
    {
        ArrayList<Movie> list = new ArrayList<Movie>();
        list.add(new Movie("Force Awakens", 8.3, 2015));
        list.add(new Movie("Star Wars", 8.7, 1977));
        list.add(new Movie("Empire Strikes Back", 8.8, 1980));
        list.add(new Movie("Return of the Jedi", 8.4, 1983));

        // Sort by rating : (1) Create an object of ratingCompare
        //                  (2) Call Collections.sort
        //                  (3) Print Sorted list
        System.out.println("Sorted by rating");
        RatingCompare ratingCompare = new RatingCompare();
        Collections.sort(list, ratingCompare);
        for (Movie movie: list)
            System.out.println(movie.getRating() + " " +
                               movie.getName() + " " +
                               movie.getYear());

    }
}
```



Output:

```
Sorted by rating
8.3 Force Awakens 2015
8.4 Return of the Jedi 1983
8.7 Star Wars 1977
8.8 Empire Strikes Back 1980
```

Generics

Instantiation

Given the following code, which of the instantiations in `main` are valid?

```
public interface MyInterface<A, B>{
    void print();
}
public class MyClass<A, B> implements MyInterface<A, B>{
    int num = 0;
    void print(){
        System.out.println("Hi");
    }
    public class Node{
        int num = 0;
    }
    public static void main(String[] args){
        MyInterface<A, B> obj = new MyClass<A, B>(); //will this work?
        MyInterface<int, String> obj = new MyClass<int, String>(); //will this work?
        MyInterface<String, String> obj = new MyClass<>(); //will this work?
        MyInterface<> obj = new MyClass<Node, Node>(); //will this work?
    }
}
```

Instantiation

Given the following code, which of the instantiations in `main` are valid?

```
public interface MyInterface<A, B>{
    void print();
}
public class MyClass<A, B> implements MyInterface<A, B>{
    int num = 0;
    void print(){
        System.out.println("Hi");
    }
    public class Node{
        int num = 0;
    }
    public static void main(String[] args){
        MyInterface<A, B> obj = new MyClass<A, B>();
        MyInterface<int, String> obj = new MyClass<int, String>();
        MyInterface<String, String> obj = new MyClass<>(); //only this one!
        MyInterface<> obj = new MyClass<Node, Node>();
    }
}
```

FILL IN THE BLANK

Multiple classes may be nearly identical, differing only in their data types they contain…

Generics allow us to implement classes without limiting the _______ that we can store in the class

<u>RULE TO FOLLOW:</u> outermost type should be a _______, it needs to be

instantiated. Inside of angled brackets use _______.

FILL IN THE BLANK

Multiple classes may be nearly identical, differing only in their data types they contain…

Generics allow us to implement classes without limiting the **data type** that we can store in the class

RULE TO FOLLOW: outermost type should be a **class**, it needs to be

instantiated. Inside of angled brackets use **interface**.

What is wrong with the below snippet of code? (Assume you have access to all variables used in the method)

```
private void expandCapacity() {
    E[] expanded = (E[])(new Object[this.size * 2]);
}
```

What is wrong with the below snippet of code? (Assume you have access to all variables used in the method)

```
@SuppressWarnings(“unchecked”)
private void expandCapacity() {
    E[] expanded = (E[])(new Object[this.size * 2]);
}
```

Data Structures

Array Lists (AL)

FILL IN THE BLANK

________ represents how many elements have been added

________ represents how much space we have in an array

To add String s into an AList, add String s at the specified ________ in the AList and shift all subsequent elements to the ________ by ________ index

To remove a String s at the specified index in the AList, remove String s at the specified ________ and shift all subsequent elements to the ________ by ________ index

FILL IN THE BLANK

Size represents how many elements have been added

Capacity represents how much space we have in an array

To add String s into an AList, add String s at the specified **index** in the AList and shift all subsequent elements to the **right** by **one** index

To remove a String s at the specified index in the AList, remove String s at the specified **index** and shift all subsequent elements to the **left** by **one** index

True or False

In a class, can a method and field have the same name.

True or False

In a class, can a method and field have the same name. **(TRUE)**

FILL IN THE BLANK

If there is an ArrayList with 3 elements in it and its capacity starts at 3 and its capacity increases by a factor of 3 during each resize

The capacity after inserting 7 elements will be _______

The capacity after inserting 9 elements will be _______

The capacity after inserting 12 elements will be _______

FILL IN THE BLANK

If there is an ArrayList with 3 elements in it and its capacity starts at 3 and its capacity increases by a factor of 3 during each resize

The capacity after inserting 7 elements will be **27**

The capacity after inserting 9 elements will be **27**

The capacity after inserting 12 elements will be **27**

Summary: ArrayList (AL) **(Unsorted, using SLL or Array)**

Method	Worst Case	Best Case
find	O(__________)	O(__________)
insert	O(__________)	O(__________)
remove	O(__________)	O(__________)

Summary: ArrayList (AL) **(Unsorted, using SLL or Array)**

Method	Worst Case	Best Case
find	O(**n**)	O(**1**)
insert	O(**n**)	O(**1**)
remove	O(**n**),	O(**1**)

Summary: ArrayList (AL) (**Sorted, using Array**)

Method	Worst Case	Best Case
find	O(__________)	O(__________)
insert	O(__________)	O(__________)
remove	O(__________)	O(__________)

Summary: ArrayList (AL) (**Sorted, using Array**)

Method	Worst Case	Best Case
find	O(**logn**) - perform binary search	O(**1**)
insert	O(**n**)	O(**1**)
remove	O(**n**)	O(**1**)

Circular Array Lists

Summary: CircularArrayList (AL) **Unsorted**

Method	Worst Case	Best Case
find	O(________)	O(________)
insert	O(________)	O(________)
remove	O(________)	O(________)
get	O(________)	O(________)
add	O(________)	O(________)
prepend	O(________)	O(________)

Summary: CircularArrayList (AL) **Unsorted**

Method	Worst Case	Best Case
find	O(**n**)	O(**1**)
insert	O(**n**)	O(**1**)
remove	O(**n**)	O(**1**)
get	O(**1**)	O(**1**)
add	O(**n**)	O(**1**)
prepend	O(**n**)	O(**1**)

Singly Linked Lists

FILL IN THE BLANK

The method add() adds an element to the ________ of the list

The method prepend() adds an element to the ________ of the list

FILL IN THE BLANK

The method add() adds an element to the **end** of the list

The method prepend() adds an element to the **front** of the list

Runtime of operations on lists

Worst case analysis of operations on ArrayList and SinglyLinkedList

	ArrayList	SinglyLinkedList
get (an index)	O(________)	O(________)
find (a value)	O(________)	O(________)
insert at beg / prepend	O(________)	O(________)
insert at end / append	O(________)	O(________)
delete	O(________)	O(________)

Runtime of operations on lists

Worst case analysis of operations on ArrayList and SinglyLinkedList

	ArrayList	SinglyLinkedList
get (an index)	O(**1**)	O(**n**)
find (a value)	O(**n**)	O(**n**)
insert at beg / prepend	O(**n**)	O(**1**)
insert at end / append	O(**n**)	O(**n**)
delete	O(**n**)	O(**n**)

Runtime of operations on lists

Best case analysis of operations on ArrayList and SinglyLinkedList

	ArrayList	SinglyLinkedList
get / find	O(__________)	O(__________)
insert at beg / prepend	O(__________)	O(__________)
insert at end / append	O(__________)	O(__________)
delete	O(__________)	O(__________)

Runtime of operations on lists

Best case analysis of operations on ArrayList and SinglyLinkedList

	ArrayList	SinglyLinkedList
get / find	O(**1**)	O(**1**)
insert at beg / prepend	O(**n**)	O(**1**)
insert at end / append	O(**1**)	O(**n**)
delete	O(**1**)	O(**1**)

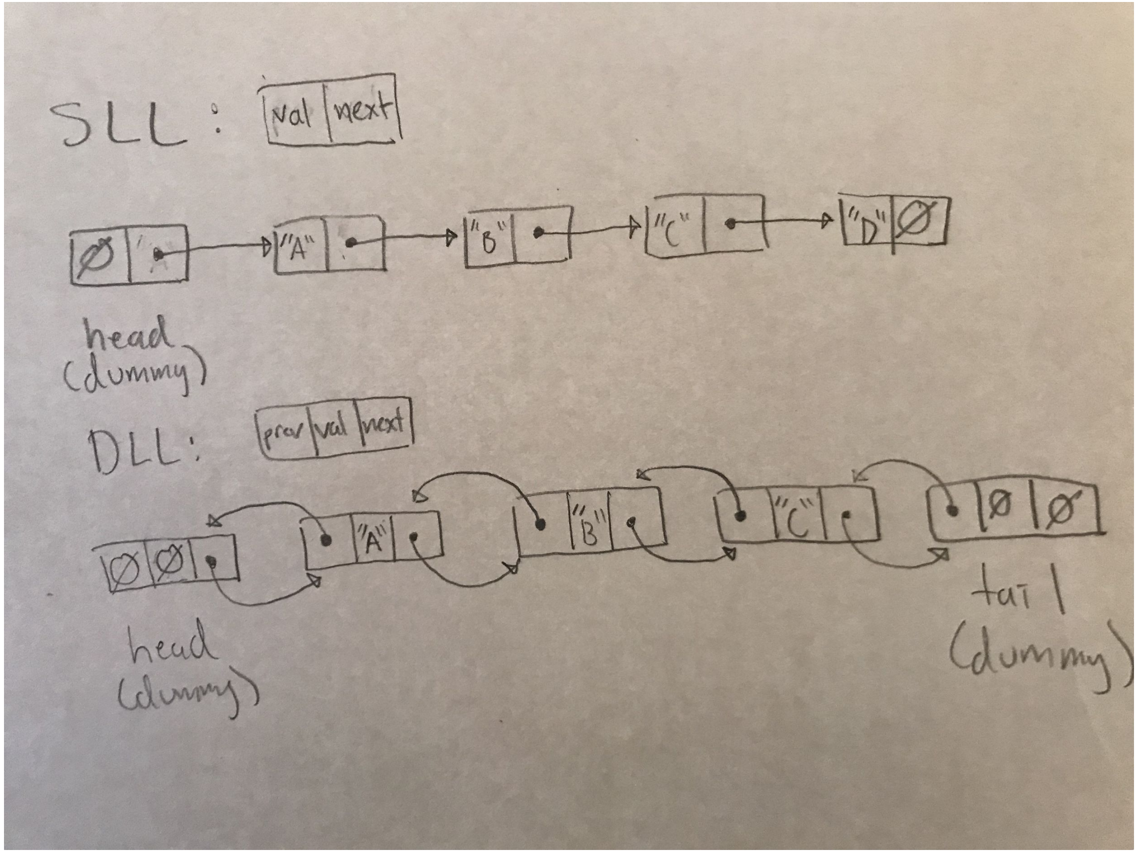
Doubly Linked Lists

What is the difference between a doubly and singly LinkedList?

What is the difference between a doubly and singly LinkedList?

Doubly LinkedLists have the fields head AND tail. Each node contains next AND prev.

Singly LinkedLists only contain the field head and each nodes ONLY contain next.



Summary: Doubly Linked List

	Best Case	Worst Case
get / find	O(__________)	O(__________)
prepend	O(__________)	O(__________)
append	O(__________)	O(__________)
delete (a given node)	O(__________)	O(__________)

Summary: Doubly Linked List

	Best Case	Worst Case
get / find	O(**1**)	O(**n**)
prepend	O(**1**)	O(**1**)
append	O(**1**)	O(**1**)
delete (a given **node**)	O(**1**)	O(**1**)
delete (a given **index or value**)	O(**1**)	O(**n**)

Stacks & Queues

FILL IN THE BLANK

_______ have the property LIFO (last in, first out)

_______ have the property FIFO (first in, first out)

Basic methods for a stack include: _______, _______

Basic methods for a queue include: _______, _______

FILL IN THE BLANK

Stacks have the property LIFO (last in, first out)

Queues have the property FIFO (first in, first out)

Basic methods for a stack include: **push**, **pop**

Basic methods for a queue include: **enqueue**, **dequeue**

Summary: Stack (using a singly linked list)

	Best Case	Worst Case
push	O(__________)	O(__________)
pop	O(__________)	O(__________)
peek	O(__________)	O(__________)

Summary: Stack (using a singly linked list)

	Best Case	Worst Case
push	O(**1**)	O(**1**)
pop	O(**1**)	O(**1**)
peek	O(**1**)	**O(1)**

Summary: Queue (using a doubly linked list)

	Best Case	Worst Case
enqueue	O(__________)	O(__________)
dequeue	O(__________)	O(__________)

Summary: Queue (using a doubly linked list)

	Best Case	Worst Case
enqueue	O(1)	O(1)
dequeue	O(1)	O(1)

Maps & HashTables

FILL IN THE BLANK

________ is a data structure which implements an array-like data type to map **keys to values**.

Uses a ________ to compute an *index* into an array of *buckets*, where the desired key-value pair can be inserted and found.

A hash function assigns an ________ **value** to a ________ **key**.

FILL IN THE BLANK

hash table is a data structure which implements an array-like data type to map **keys to values**.

Uses a **hash function** to compute an *index* into an array of *buckets*, where the desired key-value pair can be inserted and found.

A hash function assigns an **integer** **value** to a **specific** **key**.

Is this a good hash function?

```
return s.length();
```

Is this a good hash function?

```
return s.length();
```

- All strings of the same length will get mapped to the same bucket if the length of the string is < array size. So this is **NOT** a good hash function.
- A good hash function is imperative to maintain efficiency in a hash table.

FILL IN THE BLANK

What is a good hash function?

- Hash value is ________, i.e. it is determined fully by the data that is being hashed.
- Uses ________ the input data (as much information as possible)
- ________ distributes data across the set of hash values. (uniqueness of hash values).
- **Note:** ________ modulo the hash code with array size to get index.

FILL IN THE BLANK

What is a good hash function?

- Hash value is **deterministic**, i.e. it is determined fully by the data that is being hashed.
- Uses **all** the input data (as much information as possible)
- **Uniformly** distributes data across the set of hash values. (uniqueness of hash values).
- **Note: Always** modulo the hash code with array size to get index.

Load threshold & Expansion factor

- Once the ratio: **size (number of elements inserted) / array.size** of a hash table exceeds or is equal to a certain *load threshold*, the array of buckets should expand by the expansion factor.

- This is done in order to maintain efficiency of the hash table and expand range of potential hash values when table starts filling up. (To avoid collisions)

Rehashing

- Rehash all existing elements in hashtable once the load threshold is reached and the array is expanded.
- It is important to **regenerate the index of the bucket of existing keys in the hashtable**.
- Same keys can be potentially mapped to different buckets due to change in array size.

What are collisions?

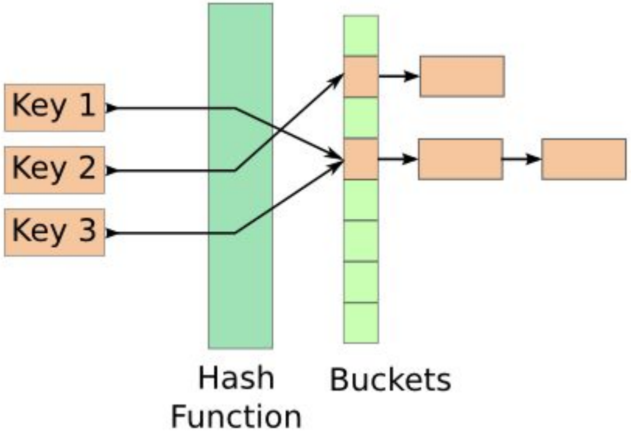
A collision occurs when a new key is inserted into a **non-empty bucket**.

Common misconception:

“Updating a key-value pair is a collision”. It is **NOT**!
That is, if a bucket contains a key-value pair and the value of a certain key is replaced via insertion, this is **NOT A COLLISION**.

Separate Chaining

- Each bucket stores a list of key-value pairs
- Multiply keys mapped to the same bucket is maintained by this list



FILL IN THE BLANK

A bucket is a collection of _______ at a specific _______

The load factor is # _______ / # _______

FILL IN THE BLANK

A bucket is a collection of **elements** at a specific **index**

The load factor is # **elements** / # **buckets**

FILL IN THE BLANK

Assign a ________ to each ________ we are trying to keep track of.

Index for entry is determined by a ________ function that calculates index using key value (USEFUL FOR QUICK ________ AND ________)

Contains methods ________(Object key), ________(K key, V value), ________(), ________(K key, V value) etc.

Keys need to be ________

put() will essentially do what ________() does, however, ________() will not place a new entry as put() does

FILL IN THE BLANK

Assign a **key** to each **value** we are trying to keep track of.

Index for entry is determined by a **hash** function that calculates index using key value (USEFUL FOR QUICK **LOOKUP** AND **INSERT**)

Contains methods **get**(Object key), **put**(K key, V value), **size**(), **replace**(K key, V value) etc.

Keys need to be **unique**

put() will essentially do what **replace**() does, however, **replace**() will not place a new entry as put() does

Runtime of operations on Maps

	Maps (best)	Maps (worst)
get / find	O(_____)	O(_____)
insert (prepend)	O(_____)	O(_____)
insert (append)	O(_____)	O(_____)
delete	O(_____)	O(_____)

Runtime of operations on Maps

	Maps (best)	Maps (worst)
get / find	O(**1**)	O(**n**)
insert (prepend)	O(**1**)	O(**n**)
insert (append)	O(**1**)	O(**n**)
delete	O(**1**)	O(**n**)

HashTables

Method	Worst Case	Best Case
find	O(____)	O(____)
insert	O(____)	O(____)
remove	O(____)	O(____)

HashTables

Method	Worst Case	Best Case
find	O(**n**)	O(**1**)
insert	O(**n**)	O(**1**)
remove	O(**n**)	O(**1**)

Binary Search Trees

https://www.cs.usfca.edu/~galles/visualization/BST.html

FILL IN THE BLANK

If a **new** element is **added** (with a unique key) to a **BST** it will always be added at a _______ most node that has _______ as a left or right child

The **height** of a tree is the number of nodes on the _______ path from the root to the _______ (or to a _______).

The **best** case height for a tree with n elements is _______. The **worst** case height for a tree with n elements is _______

FILL IN THE BLANK

If a **new** element is **added** (with a unique key) to a **BST** it will always be added at a **bottom** most node that has **null** as a left or right child

The **height** of a tree is the number of nodes on the **longest** path from the root to the **bottom** (or to a **leaf**).

The **best** case height for a tree with n elements is **lgn**. The **worst** case height for a tree with n elements is **n**

BSTs

Method	Worst Case	Best Case
find	O(_____)	O(_____)
insert	O(_____)	O(_____)
remove	O(_____)	O(_____)

BSTs

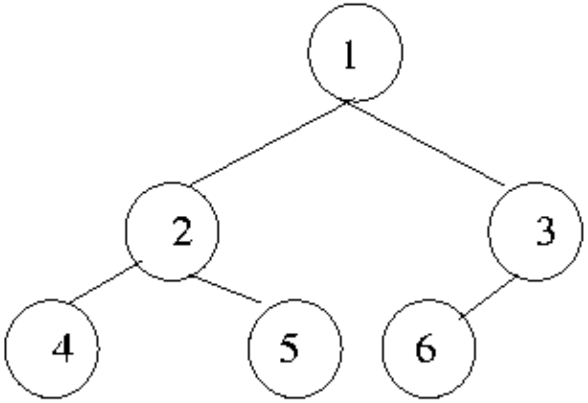
Method	Worst Case	Best Case
find	O(**n**)	O(**1**)
insert	O(**n**)	O(**1**)
remove	O(**n**)	O(**1**)

FILL IN THE BLANK:

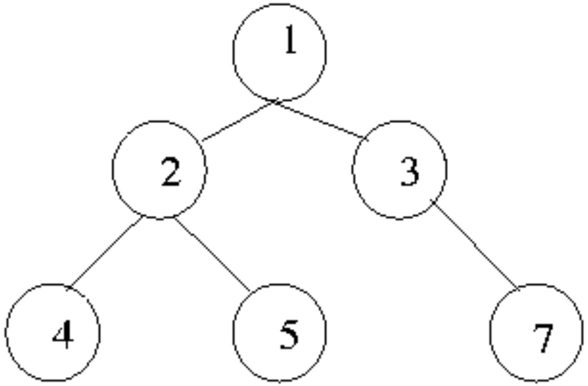
Definition: A tree is a ________ if every level but the last level is completely full, and the last level is filled starting from the leftmost node.

FILL IN THE BLANK:

Definition: A tree is a **complete tree** if every level but the last level is completely full, and the last level is filled starting from the leftmost node.



Complete Binary Tree



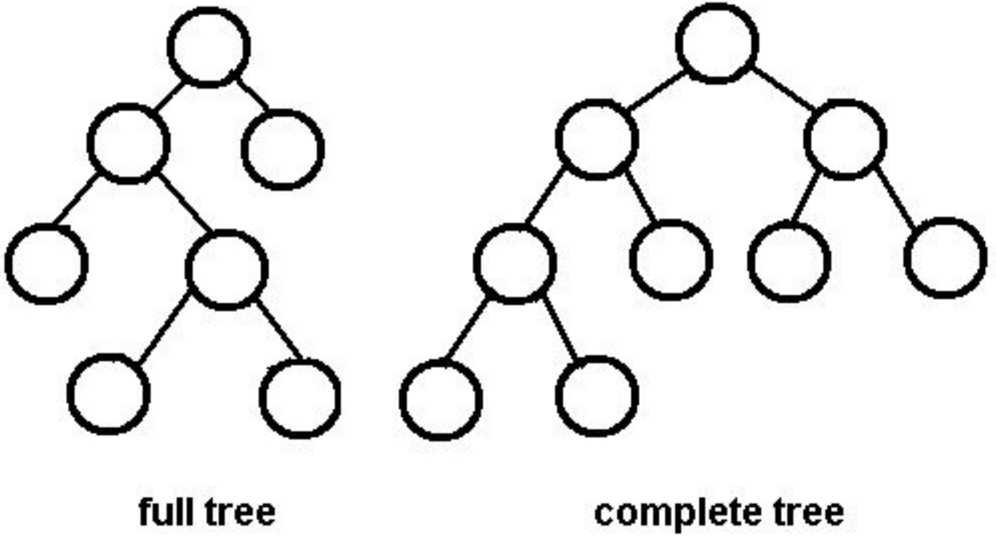
Not Complete

FILL IN THE BLANK:

Definition: A tree is a ________ if every node other than the leaves has two children

FILL IN THE BLANK:

Definition: A tree is a **full tree** if every node other than the leaves has two children

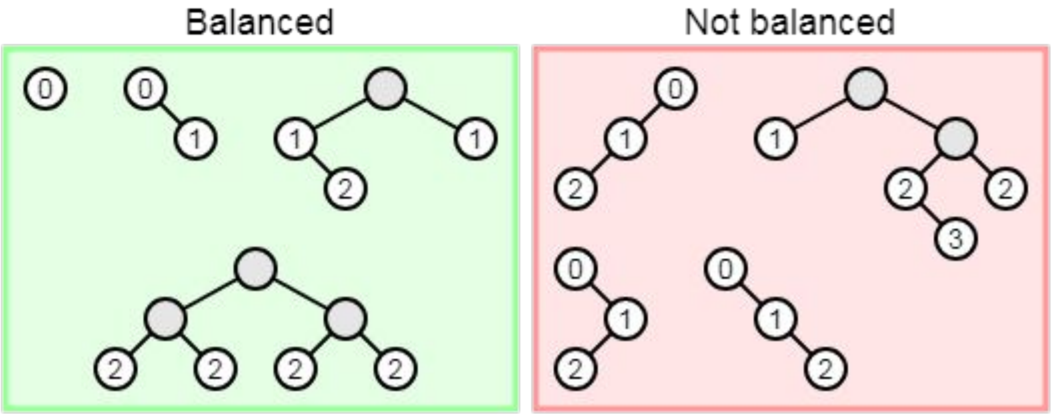


FILL IN THE BLANK:

Definition: A tree is a ________ if for each node, the left and right subtrees have levels that differ in height by at most 1

FILL IN THE BLANK:

Definition: A tree is a **height-balanced tree** if for each node, the left and right subtrees have levels that differ in height by at most 1



Heaps

https://www.cs.usfca.edu/~galles/visualization/Heap.html

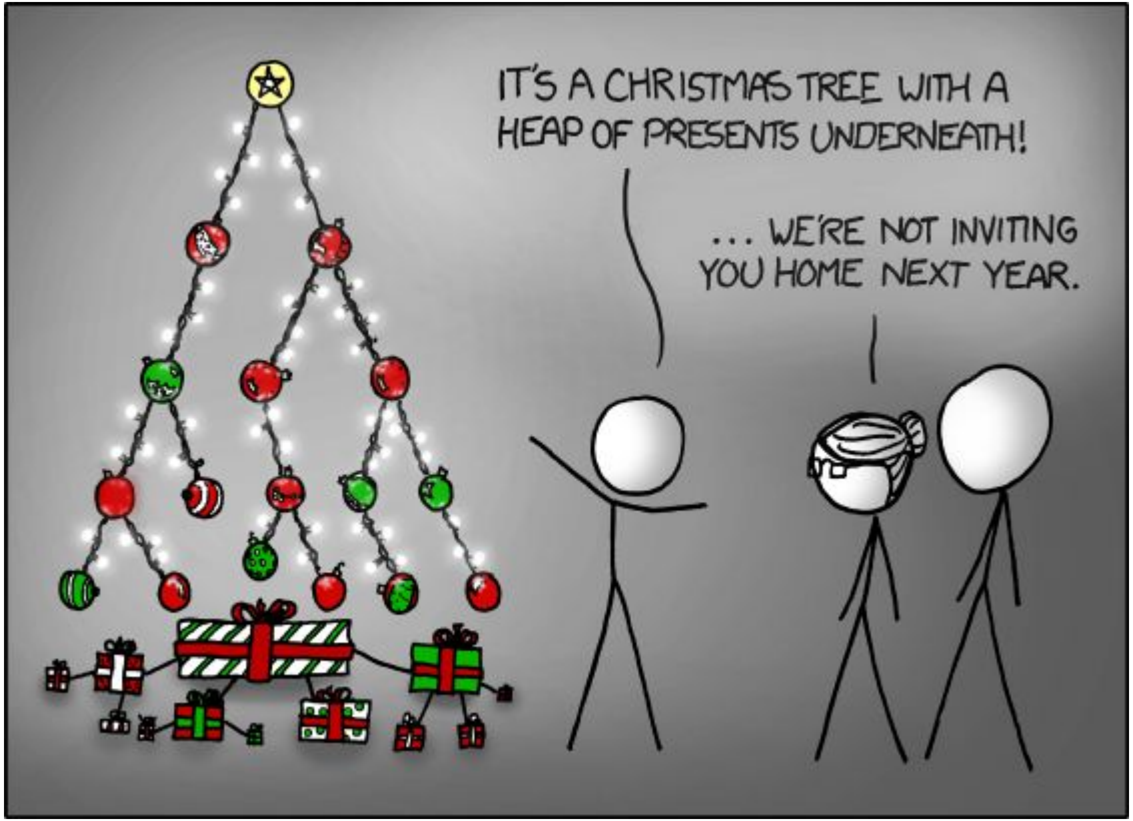
HEAPS!!!

Min-Heap:
The value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.

Main operations:
removeMin, add

Important subroutines:
Bubble Up and Bubble Down

(Is actually a max heap)


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

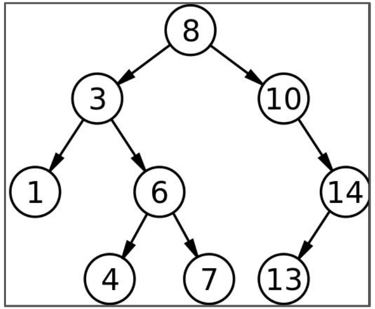
How is it different from a BST?

Min Heap:

- Binary Tree with each parent’s value < its 2 children
- Builds left to right
- Max difference in height between any 2 leafs nodes = 1

BST:

- Binary Tree with parent holding references to <=2 children
- Follows an ordering among parents and children. Usually, to get a sorted result, you go
  - Left child -> Parent -> Right child
- This means that Left child < Parent < Right Child

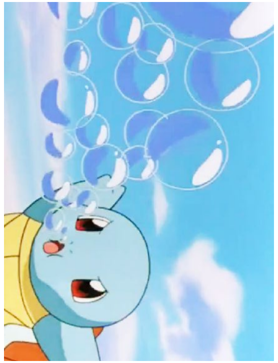


Are there other type of orderings possible….? (spoiler: yes)

Bubble Down

- Occurs when removing an element from a Heap.
- Save root value and copy last node into root (also decrease size)
- Starting from root, recursively swap current node with the **smaller** of its children.
- End recursion when leaf node is reached or children are larger than the parent.

Bubble Up



- Occurs when adding an element to a Heap.
- Recursively compare element with its parent and swap them if child < parent
- End recursion when child >= parent, or when current node is root.

```
void bubbleUp(Heap* h, int index) {
  if(index <= 0) { return; }
  int parentIndex = (index - 1) / 2;
  if(h->elements[parentIndex].key <
h->elements[index].key) { return; }
  swap(h, parentIndex, index);
  bubbleUp(h, parentIndex);
}
```

What’s the point?

Worst case	Add	removeMin
Array	O(n)	O(n)
Linked List	O(n)	**O(n)**
Heap	O(log n)	O(log n)

Pros:

- Implement other data structures (e.g. Priority Queues) faster
- Is pretty cool
- Can be used to implement Heapsort

FILL IN THE BLANK

Definition: A tree is a ______ if every level but the last level is completely full, and the last level is filled starting from the leftmost node.

Property: A complete tree's size and height are related by:
______ ~ ______

Definition: A tree is in ______ (______) ______ **order** if every node's key is **greater** (less) than or equal to all of its childrens' keys.

Definition: A **max** (min) **heap** is a ______ that is in **max** (min) **heap order**.

FILL IN THE BLANK

Definition: A tree is a **complete tree** if every level but the last level is completely full, and the last level is filled starting from the leftmost node.

Property: A complete tree's size and height are related by:
height ~ ***log(size)***

Definition: A tree is in **max** (min) **heap** **order** if every node's key is **greater** (less) than or equal to all of its childrens' keys.

Definition: A **max** (min) **heap** is a **complete tree** that is in **max** (min) **heap order**.

FILL IN THE BLANK

Because we are using a complete tree `poll()` will be directly related to:

Θ(_______) = Θ(_______)

Because we are using a complete tree `add(?)` will be directly related to:

Θ(_______) = Θ(_______)

FILL IN THE BLANK

Because we are using a complete tree `poll()` will be directly related to:

Θ(**height**) = Θ(**lg(size)**)

Because we are using a complete tree `add(?)` will be directly related to:

Θ(**height**) = Θ(**lg(size)**)

FILL IN THE BLANK

In a heap, the index of a parent’s left child is __________ and the index of a parent’s right child is __________

FILL IN THE BLANK

In a heap, the index of a parent’s left child is **index*2 + 1** and the index of a parent’s right child is **index*2 + 2**

Summary of ADT Method Signatures

List <E> {
void add(E e);
E get(int index);
}
	Queue <E> {
	void enqueue(E e);
	E dequeue();
	}
		Stack<E> {
		void push(E e);
		E pop();
		}
			Map<K,V> {
			void set(K k, V v);
			V get(K k);
			}
				PriorityQueue<K,V> {
				void set(K k, V v);
				V poll();
				}

Heaps

Method	Worst Case	Best Case
peak	O(_____)	O(_____)
insert	O(_____)	O(_____)
pop	O(_____)	O(_____)

Heaps

Method	Worst Case	Best Case
peak	O(**1**)	O(**1**)
insert	O(**logn**)	O(**1**)
pop (remove root)	O(**logn**)	O(**1**)

What’s the difference between a heap and a priority queue?

A **heap** is a **data structure** that stores things in a particular manner

A **priority queue** is an **abstract data type** that can be implemented using different data structures, one of the options being a heap

Adapter pattern! This is like how a linked list is a data structure that can be used to implement a queue, which is an abstract data type

Time Complexity

Runtime

Asymptotic Runtime Complexity

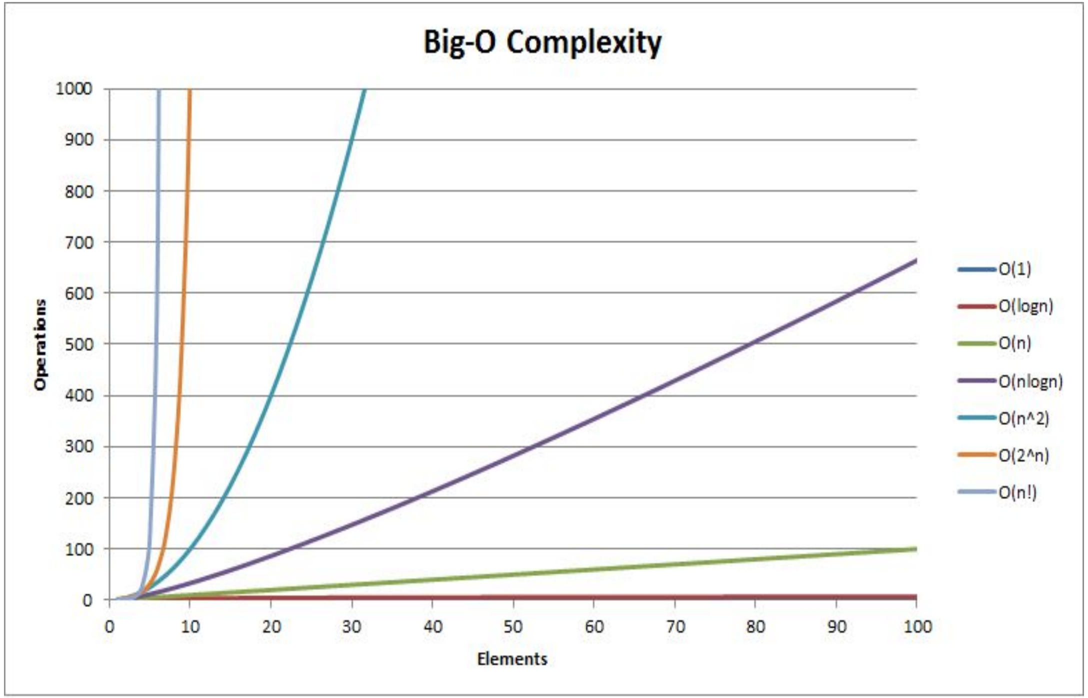
Asymptotic Runtime?

Asymptotic: becoming increasingly exact as a variable approaches a limit, usually infinity. (the n in our estimates!)

Why do we care about Asymptotic analysis?

Real world application… Data analysis, Artificial Intelligence… etc.

Big-O review



- Relative to input n
- Constants do not matter
  - O(3n)=O(n)
- Higher order values dominate
  - O(n^2 + n) = O(n^2)

True/False

- $n + 5n^3 + 8n^4 = O(n)$
- $n! + n^2 = O(n\log n)$
- $2^n + n\log n = O(n!)$

True/False

- $n + 5n^3+8n^4 = O(n)$ **False**
- $n! + n^2 = O(n\log n)$ **False**
- $2^n + n\log n = O(n!)$ **True**

True/False

- $1/n^2 + 5 = O(1/n)$
- $\log n + n\log n + \log(\log n) = \Omega(\log n)$
- $2^n + n! = \Omega(n^n)$

True/False

- $1/n^2 + 5 = O(1/n)$ **False**
- $\log n + n\log n + \log(\log n) = \Omega(\log n)$ **True**
- $2^n + n! = \Omega(n^n)$ **False**

True/False

- $n^2 + n + \text{sqrt}(n) = \Omega(\log n)$
- $n^2 + n/4 + 6 = \Theta(n^3)$
- $2^n + n = \Theta(n)$
- $1/n^{50} + \log 32 = \Theta(1)$

True/False

- $n^2 + n + \mathrm{sqrt}(n) = \Omega(\log n)$ **True**
- $n^2 + n/4 + 6 = \Theta(n^3)$ **False**
- $2^n + n = \Theta(n)$ **False**
- $1/n^{50} + \log 32 = \Theta(1)$ **True**

What is big-O of the below code?

```
int num = 0;
for (int i = 0; i <= n*n; i = i+2) {
    num = num + 2;
}
```

What is big-O of the below code?

```
int num = 0;
for (int i = 0; i <= n*n; i = i+2) {
    num = num + 2;
}
```

ANSWER:

$O(n^2)$

What is big-O of the below code?

```
int num = 1;
for (int i = 0; i < n; i = i+1) {
    for (int j = 0; j <= i; j = j+1) {
        num = num * 2;
    }
}
```

What is big-O of the below code?

```
int num = 1;
for (int i = 0; i < n; i = i+1) {
    for (int j = 0; j <= i; j = j+1) {
        num = num * 2;
    }
}
```

ANSWER:
$O(n^2)$

What is big-O of the below code?

```
int num = 0;
for (int i = n*n; i > 0; i = i/2) {
    for (int j = 1; j <= n; j = j+1) {
        num = num + i;
    }
}
```

What is big-O of the below code?

```
int num = 0;
for (int i = n*n; i > 0; i = i/2) {
    for (int j = 1; j <= n; j = j+1) {
        num = num + i;
    }
}
```

ANSWER:
O(nlogn)

O(nlogn^2) ~ O(nlogn) due to log properties

What is big-O of the below code?

```
int num = 0;
for (int i = 0; i <= n*n; i = i+1) {
    num = num + 1;
}
for (int i = 1; i <= n; i = i*2) {
    for (int j = n; j >= 1; j = j/2) {
        num = num + i;
    }
}
```

What is big-O of the below code?

```
int num = 0;
for (int i = 0; i <= n*n; i = i+1) {
    num = num + 1;
}
for (int i = 1; i <= n; i = i*2) {
    for (int j = n; j >= 1; j = j/2) {
        num = num + i;
    }
}
```

ANSWER:
$O(n^2)$

What is big-O of the below code?

```
void insert(int x, int n, int[] arr) {
  for(int i = 0; i < n; i += 1) {
    if(x < arr[i]) {
      for(int j = n; j > i; j -= 1) {
        arr[j] = arr[j - 1];
      }
      arr[i] = x;
      return;
    }
  }
  arr[n] = x;
}
public void isort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    insert(arr[i], i, arr)
  }
}
```

Best insert: ?
Worst insert: ?
Worst isort: ?

What is big-O of the below code?

```
void insert(int x, int n, int[] arr) {
  for(int i = 0; i < n; i += 1) {
    if(x < arr[i]) {
      for(int j = n; j > i; j -= 1) {
        arr[j] = arr[j - 1];
      }
      arr[i] = x;
      return;
    }
  }
  arr[n] = x;
}
public void isort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    insert(arr[i], i, arr)
  }
}
```

Best insert: **O(n)**
Worst insert: **O(n)**
Worst isort: **O(n^2)**

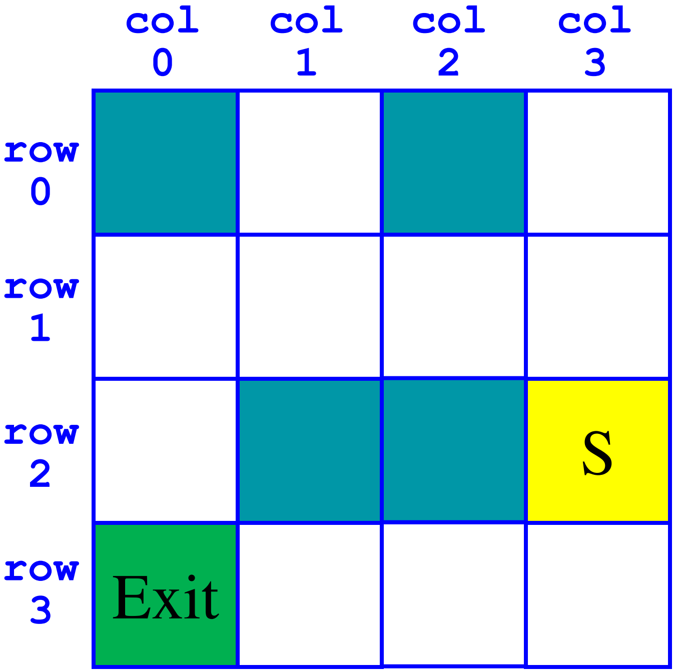
Taken from midterm #2!

Algorithms

BFS & DFS

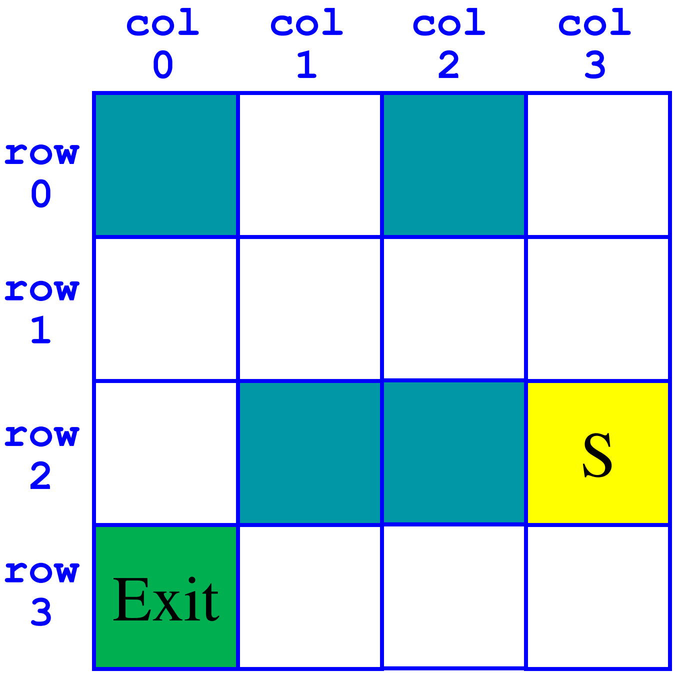
https://visualgo.net/en/dfsbfs?slide=1

Trapped in a Maze!



	col 0	col 1	col 2	col 3
row 0	▨		▨	
row 1				
row 2		▨	▨	S
row 3	Exit			

Searching for the exit



```
SearchForTheExit
•   Initialize a task list to hold Squares as we search
•   Mark starting square as visited
•   Put starting square on task list

•   While task list is not empty
        •   Remove square sq from task list
        •   Mark sq as visited
        •   If sq is the Exit, we're done!
        •   For each of square's unseen neighbors (N, S, E, W):
                •   Set neighbor's previous to sq
                •   Add neighbor to task list
```

What will be the output?

Consider doing the following operations on an initially empty stack, s:

s.push(5)

s.push(10)

s.push(11)

s.pop()

s.push(5)

What are the contents of the stack, from top (left) to bottom (right):

A) 5, 10, 11, 5

B) 5, 13, 10, 5

C) 5, 10, 5

D) 5, 11, 10

E) other

What will be the output?

Consider doing the following operations on an initially empty stack, s:

s.push(5)

s.push(10)

s.push(11)

s.pop()

s.push(5)

What are the contents of the stack, from top (left) to bottom (right):

- A) 5, 10, 11, 5
- B) 5, 11, 10, 5
- **C) 5, 10, 5**
- D) 5, 11, 10
- E) other

What will be the output?

How many stacks are needed to implement a queue? Consider the situation where no other data structure like arrays, linked list is available to you.

A) 1
B) 2
C) 3
D) 4

What will be the output?

How many stacks are needed to implement a queue? Consider the situation where no other data structure like arrays, linked list is available to you.

A) 1
B) 2
C) 3
D) 4

What will be the output?

Consider doing the following operations on an initially empty queue, q:

s.enqueue(4)

s.enqueue(10)

s.enqueue(13)

s.dequeue()

s.enqueue(5)

What are the contents of the queue, from top (left) to bottom (right):

A) 4, 10, 13, 5

B) 10, 13, 5

C) 4, 10, 5

D) 5, 10, 4

E) other

What will be the output?

Consider doing the following operations on an initially empty queue, q:

s.enqueue(4)

s.enqueue(10)

s.enqueue(13)

s.dequeue()

s.enqueue(5)

What are the contents of the queue, from top (left) to bottom (right):

A) 4, 10, 13, 5

B) 10, 13, 5

C) 4, 10, 5

D) 5, 10, 4

E) other

- **Breadth-first search** (**BFS**) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root and explores the neighbor nodes first, before moving to the next level neighbors.

- BFS can be implemented by using a queue.

- **Depth-first search (DFS)** is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

- DFS can be implemented by using a stack.

Say a reference to a Maze object is stored in a variable m. What expression would get the Square at row 3, column 5? * Mark only one

A. m.getRow(3).getCol(5)

B. m.contents[5][3]

C. m.contents[3][5]

D. m.getCol(5).getRow(3)

Say a reference to a Maze object is stored in a variable m. What expression would get the Square at row 3, column 5? * Mark only one

A. m.getRow(3).getCol(5)

B. m.contents[5][3]

C. m.contents[3][5]

D. m.getCol(5).getRow(3)

DFS, BFS, Stacks, and Queues

For more detailed examples of using stacks and queues to perform DFS and BFS, refer to the **Week 3 Review** linked in the course schedule on the course website. We perform both and show how Squares and the stack/queue is updated as the algorithm proceeds

Sorting

FILL IN THE BLANK

Selection Sort: Repeatedly find the ________ element and move it to the ________ of a ________ **prefix** of the array.

Insertion Sort: Repeatedly take the next element and insert it into the **correct ________ position within** a ________ **prefix** of the array.

FILL IN THE BLANK

Selection Sort: Repeatedly find the **minimum** element and move it to the **end** of a **sorted** **prefix** of the array.

Insertion Sort: Repeatedly take the next element and insert it into the **correct** **ordered** **position within** a **sorted** **prefix** of the array.

Given the array below after a call to partition, what indices have elements that could have been the partition (Assume partition chose a random element)

8	2	1	9	10	18	20	16	30
0	1	2	3	4	5	6	7	8

Given the array below after a call to partition, what indices have elements that could have been the partition (Assume partition chose a random element)

8	2	1	9	10	18	20	16	30
0	1	2	3	4	5	6	7	8

Indices: 3, 4, 8

Given the array below after a call to partition, what indices have elements that could have been the partition (Assume partition chose a random element)

1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8

Given the array below after a call to partition, what indices have elements that could have been the partition (Assume partition chose a random element)

1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8

ALL OF THEM!

Given the array below after a call to partition, what indices have elements that could have been the partition (Assume partition chose a random element)

1	10	9	8	7	6	5	4	3
0	1	2	3	4	5	6	7	8

Given the array below after a call to partition, what indices have elements that could have been the partition (Assume partition chose a random element)

1	10	9	8	7	6	5	4	3
0	1	2	3	4	5	6	7	8

Index 0 ONLY

Match the Sorting Algorithm!

```
static int[] combine(int[] p1, int[] p2) {...}

static int[] sort(int[] arr) {
  int len = arr.length
  if(len <= 1) { return arr; }
  else {
    int[] p1 = Arrays.copyOfRange(arr, 0, len / 2);
    int[] p2= Arrays.copyOfRange(arr, len / 2, len);
    int[] sortedPart1 = sort(p1);
    int[] sortedPart2 = sort(p2);
    int[] sorted = combine(sortedPart1, sortedPart2);
    return sorted;
  }
}
```

```
static void sort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
}
```

```
static void sort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
      else { break; }
    }
  }
}
```

```
static int partition(String[] array, int l, int h) {...}

static void sort2(String[] array, int low, int high) {
  if(high - low <= 1) { return; }
  int splitAt = partition(array, low, high);
  sort2(array, low, splitAt);
  sort2(array, splitAt + 1, high);
}

public static void sort1(String[] array) {
  sort2(array, 0, array.length);
}
```

Match the Sorting Algorithm!

```java
static int[] combine(int[] p1, int[] p2) {...}

static int[] mergeSort(int[] arr) {
  int len = arr.length
  if(len <= 1) { return arr; }
  else {
    int[] p1 = Arrays.copyOfRange(arr, 0, len / 2);
    int[] p2= Arrays.copyOfRange(arr, len / 2, len);
    int[] sortedPart1 = sort(p1);
    int[] sortedPart2 = sort(p2);
    int[] sorted = combine(sortedPart1, sortedPart2);
    return sorted;
  }
}
```

Merge

```java
static void selectionSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
}
```

Selection

```java
static void insertionSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
      else { break; }
    }
  }
}
```

Insertion

```java
static int partition(String[] array, int l, int h) {...}

static void quickSort(String[] array, int low, int high) {
  if(high - low <= 1) { return; }
  int splitAt = partition(array, low, high);
  quickSort(array, low, splitAt);
  quickSort(array, splitAt + 1, high);
}

public static void sort(String[] array) {
  sort2(array, 0, array.length);
}
```

Quick

FILL IN THE BLANK

_________ (algorithm) finds the minimum element in a list and moves it to the end of a sorted prefix in the list.

_________ (algorithm) repeatedly takes the next element in a list inserts it into the correct ordered position within a sorted prefix of the list.

FILL IN THE BLANK

Selection sort (algorithm) finds the minimum element in a list and moves it to the end of a sorted prefix in the list.

Insertion sort (algorithm) repeatedly takes the next element in a list inserts it into the correct ordered position within a sorted prefix of the list.

FILL IN THE BLANK

_________ (algorithm) repeatedly splits a list of elements at a certain index, where all elements to the left are <= and all elements to the right are >=. Then, the single elements at the end are already in order to be recombined.

_________ (algorithm) repeatedly splits a list of elements in half until we have single elements. Then, the single elements at the end are compared by value and recombined to put them in order.

FILL IN THE BLANK

Quick sort (algorithm) repeatedly splits a list of elements at a certain index, where all elements to the left are <= and all elements to the right are >=. Then, the single elements at the end are already in order to be recombined.

Merge sort (algorithm) repeatedly splits a list of elements in half until we have single elements. Then, the single elements at the end are compared by value and recombined to put them in order.

FILL IN THE BLANK

	Insertion	**Selection**	**Merge**	**Quick**
Best case time	O(____)	O(____)	O(____)	O(____)
Worst case time	O(____)	O(____)	O(____)	O(____)

Key operations	`swap(a, j, j-1)`
	(until in the right place)
		`swap(a, i, indexOfMin)`
		(after finding minimum value)
			`l = copy(a, 0, len/2)`
			`r = copy(a, len/2, len)`
			`ls = sort(l)`
			`rs = sort(r)`
			`merge(ls, rs)`
				`p = partition(a, l, h)`
				`sort(a, l, p)`
				`sort(a, p + 1, h)`

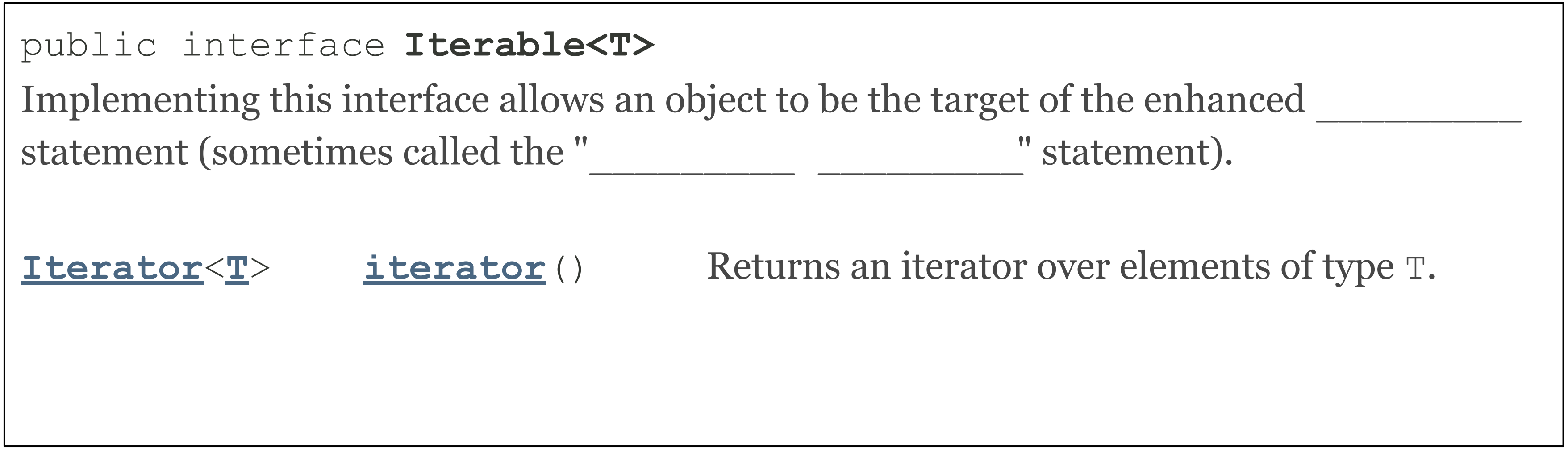
FILL IN THE BLANK

	Insertion	**Selection**	**Merge**	**Quick**
Best case time	O(**n**)	O(**n^2**)	O(**n*logn**)	O(**n*logn**)
Worst case time	O(**n^2**)	O(**n^2**)	O(**n*logn**)	O(**n^2**)

Key operations	`swap(a, j, j-1)`
	(until in the right place)
		`swap(a, i, indexOfMin)`
		(after finding minimum value)
			`l = copy(a, 0, len/2)`
			`r = copy(a, len/2, len)`
			`ls = sort(l)`
			`rs = sort(r)`
			`merge(ls, rs)`
				`p = partition(a, l, h)`
				`sort(a, l, p)`
				`sort(a, p + 1, h)`

Iterators

<FILL IN THE BLANK>



`public interface` **`Iterable<T>`**

Implementing this interface allows an object to be the target of the enhanced __________ statement (sometimes called the "__________ __________" statement).

`Iterator<T>`	`iterator()`	Returns an iterator over elements of type `T`.

<FILL IN THE BLANK>

`public interface` **`Iterable<T>`**

Implementing this interface allows an object to be the target of the enhanced **`for`** statement (sometimes called the "**for-each loop**" statement).

`Iterator<T>`	`iterator()`	Returns an iterator over elements of type `T`.

<FILL IN THE BLANK>

```
List<String> lst = new ArrayList<String>();
lst.add("a"); lst.add("b"); lst.add("c");
for(String s: lst) {
  System.out.println(s);
}
```

The highlighted kind of loop only works if that thing (lst) is an ________ or implements the interface ________<T>, ________ are a special case.

<FILL IN THE BLANK>

```
List<String> lst = new ArrayList<String>();
lst.add("a"); lst.add("b"); lst.add("c");
for(String s: lst) {
  System.out.println(s);
}
```

The highlighted kind of loop only works if that thing (lst) is an **array** or implements the interface **Iterable**<T>, **Arrays** are a special case.

<FILL IN THE BLANK>

public interface **Iterator<E>**

An iterator over a __________.

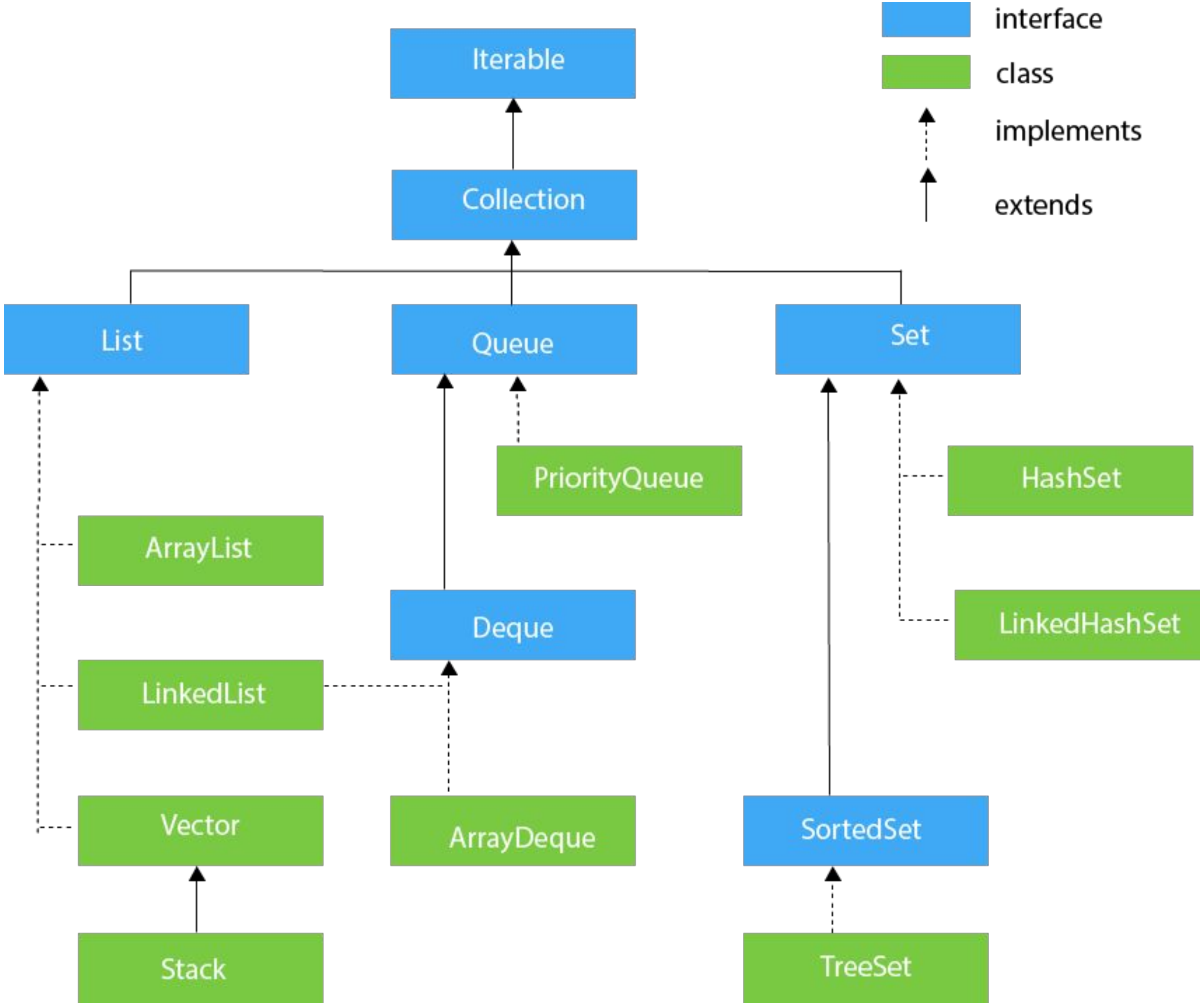
boolean	**hasNext**()	Returns _________ if the iteration has more elements.
E	**next**()	Returns the next _________ in the iteration.

<FILL IN THE BLANK>

`public interface` **`Iterator<E>`**

An iterator over a **collection**.

`boolean`	`hasNext()`	Returns **`true`** if the iteration has more elements.
`E`	`next()`	Returns the next **element** in the iteration.


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

<FILL IN THE BLANK>

```
List<String> lst = new ArrayList<>();
lst.add(“a”); lst.add(“b”); lst.add(“c”);

// create an iterator from lst
Iterator<String> iter = ___________
```

<FILL IN THE BLANK>

```
List<String> lst = new ArrayList<>();
lst.add(“a”); lst.add(“b”); lst.add(“c”);

// create an iterator from lst
Iterator<String> iter = lst.iterator();
```

What is printed by the highlighted line? (Assume the highlighted line and any lines above have been executed).

```
List<String> lst = new ArrayList<>();
lst.add("a");
lst.add("b");
lst.add("c");

// create an iterator from lst
Iterator<String> iter = lst.iterator();

System.out.println(iter.hasNext());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.hasNext());
System.out.println(iter.next());
```

What is printed by the highlighted line? (Assume the highlighted line and any lines above have been executed).

```
List<String> lst = new ArrayList<>();
lst.add("a");
lst.add("b");
lst.add("c");

// create an iterator from lst
Iterator<String> iter = lst.iterator();

System.out.println(iter.hasNext());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.hasNext());
System.out.println(iter.next());
```

```
true
```

What is printed by the highlighted line? (Assume the highlighted line and any lines above have been executed).

```
List<String> lst = new ArrayList<>();
lst.add("a");
lst.add("b");
lst.add("c");

// create an iterator from lst
Iterator<String> iter = lst.iterator();

System.out.println(iter.hasNext());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.hasNext());
System.out.println(iter.next());
```

```
true
```

What is printed by the highlighted line? (Assume the highlighted line and any lines above have been executed).

```
List<String> lst = new ArrayList<>();
lst.add("a");
lst.add("b");
lst.add("c");

// create an iterator from lst
Iterator<String> iter = lst.iterator();

System.out.println(iter.hasNext());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.hasNext());
System.out.println(iter.next());
```

```
true
a
```

What is printed by the highlighted line? (Assume the highlighted line and any lines above have been executed).

```
List<String> lst = new ArrayList<>();
lst.add("a");
lst.add("b");
lst.add("c");

// create an iterator from lst
Iterator<String> iter = lst.iterator();

System.out.println(iter.hasNext());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.hasNext());
System.out.println(iter.next());
```

```
true
a
```

What is printed by the highlighted line? (Assume the highlighted line and any lines above have been executed).

```
List<String> lst = new ArrayList<>();
lst.add("a");
lst.add("b");
lst.add("c");

// create an iterator from lst
Iterator<String> iter = lst.iterator();

System.out.println(iter.hasNext());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.hasNext());
System.out.println(iter.next());
```

```
true
a
b
```

What is printed by the highlighted line? (Assume the highlighted line and any lines above have been executed).

```
List<String> lst = new ArrayList<>();
lst.add("a");
lst.add("b");
lst.add("c");

// create an iterator from lst
Iterator<String> iter = lst.iterator();

System.out.println(iter.hasNext());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.hasNext());
System.out.println(iter.next());
```

```
true
a
b
```

What is printed by the highlighted line? (Assume the highlighted line and any lines above have been executed).

```
List<String> lst = new ArrayList<>();
lst.add("a");
lst.add("b");
lst.add("c");

// create an iterator from lst
Iterator<String> iter = lst.iterator();

System.out.println(iter.hasNext());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.hasNext());
System.out.println(iter.next());
```

```
true
a
b
c
```

What is printed by the highlighted line? (Assume the highlighted line and any lines above have been executed).

```
List<String> lst = new ArrayList<>();
lst.add("a");
lst.add("b");
lst.add("c");

// create an iterator from lst
Iterator<String> iter = lst.iterator();

System.out.println(iter.hasNext());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.hasNext());
System.out.println(iter.next());
```

```
true
a
b
c
```

What is printed by the highlighted line? (Assume the highlighted line and any lines above have been executed).

```java
List<String> lst = new ArrayList<>();
lst.add("a");
lst.add("b");
lst.add("c");

// create an iterator from lst
Iterator<String> iter = lst.iterator();

System.out.println(iter.hasNext());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.hasNext());
System.out.println(iter.next());
```

```
true
a
b
c
false
```

What is printed by the highlighted line? (Assume the highlighted line and any lines above have been executed).

```
List<String> lst = new ArrayList<>();
lst.add("a");
lst.add("b");
lst.add("c");

// create an iterator from lst
Iterator<String> iter = lst.iterator();

System.out.println(iter.hasNext());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.hasNext());
System.out.println(iter.next());
```

```
true
a
b
c
false
```

What is printed by the highlighted line? (Assume the highlighted line and any lines above have been executed).

```
List<String> lst = new ArrayList<>();
lst.add("a");
lst.add("b");
lst.add("c");

// create an iterator from lst
Iterator<String> iter = lst.iterator();

System.out.println(iter.hasNext());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.next());
System.out.println(iter.hasNext());
System.out.println(iter.next());
```

```
true
a
b
c
false


// Nothing prints: RUNTIME ERROR!!!
// java.util.NoSuchElementException
```

Which of the below blocks of code is Java running “under the hood” when running an enhanced for loop such as:

```
for(String s: lst) { System.out.println(s);}
```

A)

```
List<String> lst = new ArrayList<String>();
lst.add("a"); lst.add("b"); lst.add("c");
while(lst.hasNext()) {
  String s = lst.next();
  System.out.println(s);
}
```

B)

```
List<String> lst = new ArrayList<String>();
lst.add("a"); lst.add("b"); lst.add("c");
Iterable<String> iter = lst.iterator();
while(iter.hasNext()) {
  String s = iter.next();
  System.out.println(s);
}
```

C)

```
List<String> lst = new ArrayList<String>();
lst.add("a"); lst.add("b"); lst.add("c");
Iterator<String> iter = lst.iterator();
while(iter.hasNext()) {
  String s = iter.next();
  System.out.println(s);
}
```

Which of the below blocks of code is Java running “under the hood” when running an enhanced for loop such as:

`for(String s: lst) { System.out.println(s);}`

A)

```
List<String> lst = new ArrayList<String>();
lst.add("a"); lst.add("b"); lst.add("c");
while(lst.hasNext()) { // Calling on list itself!
  String s = lst.next(); // Calling on the list itself!
  System.out.println(s);
}
```

B)

```
List<String> lst = new ArrayList<String>();
lst.add("a"); lst.add("b"); lst.add("c");
Iterable<String> iter = lst.iterator();
while(iter.hasNext()) {
  String s = iter.next();
  System.out.println(s);
}
```

C)

```
List<String> lst = new ArrayList<String>();
lst.add("a"); lst.add("b"); lst.add("c");
Iterator<String> iter = lst.iterator();
while(iter.hasNext()) {
  String s = iter.next();
  System.out.println(s);
}
```

```java
class AList<E> implements List<E>, Iterable<E> {
  class AListIterator implements Iterator<E> {







  }

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public Iterator<E> iterator() {



  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  public int size() {
    return this.size;
  }

  /* ... set, expandCapacity omitted ... */

}
```

What fields does **AListIterator** need? Keeping in mind that we will be implementing `next()` and `hasNext()`.

```
class AList<E> implements List<E>, Iterable<E> {
  class AListIterator implements Iterator<E> {







  }

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public Iterator<E> iterator() {




  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  public int size() {
    return this.size;
  }

  /* ... set, expandCapacity omitted ... */

}
```

What fields does **AListIterator** need? Keeping in mind that we will be implementing `next()` and `hasNext()`.

int currentIndex;
int size;
AList<E> alist;