

# Announcements

---

Midterm 1 is Monday, 2/12, 8-10PM.

- Details here: <https://piazza.com/class/j9j0udrxjip758?cid=1105>
- Closed note, except you can bring one front/back handwritten sheet.
- Bring your Berkeley student ID (if you have one).
- Covers material up through inheritance3 (2/7, Wednesday's lecture).

Exam studying:

- Midterm 1 Review Session on Friday 2/9, 8-10PM in 155 Dwinelle
- Midterm 1 Guerilla Section on Saturday 2/10, 12-2PM in 271-275 Soda
- Lecture next Monday will also be an AMA/review thing.

# CS61B: 2018

---

## Lecture 11: Libraries

- Java Libraries
- Interfaces and Abstract Classes
- Packages

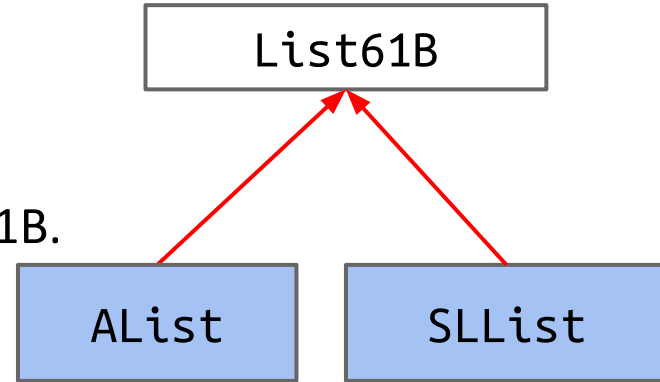
# Java Libraries

# Main Story of the Course (so far): Implementing Abstract Data Types

---

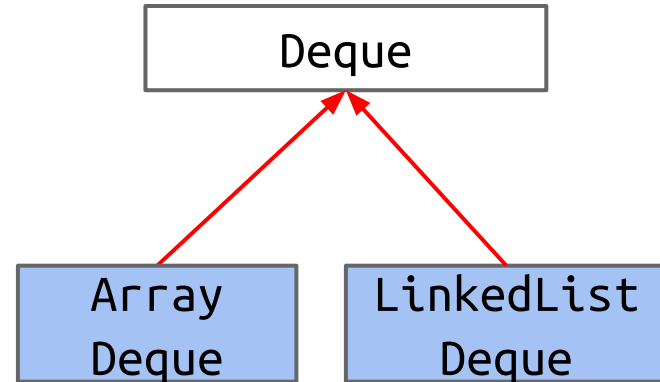
In Lecture:

- Developed ALists and SLLists.
- Created an interface List61B.
  - Modified AList and SLList to implement List61B.
  - List61B provided default methods.



In Projects:

- Developed ArrayDeque and LinkedListDeque.
- Created an interface Deque.
  - Modified AD and LLD to implement Deque.

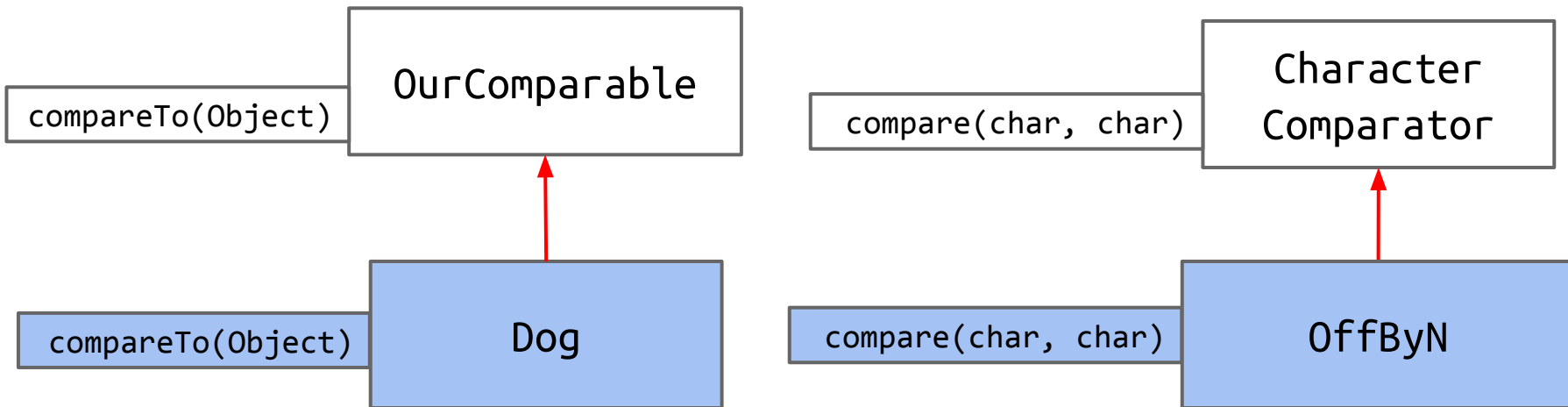


## Side Story: Interface Inheritance for Comparison

---

Two more uses for interfaces:

- Specify a contract for common behavior shared by many data structures (e.g. implementing `OurComparable` means Dogs can be compared).
- Provide a way to containerize common functions (e.g. the `OffByN` `CharacterComparator` lets us compare two characters in a special way).

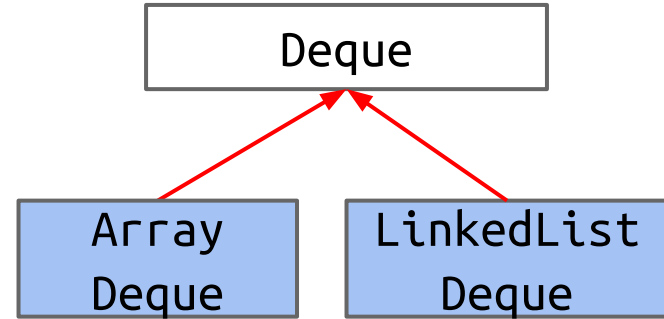


# Abstract Data Types

An **Abstract Data Type (ADT)** is defined only by its operations, not by its implementation.

Deque ADT:

- `addFirst(Item x);`
- `addLast(Item x);`
- `boolean isEmpty();`
- `int size();`
- `printDeque();`
- `Item removeFirst();`
- `Item removeLast();`
- `Item get(int index);`



ArrayDeque and LinkedList Deque are implementations of the Deque ADT.



# The Stack ADT: yellkey.com/stuff

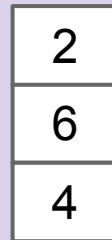
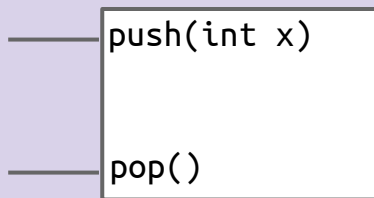
---

The Stack ADT supports the following operations:

- `push(int x)`: Puts `x` on top of the stack.
- `int pop()`: Removes and returns the top item from the stack

Which implementation do you think would result in faster overall performance?

- A. Linked List
- B. Array



# The Stack ADT: yellkey.com/stuff

---

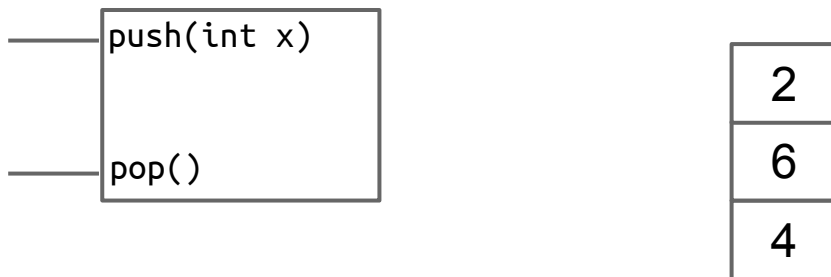
The Stack ADT supports the following operations:

- `push(int x)`: Puts `x` on top of the stack.
- `int pop()`: Removes and returns the top item from the stack

Which implementation do you think would result in faster overall performance?

**A. Linked List**

**B. Array**



Both are about the same. No resizing for linked lists, so probably a lil faster.



# The GrabBag ADT: yellkey.com/yet

---

The GrabBag ADT supports the following operations:

- `insert(int x)`: Inserts `x` into the grab bag.
- `int remove()`: Removes a random item from the bag.
- `int sample()`: Samples a random item from the bag (without removing!)
- `int size()`: Number of items in the bag.

Which implementation do you think would result in faster overall performance?

- A. Linked List
- B. Array

—	<code>insert(int x)</code>
—	<code>remove()</code>
—	<code>sample()</code>
—	<code>size(int i)</code>

# The GrabBag ADT: yellkey.com/yet

---

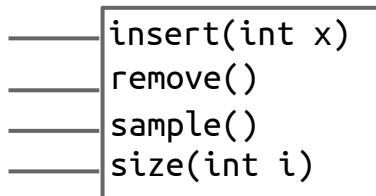
The GrabBag ADT supports the following operations:

- `insert(int x)`: Inserts `x` into the grab bag.
- `int remove()`: Removes a random item from the bag.
- `int sample()`: Samples a random item from the bag (without removing!)
- `int size()`: Number of items in the bag.

Which implementation do you think would result in faster overall performance?

A. Linked List

**B. Array**

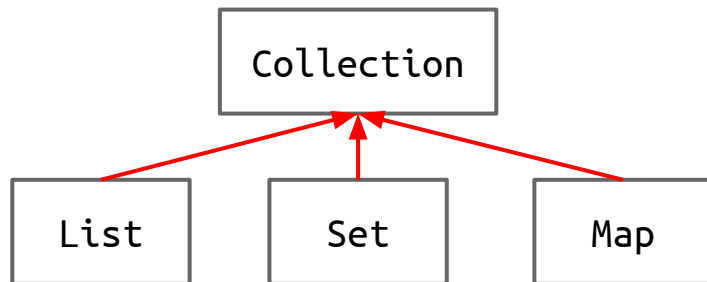


# Collections

---

Among the most important interfaces in the java.util library are those that extend the Collection interface (btw interfaces can extend other interfaces).

- Lists of things.
- Sets of things.
- Mappings between items, e.g. jhug's grade is 88.4.
  - Maps also known as associative arrays, associative lists (in Lisp), symbol tables, dictionaries (in Python).



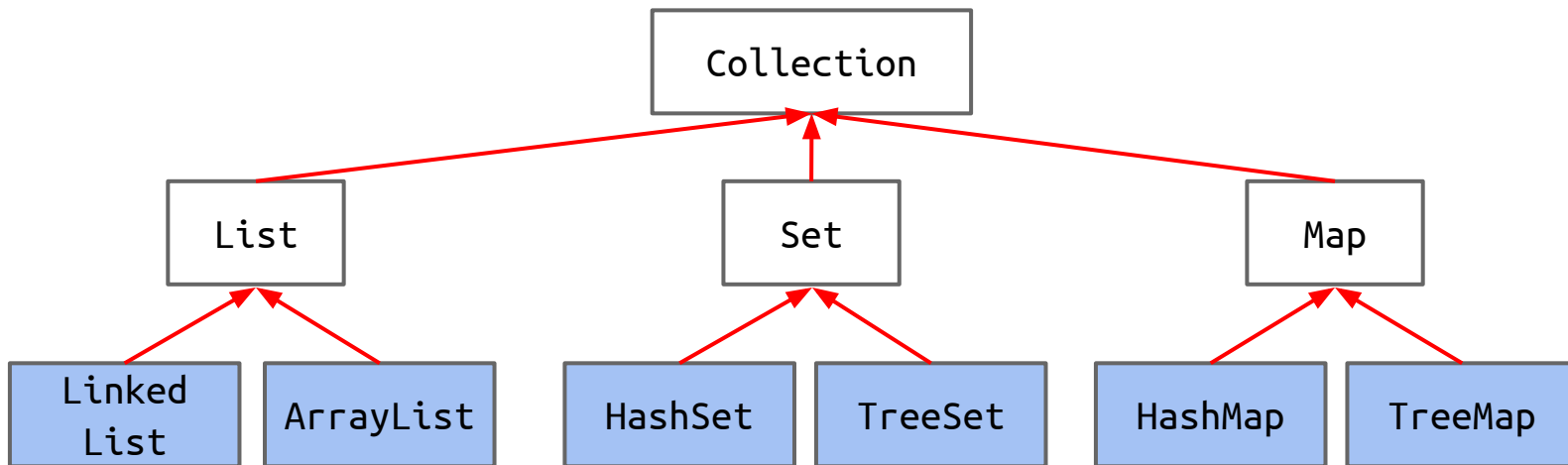
# Java Libraries

---

The built-in java.util package provides a number of useful:

- Interfaces: ADTs (lists, sets, maps, priority queues, etc.) and other stuff.
- Implementations: Concrete classes you can use.

Let's try them out (next slide).



# Tasks

---


3 tasks, given the text of a book:

- Create a list of all words in the book.
- Count the number of unique words.
- Keep track of the number of times that specific words are mentioned.

## Example: Using a Set to Count Unique Words

---

```
public static int countUniqueWords(List<String> words) {  
    Set<String> ss = new HashSet<>();  
    for (String s : words) {  
        ss.add(s);  
    }  
    return ss.size();  
}
```



This is known as an  
“enhanced for loop” or  
sometimes a foreach loop.

```
public static int countUniqueWords(List<String> words) {  
    Set<String> ss = new HashSet<>();  
    ss.addAll(words);  
    return ss.size();  
}
```

## Example: Using a Map to Create Counts of Specific Words

---

```
public static Map<String, Integer>
    collectWordCount(List<String> words, List<String> targets) {
    Map<String, Integer> wordCounts = new HashMap<>();
    for (String s : targets) {
        wordCounts.put(s, 0);
    }
    for (String s : words) {
        if (wordCounts.containsKey(s)) {
            int oldCount = wordCounts.get(s);
            wordCounts.put(s, oldCount + 1);
        }
    }
    return wordCounts;
}
```

# Java vs. Python

---

```
public static Map<String, Integer>
    collectWordCount(List<String> words, List<String> targets) {
    Map<String, Integer> wordCounts = new HashMap<>();
    for (String s : targets) {
        wordCounts.put(s, 0);
    }
    for (String s : words) {
        if (wordCounts.containsKey(s)) {
            int oldCount = wordCounts.get(s);
            wordCounts.put(s, oldCount + 1);
        }
    }
    return wordCounts;
}
```

```
def find_word_count(words, targets):
    word_counts = {}
    for s in targets:
        word_counts[s] = 0

    for s in words:
        if s in word_counts:
            word_counts[s] += 1

    return word_counts
```



# Java vs. Python

---

In every language, there are some features that are “first class citizens”.

- Often a special syntax for creating and using such objects.

Sets, dictionaries, tuples and lists in Python have special syntax.

- These collections do not have a special syntax in Java.
- Idea of “collection literals” was scrapped in [2014](#).

```
num_legs = {"horse": 4, "dog":  
4, "human": 2, "fish": 0}
```

```
Map<String, Integer> numLegs =  
new TreeMap<>();  
numLegs.put("horse", 4);  
numLegs.put("dog", 4);  
numLegs.put("human", 2);  
numLegs.put("fish", 0);
```

# Java vs. Python

---

In Java, programmer can decide which *implementation* of an *abstract data type* that they want to use.

- Allows power user to explicitly handle engineering tradeoffs.

Example: Basic Hashmaps ops are faster than TreeMaps, but TreeMaps provide efficient operations that involve order (e.g. get all keys less than).

```
Map<String, Integer> numLegs =  
new HashMap<>();  
numLegs.put("horse", 4);  
numLegs.put("dog", 4);  
numLegs.put("human", 2);  
numLegs.put("fish", 0);
```

```
Map<String, Integer> numLegs =  
new TreeMap<>();  
numLegs.put("horse", 4);  
numLegs.put("dog", 4);  
numLegs.put("human", 2);  
numLegs.put("fish", 0);
```

# Factory Methods for Collections (in Java 9)

---

In Java 9, factory methods were added to the language.

- Example: `Set<Integer> S = Set.of(3, 4, 5, 6);`
- Similar to `IntList.of()`.
- Allows you to create immutable collections in one line.
  - Cannot add or remove items!
- Underlying implementation is hidden from user (don't know your map is `HashMap`, a `TreeMap`, or something else).

```
Map<String, Integer> numLegs =  
new HashMap<>();  
numLegs.put("horse", 4);  
numLegs.put("dog", 4);  
numLegs.put("human", 2);  
numLegs.put("fish", 0);
```

```
Map<String, Integer> numLegs =  
Map.of("horse", 4, "dog", 4,  
"human", 2, "fish", 0);
```

# Why Java in 61B?

---

Arguably, takes less time to write programs, due to features like:

- Static types (provides type checking and helps guide programmer).
- Bias towards interface inheritance leading to cleaner subtype polymorphism.
- Access control modifiers make abstraction barriers more solid.

More efficient code, due to features like:

- Ability to have more control over engineering tradeoffs.
- Single valued arrays lead to better performance.

Basic data structures more closely resemble underlying hardware:

- Would be weird to do ArrayDeque in Python, since there is no need for array resizing. However, in hardware (see 61C), variable length arrays don't exist.

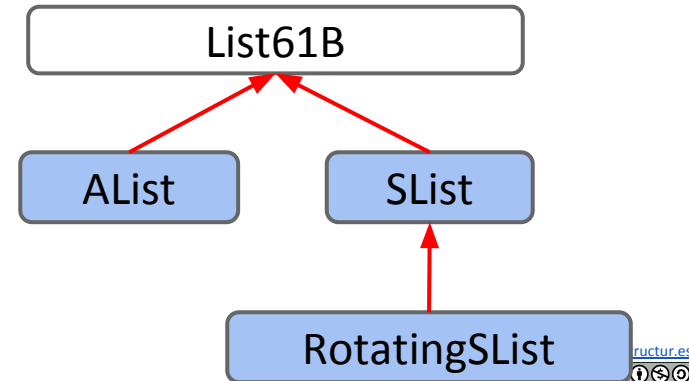
# **Interfaces and Abstract Classes**

# Inheritance Summary (So Far)

---

In the last three lectures we've seen how implements and extends can be used to enable **interface inheritance** and **implementation inheritance**.

- Interface inheritance: What (the class can do).
- Implementation inheritance: How (the class does it).



# A Closer Look at Interfaces

---

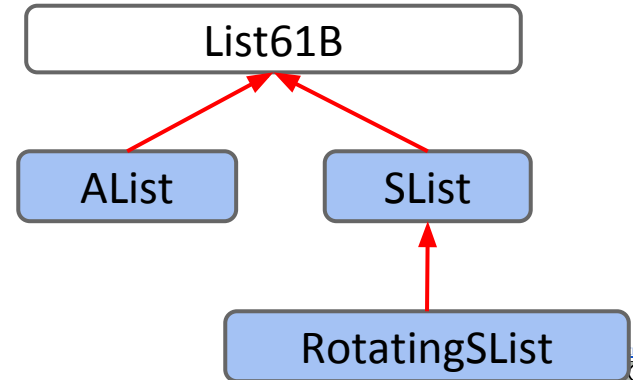
Interfaces may combine a mix of **abstract** and **default** methods.

- Abstract methods are **what**. And must be overridden by subclass.
- Default methods are **how**.

Not explicitly mentioned on a slide before:

- Unless you use the keyword **default**, a method will be **abstract**.
- Unless you specify an access modifier, a method will be **public**.

```
public interface List61B<Item> {  
    void insertFront(Item x);  
    ...  
    default public void print() { ... }  
}
```



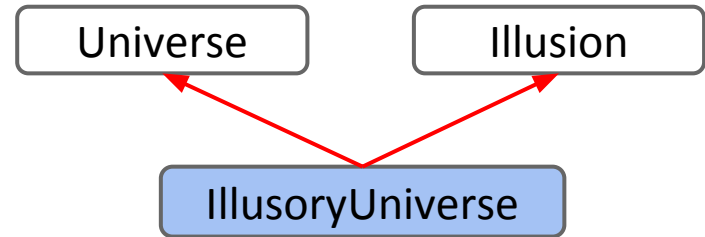
# Interfaces, Even More

---

More interface details:

- Can provide variables, but they are **public static final**.
  - final means the value can never change. Use for constants:  $G=6.67e-11$
- A class can implement multiple interfaces.

```
public interface Universe {  
    double gravity = 6.67e-11;  
    void update(double dt);  
    ...  
    default void draw() { ... }  
}
```





# Interface Summary

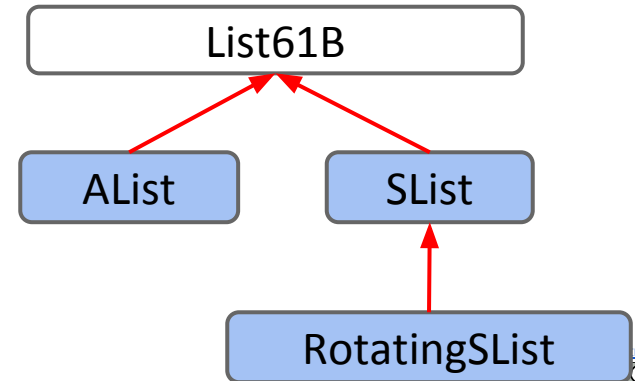
---

Interfaces:

- Cannot be instantiated.
- Can provide either **abstract** or **concrete** methods.
  - Use no keyword for abstract methods.
  - Use **default** keyword for concrete methods.
- Can provide only public static final variables.

no instance  
variables

Java 9 added private methods to interfaces.



# Introducing: Abstract Classes

Abstract classes are an intermediate level between interfaces and classes.

- Cannot be instantiated.
- Can provide either **abstract** or **concrete** methods.
  - Use **abstract** keyword for abstract methods.
  - Use no keyword for concrete methods.
- Can provide variables (any kind).
- Can provide protected and package private methods [after mt1].

Similarities

Differences

opposite of  
interfaces

```
public abstract class GraphicObject {  
    public int x, y;  
    ...  
    public void moveTo(int newX, int newY) { ... }  
    public abstract void draw();  
    public abstract void resize();  
}
```

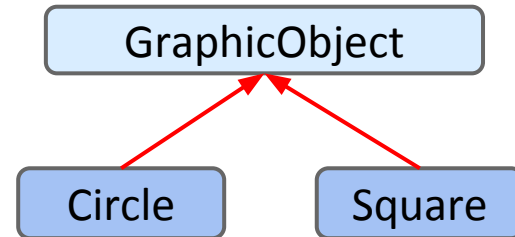
GraphicObject

## Example (From Oracle's Abstract Class Tutorial)

---

```
public abstract class GraphicObject {  
    public int x, y;  
    ...  
    public void moveTo(int newX, int newY) { ... }  
    public abstract void draw();  
    public abstract void resize();  
}
```

```
public class Circle extends GraphicObject {  
    public void draw() { ... }  
    public void resize() { ... }  
}
```

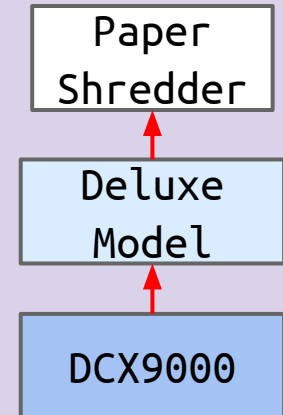


Implementations must override ALL abstract methods.

```
public interface PaperShredder {  
    void shred(Document d);  
    void shredAll(Document[] d);  
}  
  
public abstract class DeluxeModel  
    implements PaperShredder {  
    public int count = 0;  
    public void count() { return count; }  
  
    public shredAll(Document[] d) {  
        for (int i = 0; i < d.length; i += 1) {  
            shred(d);  
        }  
    }  
  
    public abstract void connectToWifi();  
}
```

How many abstract methods  
**must** DCX9000 override?

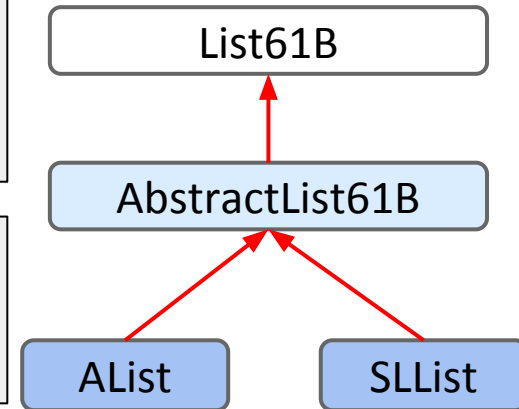
- A. 0
- B. 1
- C. 2
- D. 3



# Common Use Case: Providing Basics for Interface Implementation

```
public abstract class AbstractList61B<T>
    implements List61B<T> {
    int size = 0;
    public AbstractList61B() { size = 0; }
    @Override
    public int size() {
        return size;
    }
}
```

```
public class AList<T>
    extends AbstractList61B<T> { ...
```

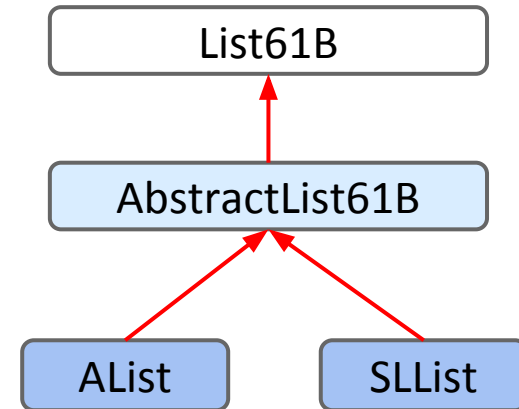


# Common Use Case: Providing Basics for Interface Implementation

---

Abstract classes are often used as partial implementations as interfaces.

- Good: Avoids the need to write a `size()` method or declare a size variable in `AList` and `SLList`.
- Bad: Starts getting confusing to understand where things are defined. May make too strong constraints on the implementations of our concrete classes (e.g. maybe you don't want a size variable).



# Summary: Abstract Classes vs. Interfaces

---

## Interfaces:

- Primarily for interface inheritance. Limited implementation inheritance.
- Classes can implement multiple interfaces.

## Abstract classes:

- Can do anything an interface can do, and more.
- Subclasses only extend one abstract class.

In my opinion, you should generally prefer interfaces whenever possible.

- Why? More powerful programming language constructs introduce complexity.
- If you're curious, see Oracle's [examples of when to use each](https://datastructure.es/examples-of-when-to-use-each).

# Where Abstract Classes Are Used in Java Standard Libraries

---

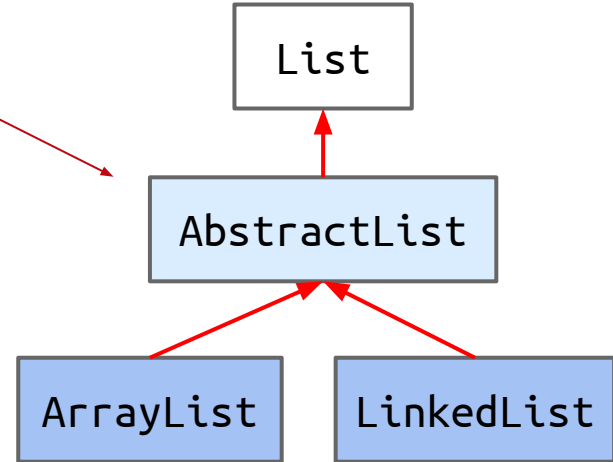
A more accurate hierarchy for lists is shown below.

- AbstractList provides default implementations for methods.
- Why not just put them in List itself? No default methods in Java interfaces until 2014, and the AbstractList was public so can't just throw it away.

```
public abstract class AbstractList<E>
    implements List<E> {
    ...

    public boolean add(E e) {
        add(size(), e);
        return true;
    }
}
```

Not absolutely necessary. Default methods could have been in List interface instead.



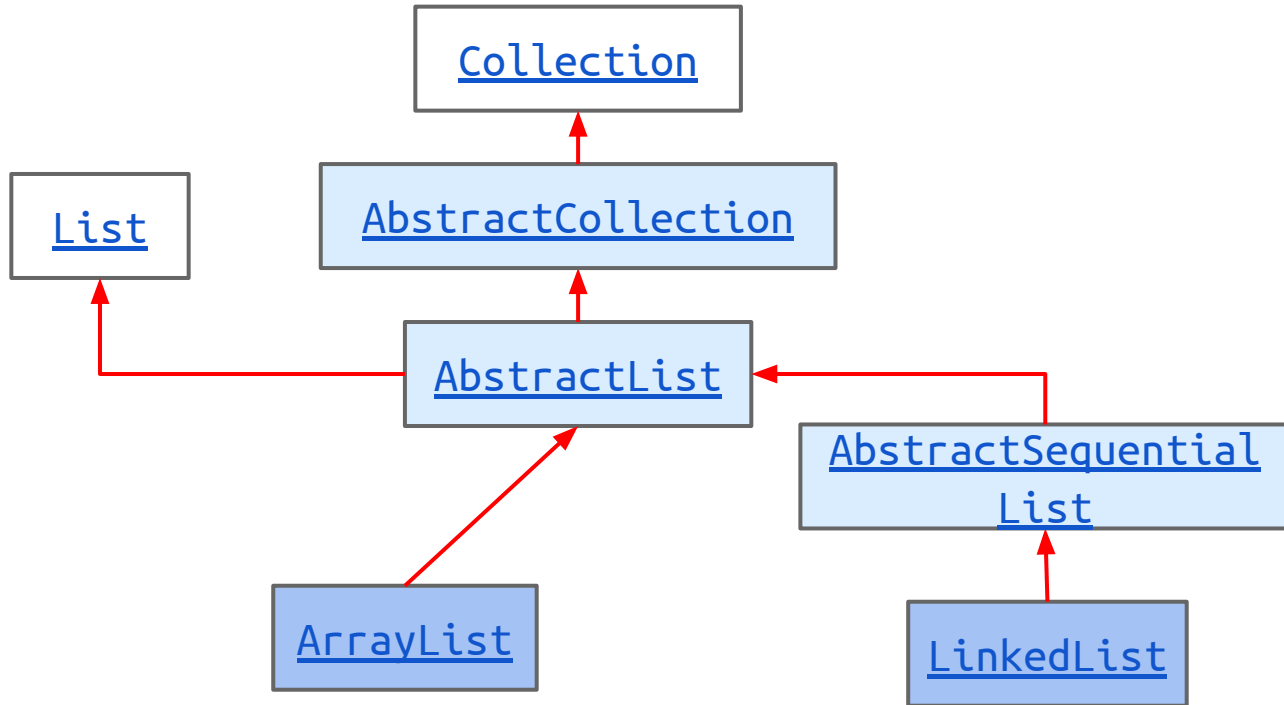


# Where Abstract Classes Are Used in Java Standard Libraries

---

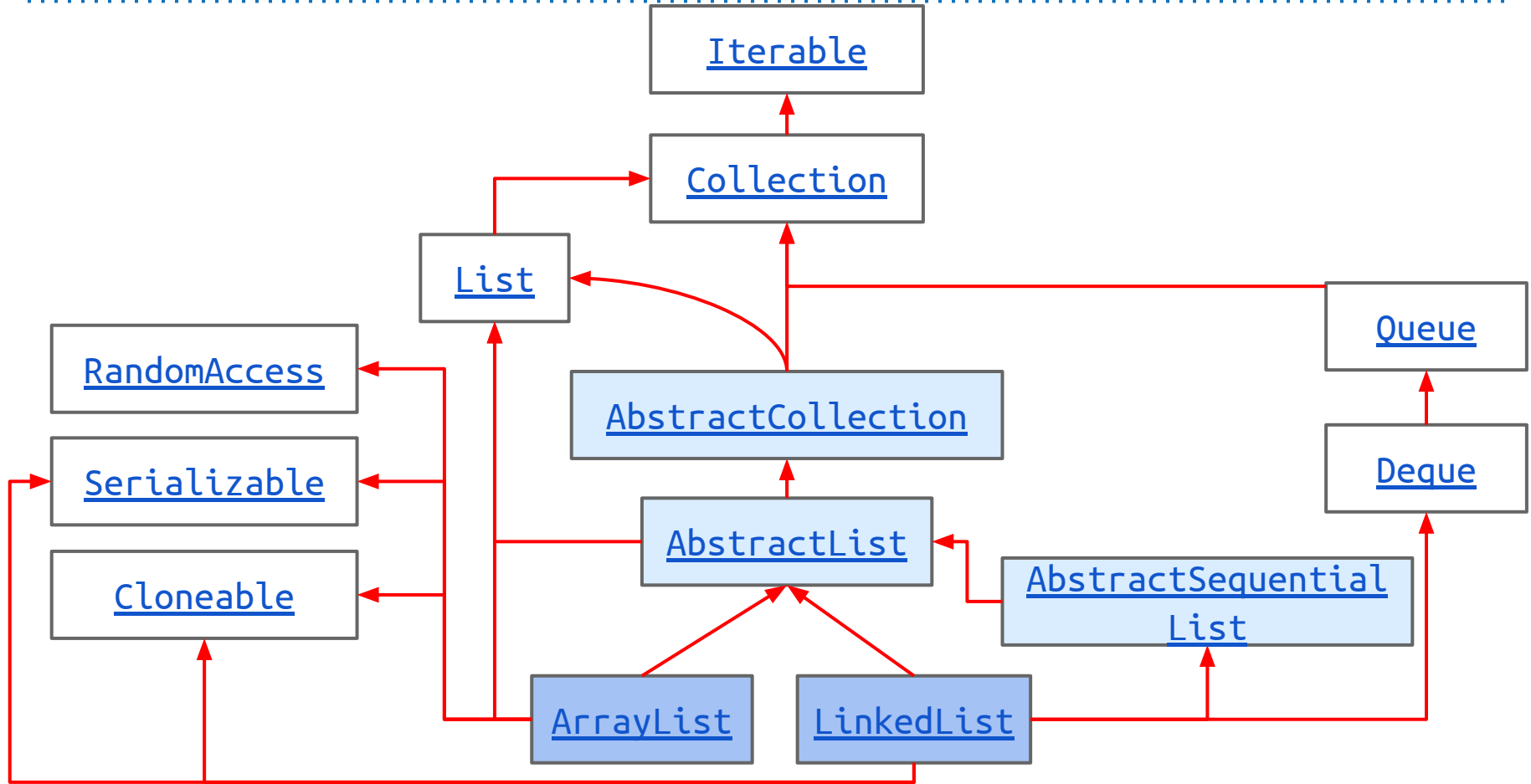
(Most of) the real list hierarchy is shown below.

- This isn't important, but you might find it interesting.
- Click links if you want to browse the source code.



# The Whole Shebang

---



# Packages

# The Zen of Python

---

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one-- and preferably only one --obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *\*right\** now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea -- let's do more of those!

# Canonicalization

---

What we'd really like is the ability to provide a canonical name for everything.

- ***Canonical representation***: A unique representation for a thing.
- Not-canonical: License plate number (can be reused, can change).
- Canonical: The VIN number **JYA3AWC04VA071286** (refers to a specific motorcycle).

In Java, we (attempt to) provide canonicity through by giving every a class a “package name”.

- A package is a ***namespace*** that organizes classes and interfaces.
- Today is just a brief look. We'll talk more about packages in two weeks.
- In HW1, after the midterm, we'll make our own.

# Packages

---

To address the fact that classes might share names:

We won't follow this rule. Our code isn't intended for distribution.

- A package is a ***namespace*** that organizes classes and interfaces.
- Naming convention: Package name starts with website address (backwards).

```
package ug.joshh.animal;  
  
public class Dog {  
    private String name;  
    private String breed;  
    private double size;  
}
```

Dog.java

If used from the outside, use entire ***canonical name***.

```
ug.joshh.animal.Dog d =  
    new ug.joshh.animal.Dog(...);
```

```
org.junit.Assert.assertEquals(5, 5);
```

If used from another class in same package (e.g. ug.joshh.animal.DogLauncher), can just use ***simple name***.

# Importing Classes

---

Typing out the entire name can be annoying.

- Entire name:

```
ug.jossh.animal.Dog d =  
    new ug.jossh.animal.Dog(...);
```

- Can use import statement to provide shorthand notation for usage of a single class in a package.

```
import ug.jossh.animal.Dog;  
Dog d = new Dog(...);
```

- Wildcard import: Also possible to import multiple classes, but this is often a bad idea!
  - Use sparingly.

```
import ug.jossh.animal.*;  
Dog d = new Dog(...);
```



Dangerous! Will cause compilation error if  
another \* imported class contains Dog.

# Importing Static Members

---

On the previous slide we saw how to import classes.

- Example:

```
import org.junit.Assert;  
Assert.assertEquals(5, 5);
```


`import static` lets us import static members of a class.

- Example:

```
import static org.junit.Assert.assertEquals;  
assertEquals(5, 5);
```

We've done this already. This is probably the only wildcard import that you should do in this course.

```
import static org.junit.Assert.*;  
assertEquals(5, 5);
```



Unlikely that any other library will have any static members with same name as members in `org.junit.Assert` class

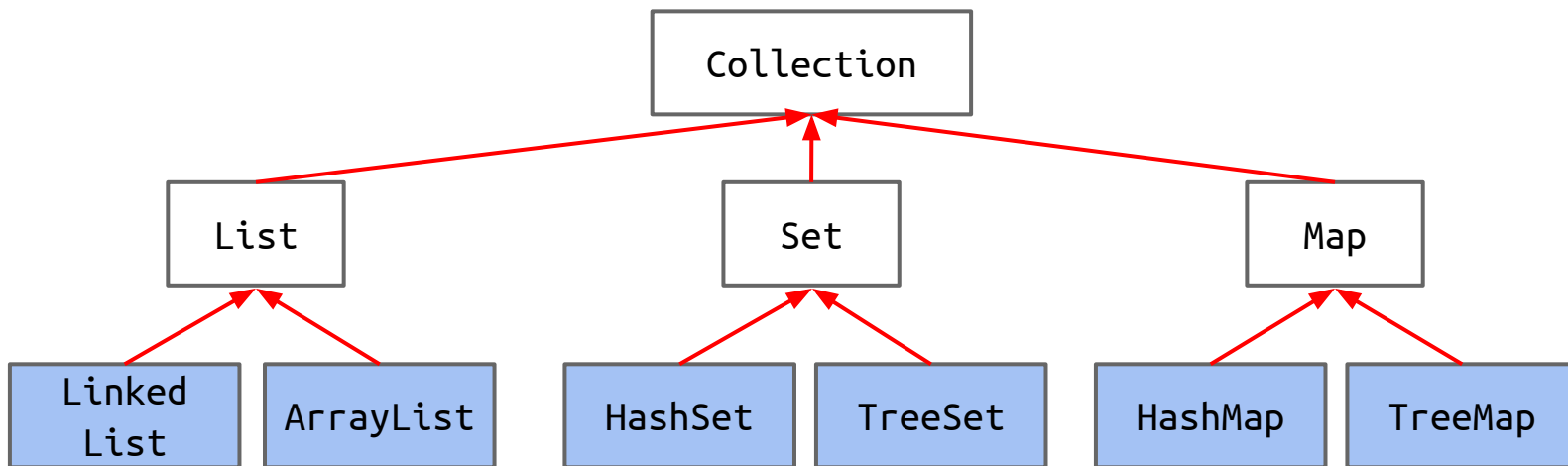


## Summary: Java Libraries

---

The built-in java.util package provides a number of useful:

- Interfaces: ADTs (lists, sets, maps, priority queues, etc.) and other stuff.
- Implementations: Concrete classes you can use.



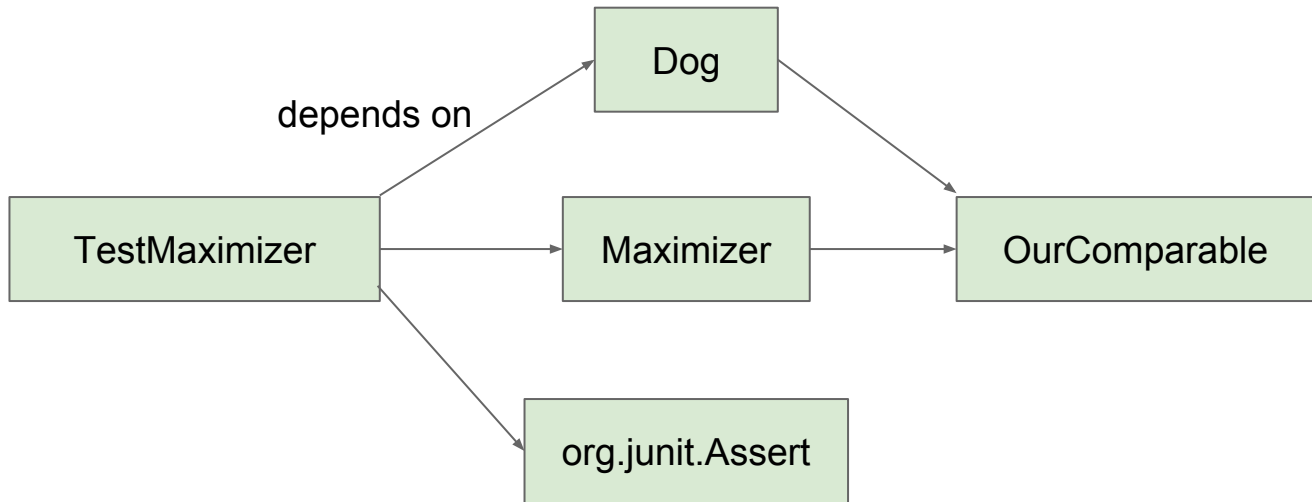
# **Extra Slides For Command Line Users (no video)**

# Dependencies

---

Java classes may be dependent on other classes.

- Dog depends on OurComparable.
- For Dog.java to compile, javac must know where OurComparable.class is.
- For Dog.class to execute, java must know where OurComparable.class is.

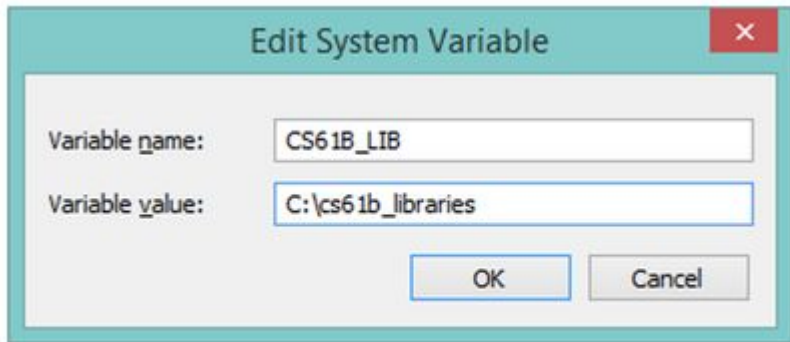


# Libraries

At the beginning of the semester, we had you do a bunch of mysterious computer setup stuff in labs 1b, 2b, and 3b.

Question:

- **How does the Java compiler/interpreter know where to find the libraries we provide in the skeleton repo (e.g. JUnit)?**



- **CLASSPATH:** Set this to `%CLASSPATH%;%CS61B_LIB%\*;.`;

A terminal window with a dark background. The prompt is 'jug@Hvlargs-MacBook-Pro'. The command entered is '\$ pico .bashrc'. The cursor is at the end of the command.

3. Now add a line to the bottom of your `.bashrc` file that says **export**

```
CLASSPATH="$CLASSPATH:%CS61B_LIB_DIR:./
```

This creates an environment variable called CLASSPATH. Everytime you use javac, it will look

# Environment Variables

---

Every program running on your computer has access to the system's "environment variables".

- Windows, Mac OS X, and Linux all provide this functionality.
- Environment variables are all strings.
- Every terminal window has its own environment variables.

Examples from my computer:

- `CLASSPATH = :/home/jug/course-materials-sp16/javalib/*:./`
- `LOGNAME=jug`
- `PWD=/home/jug/Dropbox/61b`

The Java compiler / interpreter (being programs) have access to these variables.

# The CLASSPATH

---

Example from my computer:

- CLASSPATH = `:/home/jug/course-materials-sp16/javailib/*:./`

The Java compiler and interpreter assume there is an environment variable called CLASSPATH, and look in those folders for dependencies.

- In Linux or MacOS, paths are separated by : (colon)
- In Windows, paths are separated by ; (semi-colon)

With classpath above, java and javac check for dependencies in:

- `/home/jug/course-materials-sp16/javailib/*`
- `./`

`./` means current directory  
`../` means directory one up

\* means don't just look at .class files but also look inside any .jar file.

# The World According to the Interpreter

---

The java interpreter:

- Gets a command line argument specifying the class to execute (e.g. EnvMap)
- Uses CLASSPATH variable to try to find that file (and dependencies).
  - Example: java looks for EnvMap.class and dependencies in two places:
    - `./`
    - `/Users/jug/work/61b/course-materials/lib/`

Paths may be:

- relative, e.g. `./`
- absolute

Note: The echo command is my OS's version of print.

```
$ echo $CLASSPATH
:/Users/jug/work/61b/course-materials/lib/*:./
jug ~/work/lectureCode-sp16/lec12/examples
$ java EnvMap
Printing your environment variables:
TERM=xterm-16color
TERM_PROGRAM_VERSION=343
SHLVL=1
```

## Naming Conflicts. PollEv.com/jhug Text “jhug” to 37607

---

Suppose we have two copies of Dog.class:

1. ~/work/lectureCode-sp16/lec12/examples/weirdDog/Dog.class
2. ~/work/lectureCode-sp16/lec12/examples/Dog.class

If we run the following command with the CLASSPATH shown, what do you think happens?

- A. Runtime error.
- B. Random results.
- C. Version #1 runs.
- D. Version #2 runs.

```
$ echo $CLASSPATH
./weirdDog:./
jug ~/work/lectureCode-sp16/lec12/examples
$ java Dog
```

Reminder: The echo command prints out an environment variable on the next line.



# Naming Conflicts Are Resolved in Classpath Order

---

Suppose we have two copies of Dog.class:

1. `~/work/lectureCode-sp16/lec12/examples/weirdDog/Dog.class`
2. `~/work/lectureCode-sp16/lec12/examples/Dog.class`

If we run the following command with the CLASSPATH shown, what do you think happens?

**C. Version #1 runs.**

```
$ echo $CLASSPATH
./weirdDog:./
jug ~/work/lectureCode-sp16/lec12/examples
$ java Dog
```

Why? The classpath has two entries. At it happens, Java scans for .class folders in the order given, running the first one it finds.

# The CLASSPATH Through Command Line

---

You can instead specify the classpath using the command line argument `-cp`.

Example: `javac -cp ./:/home/horse/stuff/:. ./ Moo.java`

- CLASSPATH environment variable totally ignored.
- Instead, `Moo.java`'s dependencies are searched for in:
  - Current directory `./`
  - `/home/horse/stuff`
  - One directory up `../`

As with the CLASSPATH variable, if multiple copies of any `Moo.java` dependency are found, they are resolved in order of the CLASSPATH.

- Current directory first, then the horse directory, then one directory up.

# The CLASSPATH and IntelliJ

---

If you're using IntelliJ, the CLASSPATH environment variable is irrelevant.

- IntelliJ automatically calls javac and java with the appropriate -cp argument.
- -cp argument that it uses is based on whatever libraries you have specified for the current project.
  - Under the hood, IntelliJ keeps a List of library folders and turns this list into a -cp string anytime you compile or run.

In case you want to see your IntelliJ classpath:

```
import java.net.URL;
import java.net.URLClassLoader;

public static void main(String[] args) {
    ClassLoader cl = ClassLoader.getSystemClassLoader();

    URL[] urls = ((URLClassLoader)cl).getURLs();

    for(URL url: urls){
        System.out.println(url.getFile());
    }
}
```

# A CLASSPATH Puzzle

---

- Suppose you submit the following to gradescope:
  - AGTestArrayDeque.class
  - ArrayDeque.java
- Suppose that:
  - The Autograder is configured with the CLASSPATH above.
  - The Autograder is run with the command `java Autograder` from a folder which contains all files submitted by the student.
  - All files created by the staff are placed in the folder `/Users/jug/work/61b/course-materials/lib/`
  - `Autograder.class` makes calls to methods in `AGTestArrayDeque.class`.

```
$ echo $CLASSPATH
./:/Users/jug/work/61b/course-materials/lib/
```

What happens when `java Autograder` executes?

# Package Creation and Invocation Gotchas

---

Common gotchas for creating and using packages from command line:

- Interpreter (java command) needs class files to be in folder structure that matches package names: `ug.josHH.animal` package must be in `ug/josHH/animal`.
  - Pro-tip: Can use `javac -d` flag to generate the appropriate folders.

```
jug ~/work/lectureCode-sp16/lec12/hugCode/ug/josHH/animal
$ ls
Dog.class  Dog.java
```

- Execution of main methods inside a package requires uses of package name, and must be done from the top folder (e.g. above `ug`).

```
jug ~/work/lectureCode-sp16/lec12/hugCode/
$ java ug.josHH.animal.DogLauncher
frankie is a barnacle dog weighing 22.0 lbs.
```