

CSE 1325: Object-Oriented Programming

Lecture 12

Polymorphism and Memory Management

Mr. George F. Rice

george.rice@uta.edu

Office Hours:

**Prof Rice 12:30 Tuesday and
Thursday in ERB 336**

For TAs [see this web page](#)

**Some drink at the fountain of knowledge.
Others just gargle.**



Exams are Graded and Posted

- Errata
 - Some minor font issues on 4b.
- Only 3 registered students failed to show up or contact me (3 other students have dropped)
- Grades are on Canvas
 - Graded exam attached as PDF file comment
 - Grades are final pending any appeals
 - Appeal via email ONLY to preserve permanent record
 - 2-week limit to *file* an appeal (decision may take longer)



Statistics and Such

- The exam was consistent in timing compared to recent semesters
 - 13% finished within an hour (16% last semester)
 - 29% finished within 70 minutes (30% last semester)
 - 36% finished before the end (46% last semester)
- Shortest exam time (all questions answered) was 51 minutes (45 minutes last semester)
- No specific problem areas were noted
 - Grades were slightly lower than typical for a first exam

Grade Distribution

with comparison to prior semesters

Overall, grades were somewhat lower than recent semesters.

- Vocabulary averaged 75% (86% / 90% over previous 2 semesters)
- Multiple choice averaged 70% (77% / 74% over previous 2 semesters)
- Coding averaged 62% (68% / 66% over last 2 semesters)

Vocabulary is particularly surprising, since students were given the exact answers more than a week before the exam. Only memorization was needed.

	2025	2024		2023		2022		2021
Min	14.0	21.5	34.5	43.0	31.0	38.5	41.0	35.5
Ave	74.8	80.5	78.3	83.7	77.9	78.4	77.2	78.2
Median	76.5	83.5	80.5	86.0	80.0	80.5	79.5	82.3
Max	108.0	107.0	99.5	103.5	107.0	103.0	101.5	101.0
Possible	107.0	107.0	106.0	105.0	107.0	107.0	105.0	107.0

Java

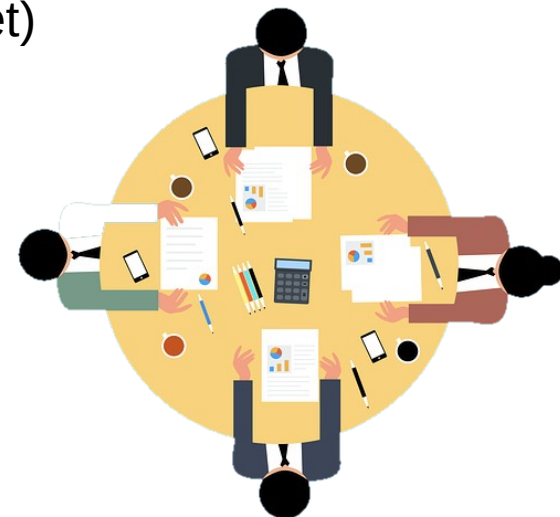


Highs and Lows

- Vocabulary was unusually low
- Coding skills were generally consistent
 - The toString and main methods were a little low (Free Response 2d and 3)
 - Inheritance and abstract classes were *excellent* (Free Response 4)
- Otherwise exam was as expected

Test Markings

- **Vocabulary** – Red “X” marks errors. +2 for each correct definition, 20 points total. Points earned are listed at bottom of page.
- **Multiple Choice** – Red “O” circles corrected answer. +2 for each correct choice, 10 points per page, 30 points total. Points earned are listed at bottom of page. WRITE ANSWER IN THE _____!
- **Free Response** – Corrections *often* marked in detail – this took a LOT of time, but hopefully you’ll READ and CAREFULLY CONSIDER each one! 50 points total.
 - Sum of points per *question* indicated beside each question on the page on which the answer was *asked* (NOT on an additional sheet)
- **Final Score** – The sum of all points on every page has been posted **on Canvas *only*** (NOT on the exam)
- **E1_Review.pdf** has been posted on Canvas, and the code used to write the exam is available at **cse1325-prof/Exam1/exam**





Overview

- Managing Memory
 - Garbage Collection
 - Reference Counting
 - Tracing
 - Memory Leaks
 - equals and hashCode
- Polymorphism

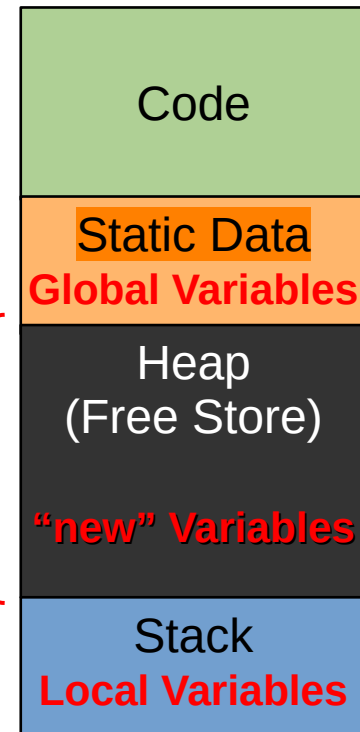


The Default JVM is Written in C++

C++ Memory Management

- **Stack** is “scratch memory” for a C++ thread
 - LIFO (Last-In, First-Out) allocation and deallocation
 - Allocated when a scope is entered
 - Automatically deallocated when that scope exits
 - Simple to track and so rarely leaks memory
- **Heap** is memory shared by all threads for dynamic allocation
 - Allocation and deallocation happens when requested by the C++ program
 - Requires manual deallocation
- Note: The Java Virtual Machine (JVM, the “java” command) is a C++ application

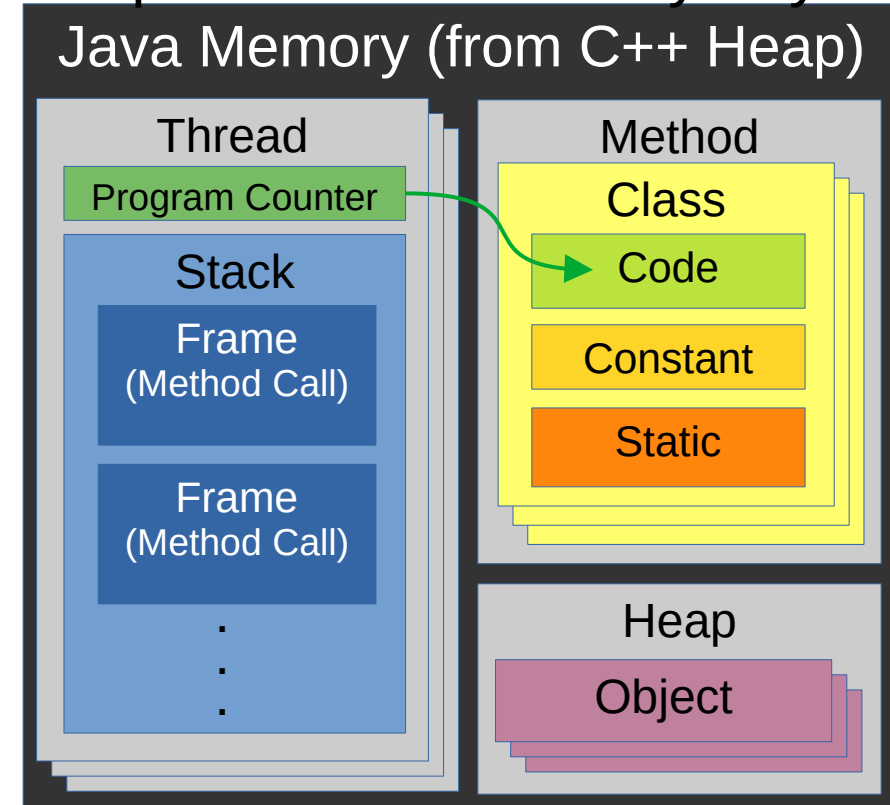
C++
Memory Layout



Java* Memory Management

- OpenJDK allocates a block of C++ heap as Java memory
- Each “Thread” is provided an area for **PC[†]** & **call frame** data
 - Multiple points of execution (threads) are supported by Java
- Each class is provided an area in “Method” for **code**, **constants**, and **static** variables
- Each object is provided an area in “Heap” for its non-static **fields**

Simplified Java Memory Layout



Deleting Memory Allocations

- “Bare metal” languages require care when allocating memory
 - In C and C++, we can just declare a *stack* variable, e.g., `int i=42;` Or `std::vector<std::string> vocabulary;`, and it “deletes itself” when the stack is popped
 - In C, we can also `malloc`, but we must later `free`
 - In C++, we can also `new`, but we must later `delete` (or `delete[]` if the heap variable is a true array)
 - Although C++ also has “smart pointers” to automatically delete heap allocations when no longer referenced. C++ is a bit complicated.
- Question: How to we free heap memory in Java?

Garbage Collection

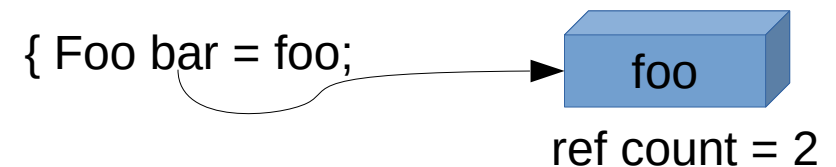
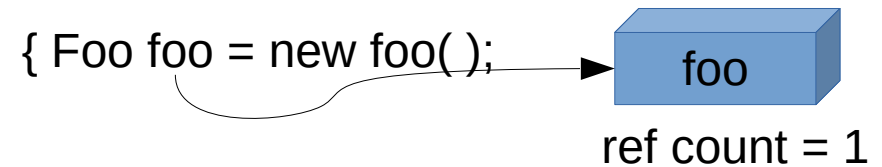
- In a Java program, an entity called the “garbage collector” (“recycler” would be better, I think) runs periodically
 - It finds and deletes every unreferenced object on the heap
 - Unreferenced objects can never be accessed again
 - So we can delete them without affecting our program’s logic
 - It consolidates memory so that unused memory is “big”
 - This avoid “memory fragmentation”, where unused heap memory is scattered among used heap memory
 - Memory fragmentation could keep us from allocating large objects on the heap in the future
- The garbage collector may run *at any time*
 - **You *cannot* make any assumptions about the GC!**
 - Important: `finalize()` is NOT a destructor! (And it’s deprecated)



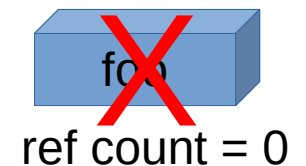
Reference Counting

- Some languages implement managed memory via reference counters

- When an object is instanced, memory is allocated and a “**reference counter**” is set to 1
- When another reference is created, its counter is incremented
- When a reference to that object is deleted (e.g., goes out of scope), its counter is decremented
- When more memory is needed, the “**garbage collector**” finds that the reference counter == 0, frees its memory, and moves in-use objects to free up a contiguous block of memory for future use



Do you see a flaw in this design?



Ref Counting Example

```
public class RefCounter {  
    public static void main(String[] args) {  
        AClass a = new AClass();           // ref count = 1  
        RefHolder r = new RefHolder(a);    // ref counter = 2  
        System.out.println("Objects: " + a + " and " + r);  
        a = null; // discard ref to AClass - ref counter = 1  
        System.out.println("Object: " + r);  
        r = null; // Discard RefHolder - ref counter = 0  
        System.gc(); // Request that the garbage collector run  
    }  
}
```

We store a *second* reference to the AClass object in the RefHolder object

```
class RefHolder {  
    public RefHolder(AClass aClass) {  
        this.aClass = aClass; // Add reference to AClass  
    }  
    @Override  
    public String toString() {  
        return super.toString() + " with RefHolder.aClass = " + aClass;  
    }  
    private AClass aClass;  
}  
class AClass {  
    public AClass() {  
        int anInt = 42;  
    }  
    private int anInt;  
}
```

We can observe the hash identity of each object using toString to verify that our AClass object survives main's setting the original reference to null

Ref Counting Example

Even after main.a is set to null, the AClass object survives because it is referenced by the RefHolder instance.

```
ricegf@antares:~/dev/202108/18$ javac RefCounter.java
ricegf@antares:~/dev/202108/18$ java RefCounter
Objects: AClass@7a81197d and RefHolder@5ca881b5 with RefHolder.aClass = AClass@7a81197d
Object: RefHolder@5ca881b5 with RefHolder.aClass = AClass@7a81197d
ricegf@antares:~/dev/202108/18$
```

Only when the reference to the RefHolder object is removed is the last reference to the Aclass object removed, allowing it to (eventually and potentially) be deleted by the garbage collector.

Of course, the garbage collector may *never* run.
We can make NO assumptions about the garbage collector *at all*.

Circular References

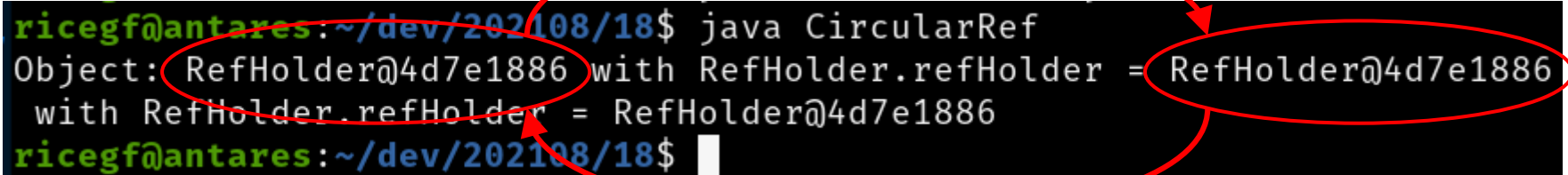
- A circular reference blocks simple reference counting
 - That is, an object that references itself *always* has at least one reference left! Uh oh...

```
public class CircularRef {  
    public static void main(String[] args) {  
        RefHolder r = new RefHolder(); // ref counter = 2  
        System.out.println("Object: " + r);  
        r = null; // Discard RefHolder ref - ref counter = 1  
    }  
}  
  
class RefHolder {  
    public RefHolder() {  
        this.refHolder = this; // Add reference to ourself  
        recursionCounter = 1;   // If we don't break recursion  
                                // when printing RefHolder.refHolder,  
                                // we'll enter an infinite loop and crash the stack.  
    }  
    private RefHolder refHolder;  
    @Override  
    public String toString() {  
        if(recursionCounter-- < 0) return super.toString();  
        return super.toString() + " with RefHolder.refHolder = " + refHolder;  
    }  
    private int recursionCounter;  
}
```

Refholder references itself!

Refholder *still* references itself!

Circular References



```
ricegfa@antares:~/dev/202108/18$ java CircularRef
Object: RefHolder@4d7e1886 with RefHolder.refHolder = RefHolder@4d7e1886
with RefHolder.refHolder = RefHolder@4d7e1886
ricegfa@antares:~/dev/202108/18$
```

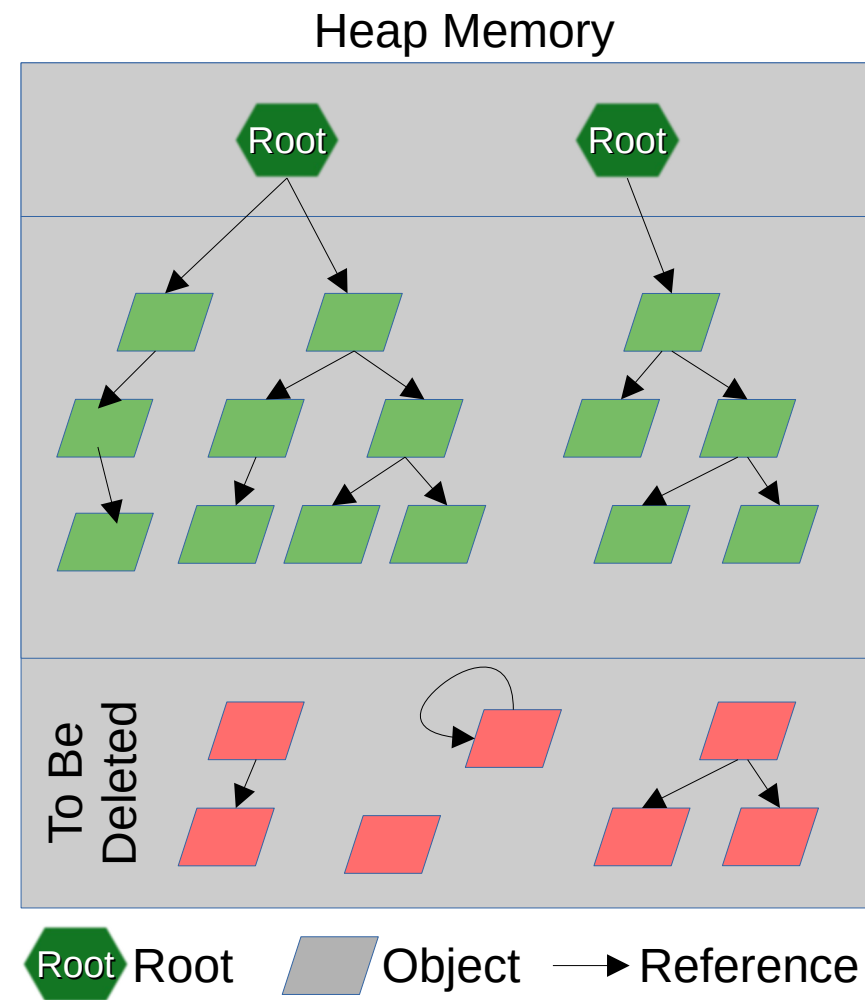
Just as we had to use `RefHolder.recursionCounter` to break out of the infinite loop, our garbage collector needs an algorithm to detect and ignore circular references so that each self-referencing object can be deleted when no longer accessible by any object *except itself*.

Java actually ignores reference counting entirely, and implements a different algorithm for garbage collection that always handles circular references correctly, called *tracing*.

Note: **Java 21** introduced an optional garbage collector named ZGC, also known as the “low-pause” garbage collector. It also ignores reference counting, but treats “young” objects likely to be deleted quickly differently than “old” objects which are likely to continue to be used. Fingers crossed for better performance!

Tracing

- Because of circular references, tracing is a popular alternative / complement to reference counters
 - First, mark all all objects as **eligible to be deleted**
 - Then, find the “root objects” known to be needed¹ and mark them **ineligible to be deleted**
 - Then, trace all references from the roots and mark all objects found as **ineligible to be deleted**
 - Finally, delete all objects still marked **eligible to be deleted**



¹This is too complicated to cover in this course.



Tracing

- Tracing algorithms intrinsically ignore circular references, because no reference trail reaches them from a root object
- The downside is that tracing uses a lot of cycles
 - Some implementations use “generational tracing”, essentially a “life counter” that segregates portions of the tree into “less likely to be eligible to be deleted” sections because they’ve survived many traces
 - “Less likely” portions are checked less frequently than newer objects
 - Like all engineering, this is a trade-off – less garbage collection overhead but less aggressive object deletion



Memory Management Strategies

- Direct
 - **C** uses malloc and free, which can be very tricky to get right.
 - **C++** native objects are explicitly deleted via destructors (if we are very careful!). “Smart pointers” delete referenced objects when the last smart pointer to them is itself deleted – much safer!
- Hybrid
 - **Python** immediately deletes an object when its reference count reaches 0, but also periodically and generationally traces references to detect and delete objects with circular references.
- Deferred
 - **Java** periodically traces all objects to detect and delete all unreferenced objects (Java 21 offers a generational gc, too). Java’s garbage collectors are frequently updated to improve performance.

4 Ways to Ensure a Java Object is Eligible for Garbage Collection

1) Reference the object in local scope

```
public void method(){MyObj a = new MyObj();} // object discarded on exit
```

- The object reference is a stack variable in a method
- When the method exits, the reference is popped and lost

2) Reassign the reference variable

```
MyObj a = new MyObj(); a = new MyObj(); // 1st object may be discarded  
// when 2nd is instanced
```

- The referencing variable is overwritten with a new reference

3) Nullify the reference variable

```
MyObj a = new MyObj(); a = null; // object may be discarded when a is null
```

- The referencing variable is overwritten with null

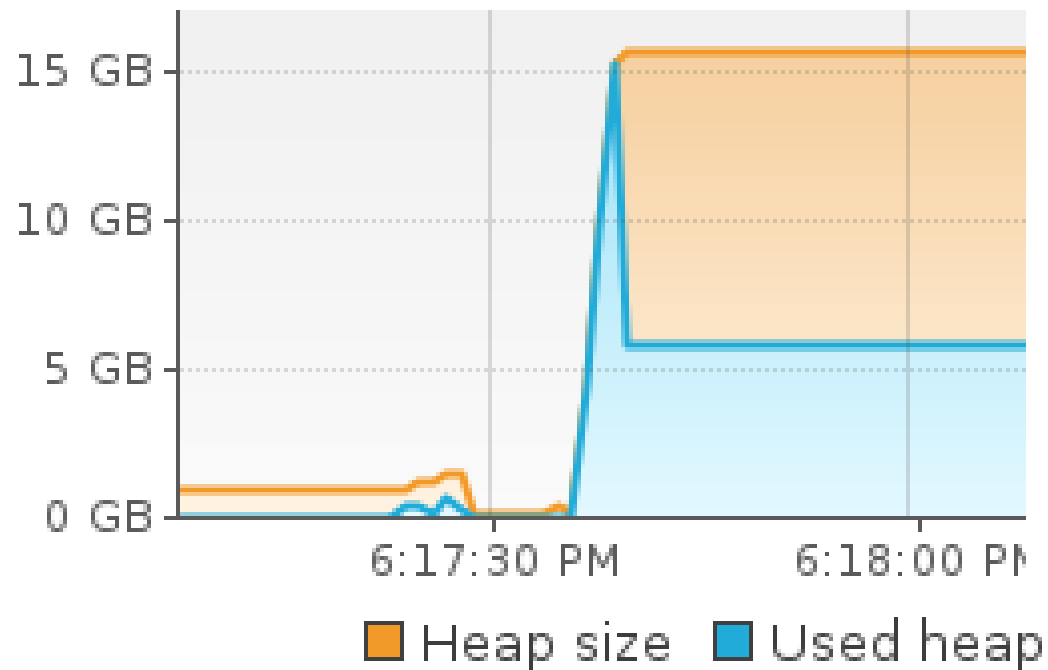
4) Use an anonymous object

```
(new MyObj()).usefulMethod(); // object may be discarded when method returns
```

- The reference is never stored

Memory Leaks in Java

- Garbage collection helps a lot
 - But we can still write leaky applications in Java
 - We need only allocate and not deallocate
- We'll look at a few common leaks



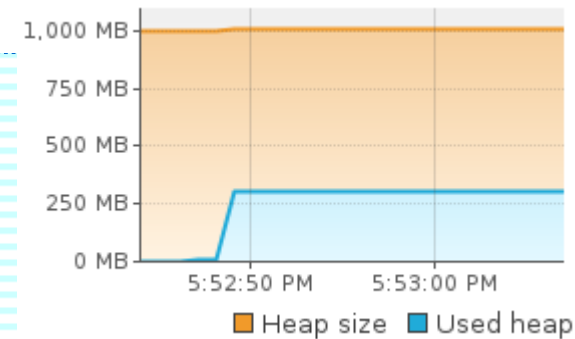
Find the Leak

- Where's the leak in class Logger?

```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.Random;

class Logger {
    public Logger() {
        Random random = new Random();
        for(int i=0; i<80000000; ++i)
            logs.add(random.nextDouble()); // Log a lot of data!
    }
    private static ArrayList<Double> logs = new ArrayList<>();
}

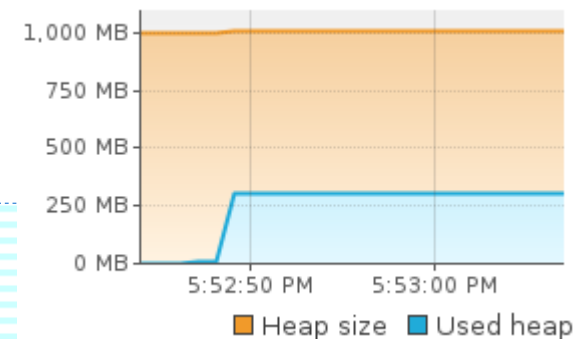
public class BadLogger {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Press Enter to continue:");
        String wait = sc.nextLine(); // Pause to configure profiler
        {
            Logger logger = new Logger(); // Looks innocent enough?
            System.out.print("Press Enter to continue:");
            wait = sc.nextLine(); // Collect some profiler data
        }
        System.out.print("Press Enter to continue:"); // Release Logger
        wait = sc.nextLine(); // Collect some more profiler data
    }
}
```



Logger's Leak

- The static ArrayList is never released
 - Static means it is maintained at the class level
 - It is independent of class instances
 - Our next instance of Logger will still have logs data
- Takeaway: Keep static fields rare and small

```
class Logger {  
    public Logger() {  
        Random random = new Random();  
        for(int i=0; i<8000000; ++i)  
            logs.add(random.nextDouble()); // Log a lot of data!  
    }  
    private static ArrayList<Double> logs = new ArrayList<>();  
}
```



Find the Leak

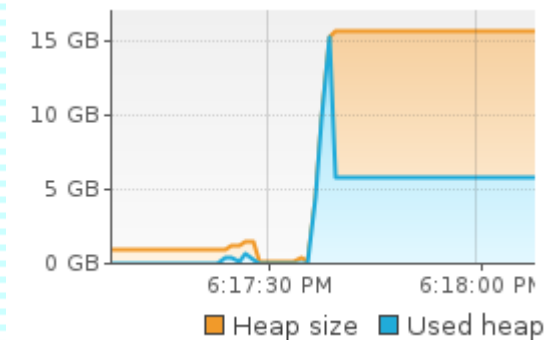
- Where's the leak in class BookFetcher?

```
// imports omitted
// main class BadReader is on the next page

class BookFetcher {
    public BookFetcher(String url) {
        this.url = url;
    }
    public String fetch()
        throws IOException, URISyntaxException {
        String str = "";
        URLConnection conn
            = new URL(url).openConnection();
        BufferedReader br = new BufferedReader(
            new InputStreamReader(conn.getInputStream(), StandardCharsets.UTF_8));

        while (br.readLine() != null)
            str += br.readLine();

        return str;
    }
    private String url;
}
```



Find the Leak

- Where's the leak in class BookFetcher?

```
// imports omitted
// class BookFetcher is on previous page

public class BadReader {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Press Enter to continue:");
        String wait = sc.nextLine();    // Wait for user to press Enter

        {
            BookFetcher remote = new BookFetcher("http://norvig.com/big.txt");
            try {
                String book = remote.fetch();
            } catch (Exception e) {
            }

            System.out.print("Press Enter to continue:");
            wait = sc.nextLine();    // Wait for user to press Enter
        }

        System.out.print("Press Enter to continue:");
        wait = sc.nextLine();    // Wait for user to press Enter
    }
}
```




BookFetcher's Leak

- BufferedReader is never closed
 - Its (substantial!) buffers hang around forever!
- The solution
 - Always use try-with-resources to automatically close all streams (similar to the garbage collector)

Find the Leak

- Code below adds 500 instances of me* to a HashMap
 - HashMaps use the first type for the index, second for the value
 - Identical index overwrites the existing value, just as with an array `a`, `a(1) = 50` overwrites the previous `a(1)`
- Where's the leak in class SSN?

```
import java.util.HashMap;

class SSN {
    public SSN(String social) {this.social = social;}
    private String social;
}

public class MapNaive {
    public static void main(String[] args) {
        HashMap<SSN, String> socials = new HashMap<>();
        for(int i=0; i<500; ++i)
            socials.put(new SSN("431-19-2021"), "Prof Rice");
        System.out.println("Size of map is " + socials.size());
    }
}
```

Hint: What will be the size of map `socials` at the end?

* Of course that's not my real social security number!

The Implications of hashMap

- What?
- The default equals() and hashCode() implementations make *each instance* of “me” a unique object
 - So a map gives each instance a separate slot
 - This is often not what we want (or expect)
 - This certainly uses a *lot* more memory for questionable gain
- Solution
 - *Always* override equals and hashCode in your objects

```
ricegf@antares:~/dev/202108/18$ javac MapNaive.java
ricegf@antares:~/dev/202108/18$ java MapNaive
Size of map is 500
ricegf@antares:~/dev/202108/18$ □
```

Object

(superclass for every class in Java)

Module java.base

Package java.lang

Class Object

java.lang.Object

Method Summary

All Methods	Instance Methods	Concrete Methods	Deprecated Methods
Modifier and Type	Method	Description	
protected Object	clone()	Creates and returns a copy of this object.	
boolean	equals(Object obj) ←	Indicates whether some other object is "equal to" this one.	
protected void	finalize()	Deprecated. The finalization mechanism is inherently problematic.	
final Class<?>	getClass()	Returns the runtime class of this Object.	
int	hashCode() ←	Returns a hash code value for the object.	

(not all members are shown here)



Equals() and hashCode()

- `Object.equals(Object)` is true only for the *same object at the same address*
 - By default, we inherit `Object.equals` in our own classes
 - This means `new Person("Prof Rice")` is NOT equal to `new Person("Prof Rice")`
 - Is this reasonable? Do we care?
- Similarly, the `hashCode` of this object is unique *per instance*
 - Is this reasonable? Do we care?
- Yes x 2! Because gross memory inefficiency can result from the default `equals()` and `hashCode()`
 - Particularly with `HashMap` and other containers that rely on an optimal definition for equality and hash

Adding Me Again 500 Times

- Let's override equals and hashCode
 - Now our “identity” depends on the field

```
class SSN {  
    public SSN(String social) {this.social = social;}  
    @Override  
    public boolean equals(Object o) {  
        if(this == o) return true;  
        if(o == null) return false;  
        if(this.getClass() != o.getClass()) return false;  
        SSN ssn = (SSN) o;  
        return social.equals(ssn.social);  
    }  
    @Override  
    public int hashCode() {  
        int hash = 7;  
        hash = 31*hash + (social == null ? 0 : social.hashCode());  
        return hash;  
    }  
    private String social;  
}
```

What will be the size of map
socials at the end this time?

SSN objects are equal iff they are of the
same type and have equal fields

SSN hashcodes are identical for identical
fields, thus equal SSN objects will always
have identical hashcodes

The Implications of hashMap

- Better
- The custom equals and hashCode make instances of “me” with equal fields equal
 - So the hashmap gives all instances a single slot
 - This is usually what we want (and expect)
 - This is certainly more memory efficient!
- For this reason, we should virtually *always* override equals and hashCode

```
ricegf@antares:~/dev/202108/18$ javac MapEquals.java
ricegf@antares:~/dev/202108/18$ java MapEquals
Size of map is 1
ricegf@antares:~/dev/202108/18$
```

Java Rules for Implementing equals and hashCode

- Given 3 objects a, b, and c, equals() must be:
 - Reflexive – `a.equals(a)`
 - Symmetric – `a.equals(b)` if and only if `b.equals(a)`
 - Transitive – if `a.equals(b)` and `b.equals(c)`
then `a.equals(c)`
- Our hashCode() must ensure:
 - 2 objects which are equals() have identical hashCode() values
(but 2 objects with identical hashCode() need NOT be equal!¹)
 - The same hashCode() value must be generated for an equal object
for the duration of execution, but MAY change between executions
- **Corollary:** Your implementation of equals() and hashCode()
must depend on exactly the same fields
 - Although “irrelevant” fields should be omitted from both

¹This is called a “hash collision”

Implementing Equals

A Cookbook Approach

- Equals compares to an Object
 - We could, *in theory*, compare SSN to another type BUT
 - Symmetry: if a.equals(b) then b.equals(a)
 - Transitive: If a.equals(b) and b.equals(c) then a.equals(c)
 - We need to test and cast to the right type regardless
 - The first 3 logical lines below are typical boilerplate, with the last line simply comparing the “significant” fields (those relevant to equality), in this case, social

```
@Override
public boolean equals(Object o) {
    if(this == o) return true;           // Is it me?
    if(o == null || this.getClass() != o.getClass())
        return false;                   // Is it my type?
    SSN ssn = (SSN) o;                  // Cast to my type
    return social.equals(ssn.social);    // Compare all relevant fields
}
```

Implementing hashCode

A Cookbook Approach

- hashCode is an int representation of an object
 - Many are possible – but this one works fairly well
 - Start with `int hash = 7` – 7 & 31 are prime numbers
 - For each field, multiply hash by 31 and then
 - Add a unique integer representation of a primitive field OR
 - Add the hashCode of an object field (or 0 if null)

```
@Override
public int hashCode() {
    int hash = 7;
    hash = 31*hash + (social == null ? 0 : social.hashCode());
    return hash;
}
```


Implementing hashCode

Using Objects (starting with Java 8)

- Java.util.Objects includes a hash generator
 - Can't get any easier than this
 - Simply list the relevant fields

```
@Override
public int hashCode() {
    return Objects.hash(social); // list all relevant fields as parameters
}
```

Note: Most IDEs can auto-generate equals() and hashCode() implementations.

You will not have an IDE on the exam!!!

So write them by hand for all assignments and save automation until you “get it”

Another hashCode Example

Using Objects (starting with Java 8)

```
Class Complex {  
    // other code here  
    @Override  
    public int hashCode() {  
        return Objects.hash(x, y); // list all relevant fields as parameters  
    }  
    private int x;  
    private int y;  
}
```

hash

```
public static int hash(Object... values)
```

Generates a hash code for a sequence of input values. The hash code is generated as if all the input values were placed into an array, and that array were hashed by calling `Arrays.hashCode(Object[])`.

This method is useful for implementing `Object.hashCode()` on objects containing multiple fields. For example, if an object that has three fields, `x`, `y`, and `z`, one could write:

```
@Override public int hashCode() {  
    return Objects.hash(x, y, z);  
}
```

Warning: When a single object reference is supplied, the returned value does not equal the hash code of that object reference. This value can be computed by calling `hashCode(Object)`.

Parameters:

values - the values to be hashed

Returns:

a hash value of the sequence of input values

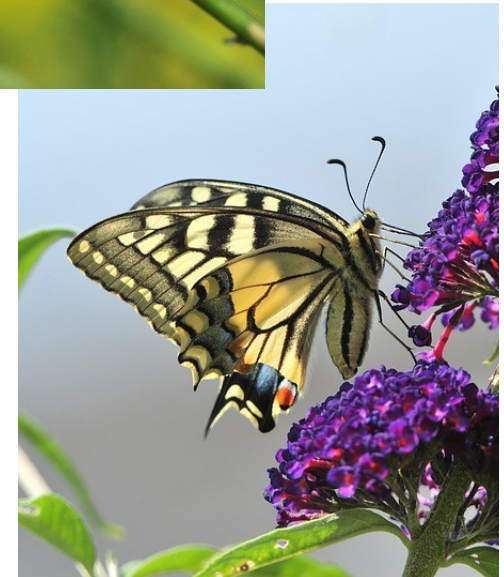
See Also:

`Arrays.hashCode(Object[])`, `List.hashCode()`



Polymorphism

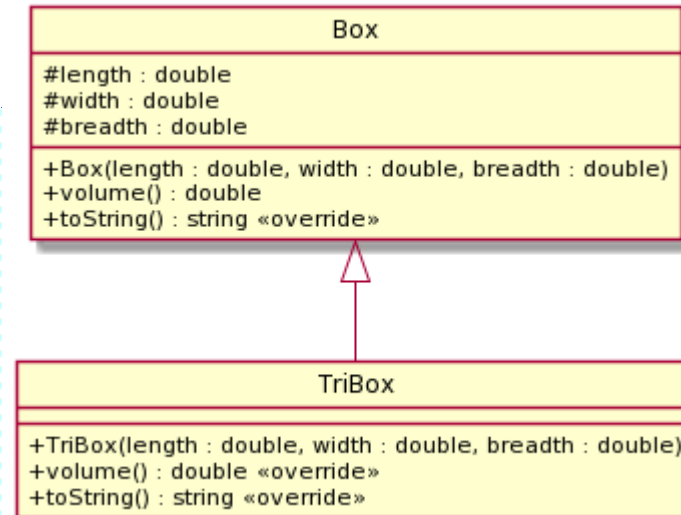
- **Polymorphism** – The provision of a single interface to multiple derived classes, enabling the same method call to invoke different derived methods to generate different results.
- In Java (unlike C++), no special coding techniques are required.
 - Although have I mentioned **@Override** on the polymorphic methods is both *wise* and *expected* in CSE1325?



The Box Class

- Consider a rectangular box

```
public class Box {  
    public Box(double length, double width, double height) {  
        this.length = length;  
        this.width = width;  
        this.height = height;  
    }  
    public double volume() {return length * width * height;}  
  
    @Override  
    public String toString() {  
        return "Rectangular box ("  
            + length + " x " + width + " x " + height + ")";  
    }  
  
    protected double length;  
    protected double width;  
    protected double height;  
}
```



“protected” means that classes derived from `Box` may read or modify (but NOT INITIALIZE) this field. Only the superclass in which it is defined may initialize it!

The Box Class

with equals and hashCode implementations

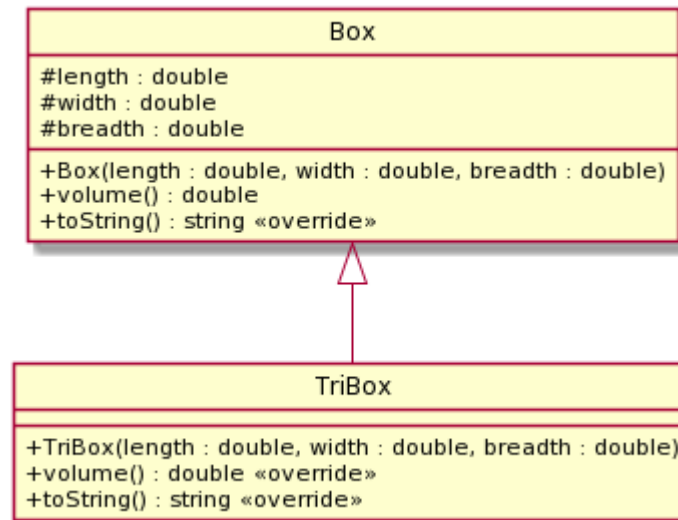
```
import java.util.Objects;

public class Box {
    public Box(double length, double width, double height) {
        this.length = length; this.width = width; this.height = height;
    }
    public double volume() {return length * width * height;}

    @Override public String toString() {
        return "Rectangular box ("
            + length + " x " + width + " x " + height + ")";
    }

    @Override public int hashCode() {
        return Objects.hash(length, width, height);
    }

    @Override public boolean equals(Object o) {
        if(o == this) return true;
        if(o == null || o.getClass() != this.getClass()) return false;
        Box b = (Box) o;
        return (b.length == length) && (b.width == width) && (b.height == height);
    }
    protected double length;
    protected double width;
    protected double height;
}
```



We note and ignore the potential round-off issues here

What About a TriBox?

- TriBox could reuse some Box code
 - The same fields as a Box
 - The same constructor parameters
 - The same methods and signatures
- It is likely we would want to store both Boxes and TriBoxes in a Java container of Boxes
- For these reasons – reuse and shared containers – inheritance may be a good choice

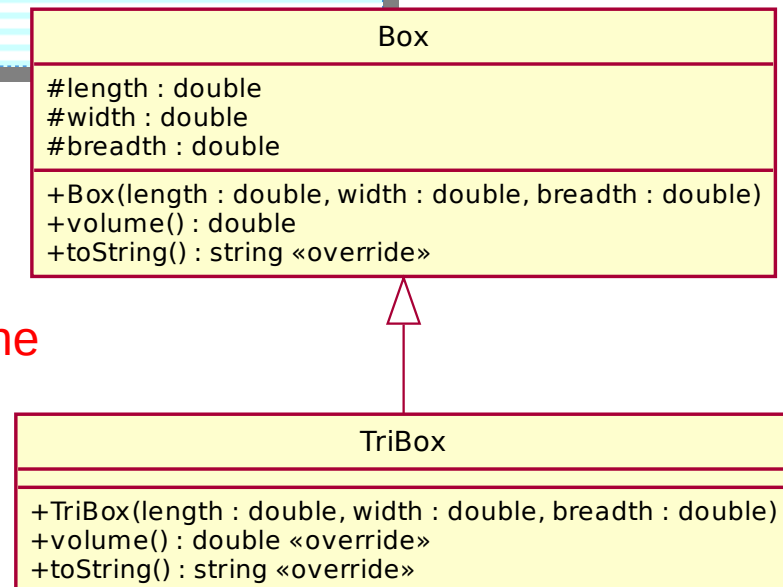


Inheriting From (“Extending”) the Box Superclass

```
public class TriBox extends Box {  
    public TriBox(double length, double width, double height) {  
        super(length, width, height);  
    }  
    @Override  
    public double volume() {return 0.5 * length * width * height;}  
    @Override  
    public String toString() {  
        return "Triangular box (" +  
            length + " x " + width + " x " + height + ")";  
    }  
    // Box.equals and Box.hashCode work for TriBox  
}
```

Since constructors don't inherit, we explicitly call (“delegate to”) the superclass' constructor (reuse). This must be the FIRST line in the constructor!

We override `volume()` and `toString()`. ALWAYS use the `@Override` annotation to tell the compiler to warn you if your superclass has no method of the same name and parametric signature to override.
It will save you HOURS of debugging!



Testing Box and Tribox

```
class Boxes {
    public static void main(String[] args) {
        // Let's try a couple of rectangular boxes
        Box box1 = new Box( 6,  7,  5);
        Box box2 = new Box(12, 13, 10);

        System.out.println("Volume of " + box1 + " is " + box1.volume());
        System.out.println("Volume of " + box2 + " is " + box2.volume());

        // Let's try a couple of triangular boxes
        TriBox box3 = new TriBox( 6,  7,  5);
        TriBox box4 = new TriBox(12, 13, 10);

        System.out.println("Volume of " + box3 + " is " + box3.volume());
        System.out.println("Volume of " + box4 + " is " + box4.volume());
    }
}
```

```
ricegfa@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ javac Boxes.java
ricegfa@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ java Boxes
Volume of Rectangular box (6.0 x 7.0 x 5.0) is 210.0
Volume of Rectangular box (12.0 x 13.0 x 10.0) is 1560.0
Volume of Triangular box (6.0 x 7.0 x 5.0) is 105.0
Volume of Triangular box (12.0 x 13.0 x 10.0) is 780.0
ricegfa@antares:~/dev/cse1325-prof/18/code_from_slides/Box$
```

Here we create objects of the derived class and use them directly.

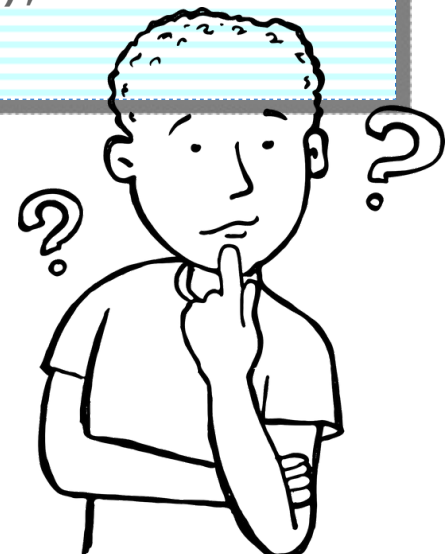
Some of the features of the base class inherit to derived scope – the protected data, for example – and some do not – the volume() method. We could also add additional fields and methods to the derived class as needed.

(Trying to) Store Extended Object in Superclass Variable

```
class Boxes {  
    public static void main(String[] args) {  
        // Let's try a couple of rectangular boxes  
        Box box1 = new Box( 6,  7,  5);  
        Box box2 = new Box(12, 13, 10);  
  
        System.out.println("Volume of " + box1 + " is " + box1.volume());  
        System.out.println("Volume of " + box2 + " is " + box2.volume());  
  
        // Let's try a couple of rectangular boxes  
        Box box3 = new TriBox( 6,  7,  5);  
        Box box4 = new TriBox(12, 13, 10);  
  
        System.out.println("Volume of " + box3 + " is " + box3.volume());  
        System.out.println("Volume of " + box4 + " is " + box4.volume());  
    }  
}
```

What if we try to store a TriBox (subclass object) in a Box variable (superclass variable)? Will we get

- A compiler error
- Output identical to a Box object
- Correct output for a TriBox object?



Storing an Extended Object in a Superclass Variable

```
class Boxes {  
    public static void main(String[] args) {  
        // Let's try a couple of rectangular boxes  
        Box box1 = new Box( 6,  7,  5);  
        Box box2 = new Box(12, 13, 10);  
  
        System.out.println("Volume of " + box1 + " is " + box1.volume());  
        System.out.println("Volume of " + box2 + " is " + box2.volume());  
  
        // Let's try a couple of rectangular boxes  
        Box box3 = new TriBox( 6,  7,  5);  
        Box box4 = new TriBox(12, 13, 10);  
  
        System.out.println("Volume of " + box3 + " is " + box3.volume());  
        System.out.println("Volume of " + box4 + " is " + box4.volume());  
    }  
}
```

No problems!
As long as the
object type is a
subclass of the
variable type, we may assign the reference without issues.

```
ricegfa@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ javac Boxes.java  
ricegfa@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ java Boxes  
Volume of Rectangular box (6.0 x 7.0 x 5.0) is 210.0  
Volume of Rectangular box (12.0 x 13.0 x 10.0) is 1560.0  
Volume of Triangular box (6.0 x 7.0 x 5.0) is 105.0  
Volume of Triangular box (12.0 x 13.0 x 10.0) is 780.0  
ricegfa@antares:~/dev/cse1325-prof/18/code_from_slides/Box$
```



Upcasting

- Storing a subclass object in a superclass variable is called “upcasting”
 - No explicit cast syntax is required

```
TriBox t = new TriBox(3,4,5);  
Box b = t; // works just fine
```
 - Only methods defined in the variable type are available in an upcast variable
 - If TriBox had added a method named foo() not defined in Box, then
 - `t.foo()` would work just fine
 - `b.foo()` would generate a compiler error
- Upcasting is *very* common in polymorphic programs

(Trying to) Store Mixed Objects in an ArrayList

```
import java.util.ArrayList;

public class BoxesArray {
    public static void main(String[] args) {
        ArrayList<Box> boxes = new ArrayList<>() {{
            add(new Box( 6,  7,  5));
            add(new Box(12, 13, 10));
            add(new TriBox( 6,  7,  5));
            add(new TriBox(12, 13, 10));
        }};

        for(Box box : boxes)
            System.out.println("Volume of " + box + " is " + box.volume());
    }
}
```

What if we try to store both Boxes and TriBoxes in a Box container (e.g., ArrayList)? Will we get

- A compiler error
- Output identical to a Box object
- Correct output for a TriBox object?



Storing Mixed Objects in an ArrayList

```
import java.util.ArrayList;

public class BoxesArray {
    public static void main(String[] args) {
        ArrayList<Box> boxes = new ArrayList<>() {{
            add(new Box( 6,  7,  5));
            add(new Box(12, 13, 10));
            add(new TriBox( 6,  7,  5));
            add(new TriBox(12, 13, 10));
        }};

        for(Box box : boxes)
            System.out.println("Volume of " + box + " is " + box.volume());
    }
}
```

Still no problem!

We may use a subclass object in virtually any

place we could use a superclass object. The initialization is another example of upcasting. Upcasting enables **polymorphism**.

```
ricegff@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ javac BoxesArray.java
ricegff@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ java BoxesArray
Volume of Rectangular box (6.0 x 7.0 x 5.0) is 210.0
Volume of Rectangular box (12.0 x 13.0 x 10.0) is 1560.0
Volume of Triangular box (6.0 x 7.0 x 5.0) is 105.0
Volume of Triangular box (12.0 x 13.0 x 10.0) is 780.0
ricegff@antares:~/dev/cse1325-prof/18/code_from_slides/Box$
```


Downcasting

- Storing a subclass object referenced by a superclass (or interface) variable in a subclass variable is called “downcasting”

- C-like casting is required

```
Box b = new TriBox(3,4,5);  
TriBox t = (TriBox) b; // Explicit cast is required
```

- If the superclass variable is not referencing an object of the subclass type, an exception will be thrown

- The instanceof operator is helpful to verify types at runtime

```
Box b = new TriBox(3,4,5);  
TriBox t;  
if(b instanceof TriBox) t = (TriBox) b;  
else t = null;
```

- Starting in Java 16, instanceof can auto-cast for you!

```
Box b = new TriBox(3,4,5);  
if(b instanceof TriBox t) System.out.println(t.toString());
```

How to Downcast

```
import java.util.ArrayList;

public class BoxesDowncast {
    public static void main(String[] args) {
        ArrayList<Box> boxes = new ArrayList<>() {{
            add(new Box( 6,  7,  5));
            add(new Box(12, 13, 10));
            add(new TriBox( 6,  7,  5));
            add(new TriBox(12, 13, 10));
        }};

        for(Box box : boxes) {
            try {
                TriBox tb = (TriBox) box;
                System.out.println("Downcast " + tb);
            } catch(ClassCastException e) {
                System.err.println("Failed to downcast " + box + ": " + e.getMessage());
            }
        }
    }
}
```

Here we add 2 Box and 2 TriBox objects to an ArrayList<Box>. Then we try to cast each element to a Tribox.

Will we get

- A compiler error
- One or more exceptions (How many? When?)
- *Still* no problem!



How to Downcast

```
import java.util.ArrayList;

public class BoxesDowncast {
    public static void main(String[] args) {
        ArrayList<Box> boxes = new ArrayList<>() {{
            add(new Box( 6,  7,  5));
            add(new Box(12, 13, 10));
            add(new TriBox( 6,  7,  5));
            add(new TriBox(12, 13, 10));
        }};

        for(Box box : boxes) {
            try {
                TriBox tb = (TriBox) box;
                System.out.println("Downcast " + tb);
            } catch(ClassCastException e) {
                System.err.println("Failed to downcast " + box + ": " + e.getMessage());
            }
        }
    }
}
```

We are attempting to downcast both Box and TriBox instances to a TriBox. TriBox instances correctly downcast, while Box instances throw an (unchecked) ClassCastException.

```
ricegfa@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ java BoxesDowncast
Failed to downcast Rectangular box (6.0 x 7.0 x 5.0): class Box cannot be cast to
class TriBox (Box and TriBox are in unnamed module of loader 'app')
Failed to downcast Rectangular box (12.0 x 13.0 x 10.0): class Box cannot be cast
to class TriBox (Box and TriBox are in unnamed module of loader 'app')
Downcast Triangular box (6.0 x 7.0 x 5.0)
Downcast Triangular box (12.0 x 13.0 x 10.0)
ricegfa@antares:~/dev/cse1325-prof/18/code_from_slides/Box$
```



What We Learned Today

- Mainstream Java virtual machines use “**deferred tracing**” rather than “reference counting” for garbage collection
- Override **equals** and **hashCode** for virtually ALL Java classes
 - Equals must be reflexive, symmetric, and transitive
 - Equal objects must generate the same hashCode (but not vice-versa)
- **Polymorphism** is the provision of a single interface to multiple derived classes, enabling the same method call to invoke different derived methods to generate different results.
 - **Upcasting** *assigns* a subclass object to a superclass variable
 - **Downcasting** *casts* a superclass variable to a subclass variable, throwing an (unchecked) `ClassCastException` if the superclass variable doesn't reference an instance of that subclass