Java Quiz!

SUBMIT ON BLACKBOARD

- Write a program to:
- Store two types of pets---cats and dogs
  - When you create a pet, constructor takes a name. Cats also take a number of lives remaining.
  - All pets have a printName() function that prints the name
  - All pets have a makeNoise() function
    - Cats: "NAME says meow" and dogs: "NAME says woof"
- Your main method should:
  - Create an ArrayList with two dogs named Fido and Spot
  - Add three cats named Fluffy, Mowzer, and Pig
  - Print the names of all pets
  - Call the makeNoise function on the first dog and second cat
- **Use good OOP practices**

CS 2113
Software Engineering

Lecture 7: Collecting Data

```
git clone https://github.com/cs2113f16/lec-7.git
```

Professor Tim Wood - The George Washington University

Lately...

- Object Oriented Programming
  - Classes, hierarchies, inheritance
  - UML Class diagrams -- maybe more on this later

- Linked List
  - C and Java
  - How is it going?

- No reading quiz in this week's lab... but expect one for next week
  - **Read: Chapters 7, 8, and 9 for Monday**
  - Worksheet on Java memory due in class on Wednesday (no programming exercise this week)

Late Policy

- You get 2 Late Passes
  - Each pass extends a deadline by 48 hours
- You must email me and yawei@gwu.edu before the original deadline to use a late pass
  - You do not need to explain why you are using a pass
- Otherwise you lose points:
  - 5 points deducted per 8 hour period = 15 points per day
  - (No late submissions for weekly exercises)

Exams

Midterm: 10/26 in class

Final: Wednesday Dec 14th 5:20-7:20PM

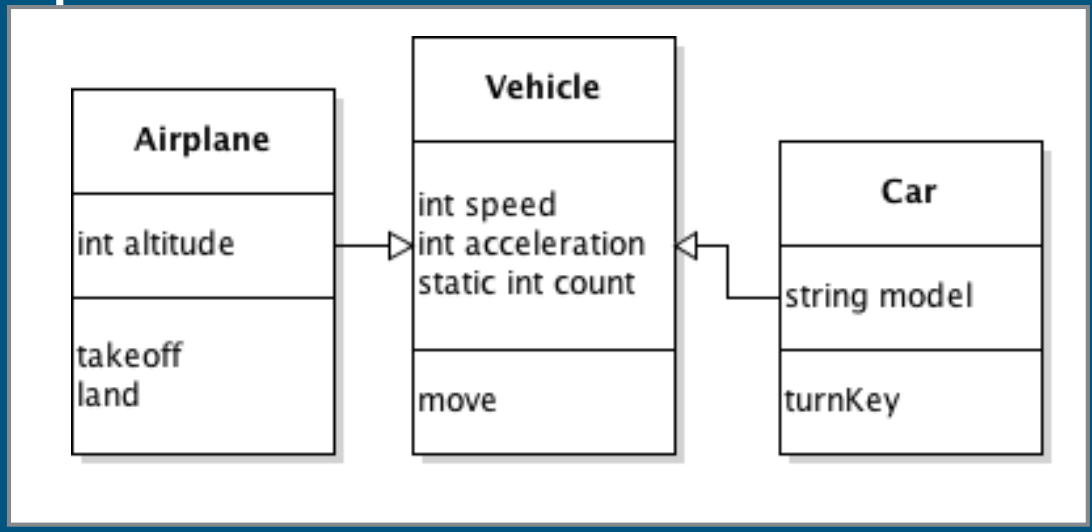
If you have conflicts, let me know in advance!

This Time

- Classes, Objects, and Memory
  - Where does it go?
  - Garbage Collection
- Advanced Data Structures
  - Writing our own
  - Java Collections
- Advanced OOP
  - Polymorphism
  - Introspection

Class Hierarchies

- UML Class Diagrams help us visualize relations



- Write the code for these class
  - Just define the functions and data members
- Fill in worksheet
  - Ignore the main function and memory layout for now…
- If you finish early, be sure to get today's files

```
git clone https://github.com/cs2113f16/lec-7.git
```

Classes, Objects, and Memory

- What will memory look like after running this code?

```
public class Vehicle {
   public int speed;
   private int acceleration;
   public static int count;
   // ...
}

public class Car extends Vehicle {
  private String model;
  public void turnKey(){ … }

  public static void main() {
    int s = 65;
    Vehicle v;
    Vehicle v2 = new Vehicle();
    v2.speed = s;
    Car c = new Car("Honda");
    c.count = 50;
  }
```

Stack		
Address	*Name*	*Contents*
10000	s	65
10008	v	NULL
10016	v2	500000
10016	c	

Heap	
Address	*Contents*
500000	vehicle speed = 65
	vehicle acceleration = 0
	car speed = 65
	car acceleration = 0
	car model = "honda"
	Vehicle.count = 0

Where Does it Go?

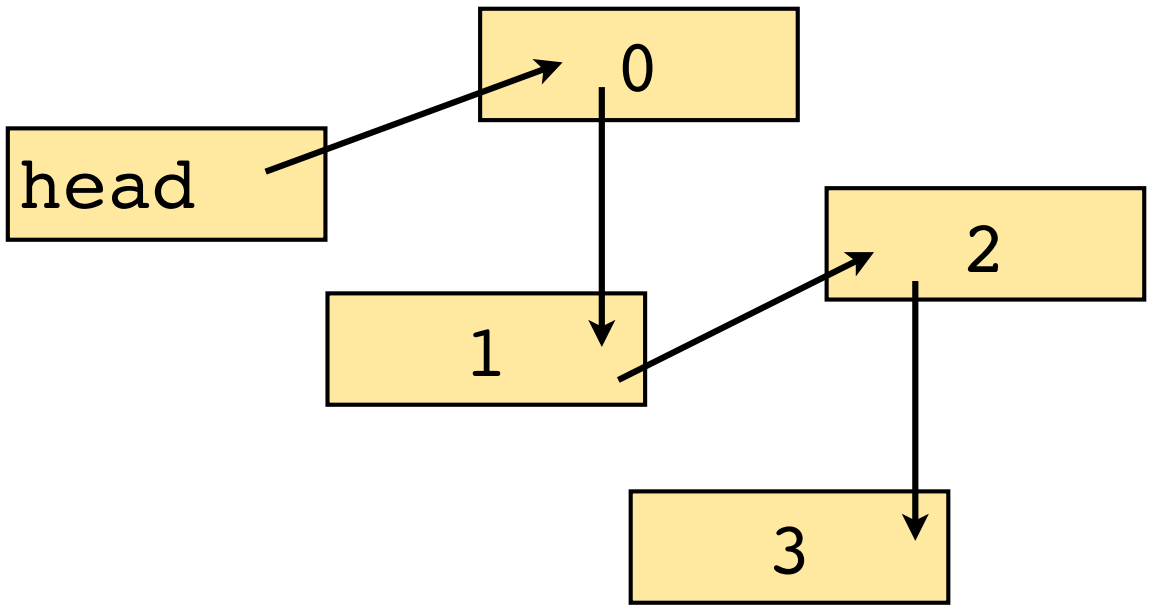
- In C we had to call "free()" to make sure that the memory we used was cleaned up
- How come we don't need to do this in Java?

```
MyLinkedList L =
  new MyLinkedList();
L.add(0);
L.add(1);
L.add(2);
L.add(3);

L.remItemAt(0);
L.remItemAt(0);
```

Garbage Collection

- The Java Run Time automatically tracks what objects are actively being used in memory

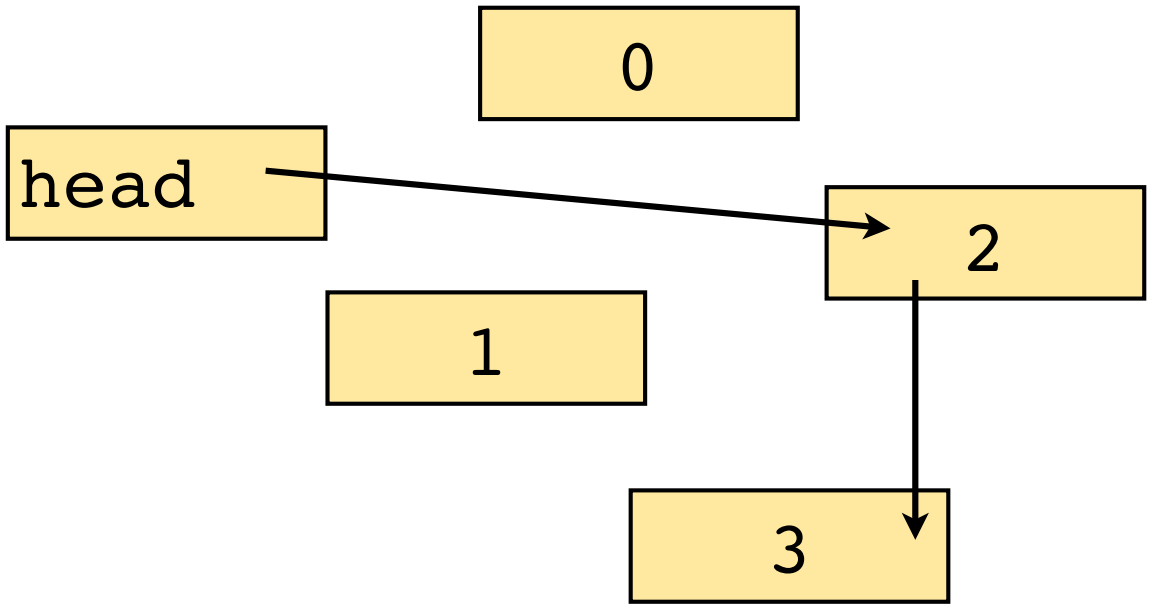
```
MyLinkedList L =
  new MyLinkedList();
L.add(0);
L.add(1);
L.add(2);
L.add(3);

L.remItemAt(0);
L.remItemAt(0);
```



Garbage Collection

- The Java Run Time automatically tracks what objects are actively being used in memory
  - If no variable references something, then that object is "lost"---can be deleted

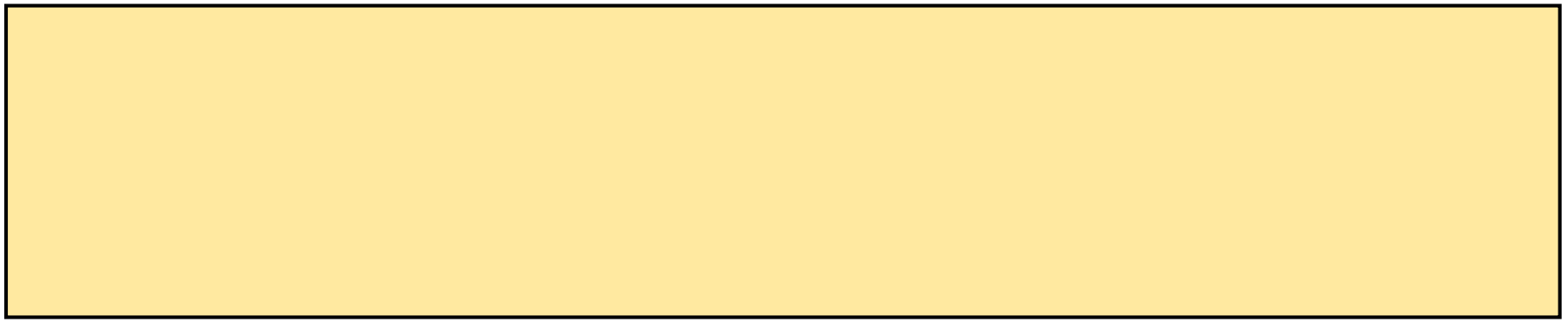
```
MyLinkedList L =
  new MyLinkedList();
L.add(0);
L.add(1);
L.add(2);
L.add(3);

L.remItemAt(0);
L.remItemAt(0);
```



How might it do this?

- How to find which objects on the heap are reachable (or not)?

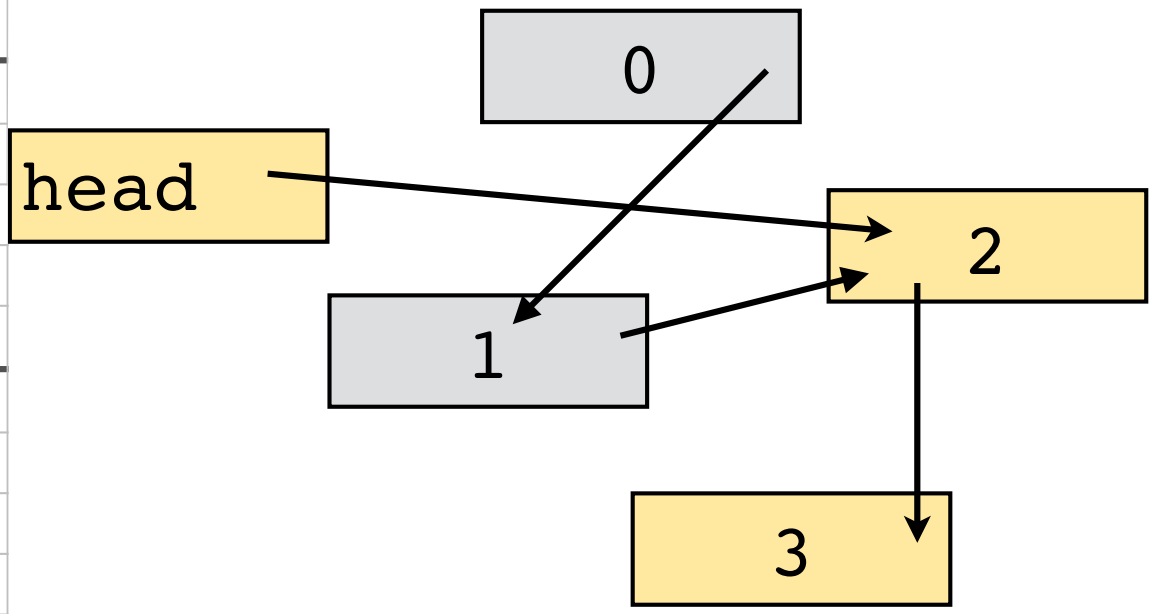
- Program knows:



Keep:
 - objects that have a reference to them from the stack
Loop:
 - keep any object referenced by kept objects

Stack		
Address	*Name*	*Contents*
10000	List L	&50000

Heap	
Address	*Contents*
50000	head = &50040
50008	value=0, next=&50024
50024	value=1, next=&50040
50040	value=2, next=&50056
50056	value=3, next=null



Mark, Sweep

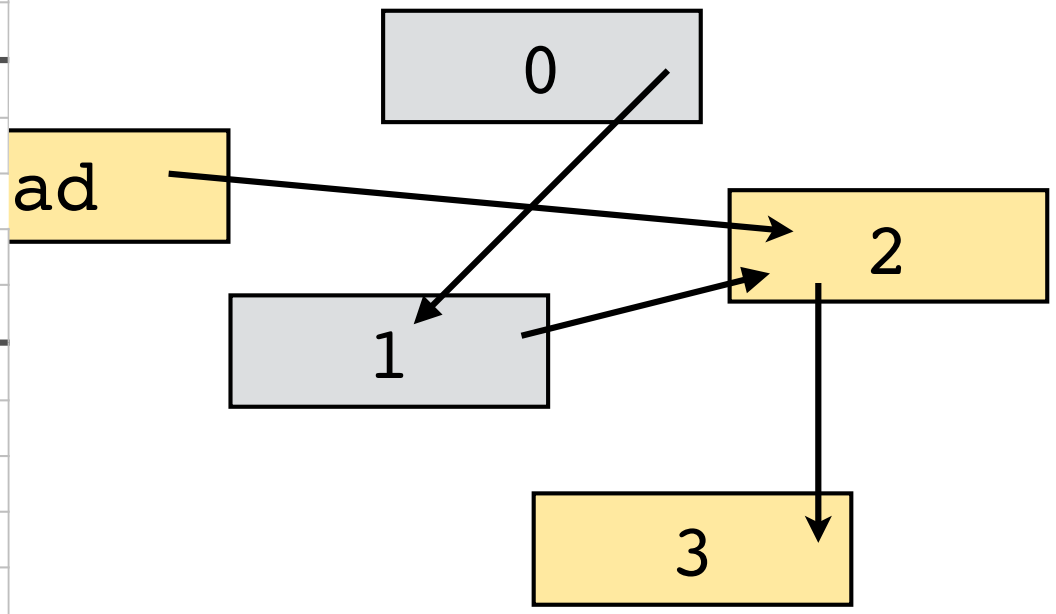
- Basic garbage collection algorithm
- Goal: find objects on the heap that are not referenced by any active object
- Maintain a list with a reference to all heap objects
  - Include a "referenced bit" with each object: 1=used, 0=lost
- Mark Phase:
  - Start from the root known object (e.g. top of stack)
  - Set referenced bit to 1 for every object it references
- Sweep Phase:
  - Step through list of all objects, delete anything not referenced

Garbage Collection

- The Java Run Time automatically tracks what objects are actively being used in memory
  - If no variable references something, then that object is "lost"---can be deleted

Stack		
Address	*Name*	*Contents*
10000	List L	&50000

Heap		
Address	*Reachable?*	*Contents*
50000	Y	head = &50040
50008	N	value=0, next=&50024
50024	N	value=1, next=&50040
50040	Y	value=2, next=&50056
50056	Y	value=3, next=null

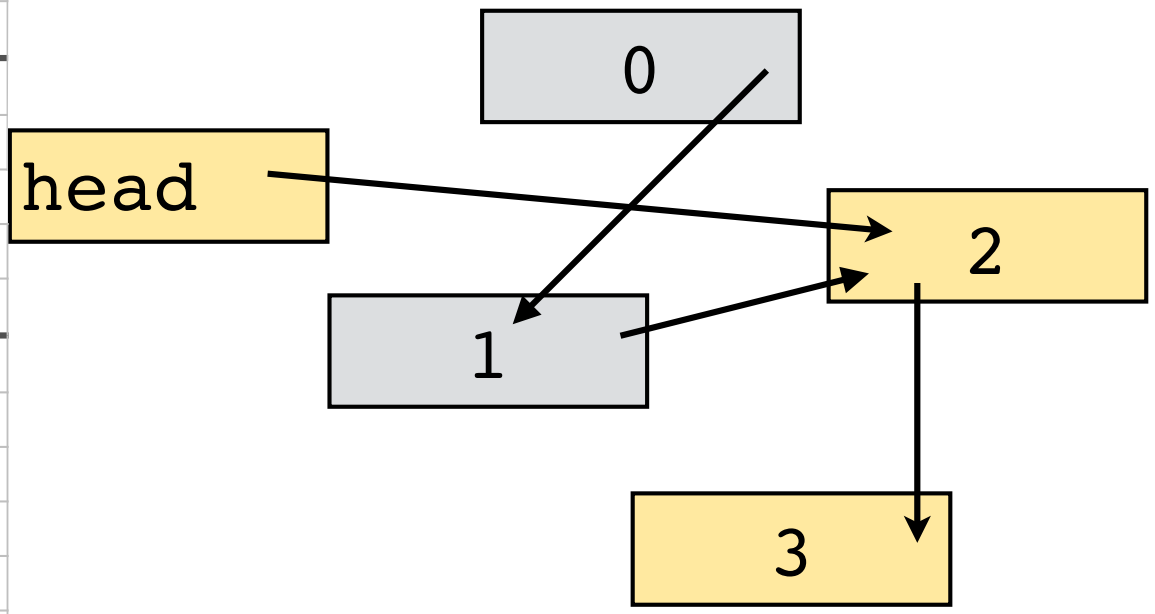


Garbage Collection

- The Java Run Time automatically tracks what objects are actively being used in memory
  - If no variable references something, then that object is "lost"---can be deleted

Stack		
Address	*Name*	*Contents*
10000	List L	&50000

Heap		
Address	*Reachable?*	*Contents*
50000	N	head = &50040
50008	N	value=0, next=&50024
50024	N	value=1, next=&50040
50040	N	value=2, next=&50056
50056	N	value=3, next=null



Benefits and Costs of GC

- Benefits:
  - No "memory leaks" from forgetting to free
  - Tighter control helps security
- Drawbacks:
  - May need to completely stop application while running garbage collection
  - Leads to unpredictable performance
  - Newer garbage collectors support parallelism
  - Just because there is a reference, doesn't mean it will be actively used again in the future

Data Collections

- The Java library already includes many data structures called Collections
- Java supports:
  - **Sets**: no-duplicates allowed, unordered
    - (a, b, c) === (b, c, a)
  - **Lists**: ordered list of elements
    - (a, b, c, b)
  - **Queues**: ordered list of elements waiting for processing
    - Optimized for accessing the ends of the list
  - **Maps:** store both a "key" and a matching "value" entity
    - Useful for looking up an object by a unique name, not position
- Many different implementations:
  - HashSet, ArrayList, LinkedList, Vector, PriorityQueue, Stack

Aside: Packages

- Packages are how java organizes classes
- Each package has its own folder on disk
  - Java classes that are part of a package must be in a folder of that name
- Packages can be nested
  - import java.util.ArrayList; ---> java/util/ArrayList.java
- You compile and run from the root of the tree:
  - javac java/util/ArrayList.java
  - javac filereader/RandReader.java
  - java filereader.RandReader

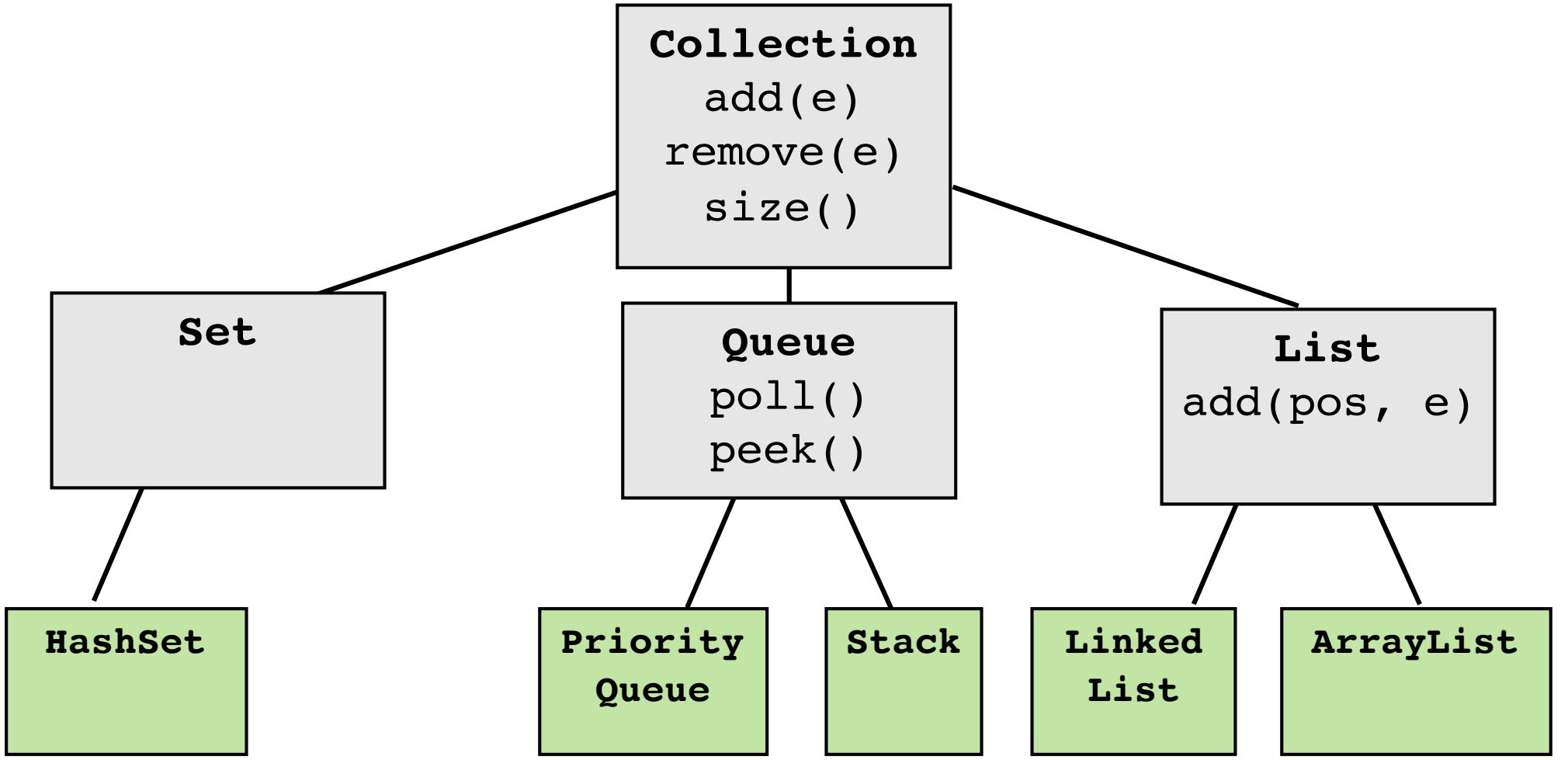
Use / when referring to folder
Use . when referring to package

- **For all exercises today you will compile and run from the lec-7 folder!**

Reading Data into a List

- ArrayList includes these functions:
  - add(Object o) -- add something to the list
  - get(int index) -- get the element at index
  - size() -- returns the number of elements in the list
- Look in the **filereader** package at **RandReader.java**
  - Make it add each line to an ArrayList
  - Print out a random line after all have been read in
  - Use Math.random() to get a double (0...1)
- Compile and run **from the lec-7/** directory:
  - `javac filereader/RandReader.java`
  - `java filereader.RandReader`

Java Collections Class Tree



(plus many other implementations)

Lists

- The most common collection type
- Do not use **List** class directly
  - Use: **ArrayList** or **Vector** (supports multi-threaded access)
- Basic usage:

```
public class ListTests {
    public static void main(String[] args) {
        ArrayList list = new ArrayList();
        SimpleElement e1 = new SimpleElement(1);
        SimpleElement e2 = new SimpleElement(2);
        SimpleElement e3 = new SimpleElement(3);

        list.add(e1);
        list.add(e2);
        list.add(e3);

        for (Object object : list) {
            System.out.println(((SimpleElement)object).value);
        }
    }
}
```

handy for loop iterator shortcut

cast from base Object type

Different Collections

- Open the **ListTests.java** file in the **listtests** package
- What does the code do?
  - What looks odd?
- What if you make **`list`** a new:
  - **`Vector`**
  - **`HashSet`**
  - **`Collection`**
- Compile and run **from the lec-7/** directory:
  - **`javac listtests/ListTests.java`**
  - **`java listtests.ListTests`**

Polymorphism

- Lets Java treat one object as another type
  - As long as it's a **subclass** or exposes the same **interface**

```
List list = new ArrayList();
   Vehicle v = new Car();
Vehicle v2 = new Airplane();


   Car c = new Vehicle();
  Airplane a = new Car();
```

Polymorphism

- Lets Java treat one object as another type
  - As long as it's a **subclass** or exposes the same **interface**

```
// good!
List list = new ArrayList();
   Vehicle v = new Car();
Vehicle v2 = new Airplane();

// bad!
   Car c = new Vehicle();
   Airplane a = new Car();
```

Collection Templates

- Collections support **generic** templates

```
Collection<SimpleElement> list =
      new ArrayList<SimpleElement>();

  for (SimpleElement se : list) {
    System.out.println(se.value);
  }
```

  - Lets you specify the type of object contained inside a list
  - Eliminates need to cast object returned by iterator
  - Prevents warnings from compiler

Stepping Through Lists

- Several ways to go through a list

- For each loop:

```
for (SimpleElement se : list)
  System.out.println(se.value);
```

- Formal iterator:

```
for (Iterator iterator = list.iterator(); iterator.hasNext();) {
  SimpleElement se = (SimpleElement) iterator.next();
  // ...
}
```

- Plain **for** loop: (only works for Lists, not sets or queues)

```
for(int i = 0; i < list.size(); i++) {
  SimpleElement se = list.get(i);
}
```