Review Session

CS2110 Prelim #1

Primitive types vs classes

- Variabledeclarations:
  - `int i = 5;`
  - `Animal a = new Animal("Bob");`
- How does "==" behave?

Default values

- What value does a field contain when it is declared but not instantiated?
  - `Animal a;` `//null`
  - `Object ob;` `//null`
  - `int i;` `//0`
  - `boolean b;` `//false`
  - `char c;` `//'\0' (null byte)`
  - `double d;` `//0.0`

Wrapper Classes (Boxing)

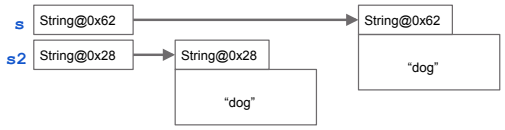
class Character contains useful methods

- Examples of useful static `Character` methods:
  - `Character.isDigit(c)`
  - `IntCharacter.isLetter(c)`
- Autoboxing –should be called autowrapping!
  - `Integer x = 100;`
  - `int y = x;`

String literals

String instantiation:

- Constructor: `String s = new String("dog");`
- Literal: `String s2 = "dog";`
- Roughly equivalent, but literal is preferred



Strings are immutable

Once a String is created, it cannot be changed

- Methods such as `toLowerCase` and `substring` return new Strings, leaving the original one untouched
- In order to "modify" Strings, you instead construct a new String and then reassign it to the original variable:
  - `String name = "Gries";`
  - `name = name + ", ";`
  - `name = name + "David";`

String catenation

Operator `+` operator is called catenation, or concatenation

- If one operand is a String and the other isn't, the other is converted to a String
- Important case: Use `"" + exp` to convert `exp` to a String.
- Evaluates left to right. Common mistake:
  - `System.out.println("sum: " + 5 + 6);`
    - Prints `"sum: 56"`
  - `System.out.println("sum: " + (5 + 6));`
    - Prints `"sum: 11"`

Other String info

- Always use `equals` to compare Strings:
  - `str1.equals(str2)`
- Very useful methods:
  - `length`, `substring` (overloaded), `indexOf`, `charAt`
- Useful methods:
  - `lastIndexOf`, `contains`, `compareTo`

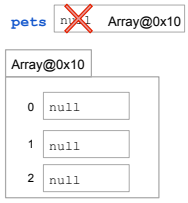
1D Array Review

```
Animal[] pets = new Animal[3];
```

```
pets.length is 3
pets[0] = new Animal();
pets[0].walk();
```

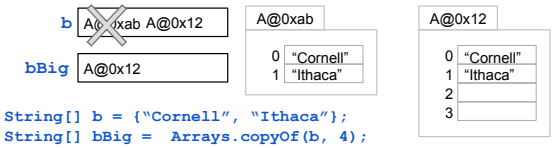
Why is the following illegal?

```
pets[1] = new Object();
```



Java arrays

Java arrays do not change size!



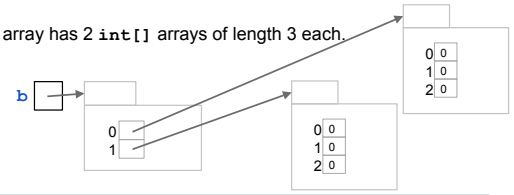

```
String[] b = {"Cornell", "Ithaca"};
String[] bBig =  Arrays.copyOf(b, 4);
b = bBig;
```


2D arrays: An array of 1D arrays.

Java only has 1D arrays, whose elements can also be arrays.

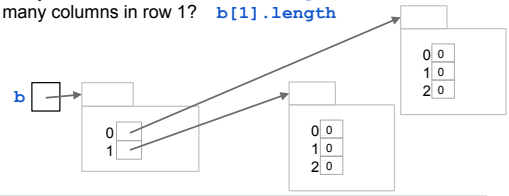
```
int[][] b = new int[2][3];
```

This array has 2 `int[]` arrays of length 3 each.



2D arrays: An array of 1D arrays.

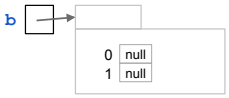
How many rows in `b`? `b.length`
How many columns in row 0? `b[0].length`
How many columns in row 1? `b[1].length`



2D arrays: An array of 1D arrays.

```
int[][] b = new int[2][];
```

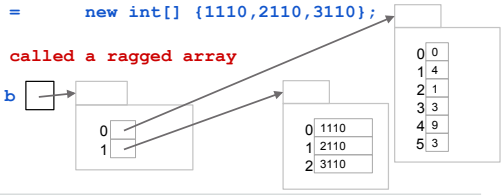
The elements of b are of type `int[]`.



2D arrays: An array of 1D arrays.

```
int[][] b = new int[2][];
b[0] =      new int[] {0,4,1,3,9,3};
b[1] =      new int[] {1110,2110,3110};
```

b is called a ragged array



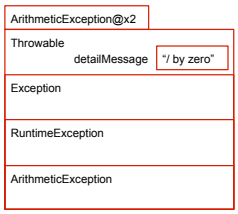
The superclass of exceptions: Throwable

`class Throwable:`
- Superclass of Error and Exception
- Does the "crashing"
- Contains the constructors and methods
- `Throwable()`
- `Throwable(String)`

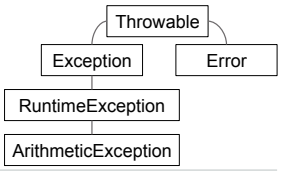
`class Error:`
- A very serious problem and should not be handled
  Example: StackOverflowError

`class Exception:`
- Reasonable application might want to crash or handle the Exception in some way

A Throwable instance: ArithmeticException



There are so many exceptions we need to **organize** them.



Bubbling up exceptions

Exceptions will bubble up the call stack and crash the methods that called it.

Method call: `first();`

`Console:`
```
Exception in thread "main"
  java.lang.ArithmeticException:
        at Ex.third(Ex.java:11)
        at Ex.second(Ex.java:7)
        at Ex.first(Ex.java:3)
```


```
1  class Ex {
2      void first() {
3              second();
4          }
5
6          void second() {
7              third();
8      }
9
10     void third() {
11             int c = 5/0;
12     }
13 }
```

AE = ArithmeticException

Try-catch blocks

An exception will bubble up the call stack and crash the methods that called it
… unless it is caught.

`catch` will handle any exceptions of type *Exception* (and its subclasses) that happened in the `try` block

`Console:`
```
in
error
```

```
1  class Ex {
2      void first() {
3              second();
4          }
5          void second() {
6              try {
7
8      System.out.println("in");
9              third();
10
11     System.out.println("out");
12             } catch (Exception e){
13
14     System.out.print("error");
15             }
16 }
17
18     void third() {
           int c = 5/0;
```

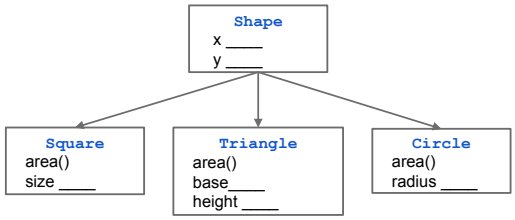
Exception Type

ArithmeticException!

How to write an exception class

```
/** An instance is an exception */
public class OurException extends Exception {

        /** Constructor: an instance with message m*/
        public OurException(String m) {
        super(m);
        }

        /** Constructor: an instance with default message */
        public OurException() {
        this("Default message!");
        }
}
```


A Little More Geometry!



A Partial Solution:

Add method area to class Shape:

```
public double area() {
        return 0;
}

public double area() {
        throw new RuntimeException("area not
overridden");
}
```


Problems not solved

1. What is a Shape that isn't a Circle, Square, Triangle, etc? What is *only* a shape, nothing more specific?
   `a. Shape s = new Shape(...);` Should be disallowed

2. What if a subclass doesn't override area()?
   a. Can't force the subclass to override it!
   b. Incorrect value returned or exception thrown.

Solution: Abstract classes

Abstract class
Can't be instantiated.
(`new Shape()` illegal)

```
public abstract class Shape {

      public double area() {
      return 0;
   }
}
```


Solution: Abstract methods

```
public abstract class Shape {

      public abstract double area();
}
```

Abstract method
Subclass must override.

- Can have implemented methods, too
- Place abstract method only in abstract class.
- Semicolon instead of body.

Abstract Classes, Abstract Methods

1. **Cannot instantiate an object of an abstract class.**
   (Cannot use new-expression)

1. **A subclass must override abstract methods.**

(but no multiple inheritance in Java, so…)

Interfaces

```
public interface Whistler {
      void whistle();
      int MEANING_OF_LIFE= 42;
}

class Human extends Mammal implements Whistler {
}
```

- methods are automatically `public` and `abstract`
- fields are automatically `public`, `static`, and `final` (i.e. constants)

Must implement all methods in the implemented interfaces

Multiple interfaces

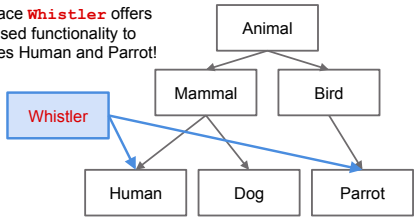
```
public interface Singer {
      void singTo(Human h);
}

class Human extends Mammal implements Whistler, Singer {
}
```

Classes can implement several interfaces! They must implement all the methods in those interfaces they implement.

Must implement `singTo(Human h)` and `whistle()`

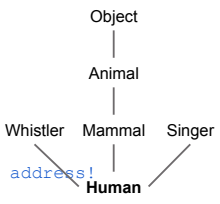
Solution: Interfaces

Interface **`Whistler`** offers promised functionality to classes Human and Parrot!



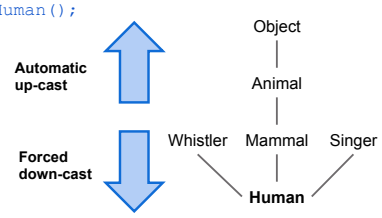
Casting

```
Human h  = new Human();
Object o = (Object) h;
Animal a = (Animal) h;
Mammal m = (Mammal) h;

Singer s = (Singer) h;
Whistler w = (Whistler) h;

All point to the same memory address!
```



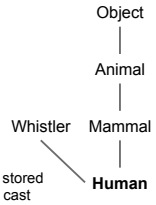
Casting

```
Human h  = new Human();
Object o = h;
Animal a = h;
Mammal m = h;
Singer s = h;
Whistler w = h;
```



Casting up to an interface automatically

```
class Human … implements Whistler {
   void listenTo(Whistler w) {...}
}
Human h = new Human(...);
Human h1 = new Human(...);
h.listenTo(h1);
Parrot p = new Parrot(...);
h.listenTo(p);
```



Arg h1 of the call has type Human. Its value is being stored in w, which is of type Whistler. Java does an upward cast automatically. Same thing for p of type Parrot.

Shape implements `Comparable<T>`

```
public class Shape implements Comparable<Shape> {
    ...
    /** … */
    public int compareTo(Shape s) {
       double diff= area() - s.area();
       return (diff == 0 ? 0 : (diff < 0 ? -1 : +1));
    }
}
```


Beauty of interfaces

`Arrays.sort` sorts an array of *any* class C, as long as C implements interface **`Comparable<T>`** without needing to know any implementation details of the class.

Classes that implement Comparable:

`Boolean`	`Byte`	`Double`	`Integer`
`String`	`BigDecimal`	`BigInteger`	`Calendar`
`Time`	`Timestamp`	and 100 others	

String sorting

`Arrays.sort(Object[] b)` sorts an array of *any* class C, as long as C implements interface **`Comparable<T>`**.

`String` implements `Comparable`, so you can write

```
String[] strings= ...;  ...
Arrays.sort(strings);
```

During the sorting, when comparing elements, a String's compareTo function is used

Abstract Classes vs. Interfaces

- Abstract class represents something
- Sharing common code between subclasses

- Interface is what something can do
- A contract to fulfill
- Software Engineering purpose

Similarities:
- Can't instantiate
- Must implement abstract methods

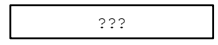
Four loopy questions

```
//Precondition
Initialization;
// invariant: P
while ( B ) { S }
```

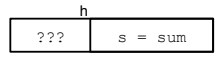
1. Does it **start** right? Does initialization make invariant P true?
2. Does it **stop** right? Does P and !B imply the desired result?
3. Does repetend S make **progress** toward termination?
4. Does repetend S **keep** invariant P true?

Add elements backwards

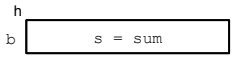
Precondition



Invariant

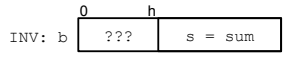


Postcondition



Add elements backwards

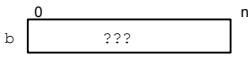
```
int s = 0;
int h = b.length-1;
while (h >= 0) {
      s = s + b[h];
      h--;
}
```



1. Does it **start** right?
2. Does it **stop** right?
3. Does it **keep** the invariant true?
4. Does it make **progress** toward termination?

Linear search time

Linear search for v in an array b of length n



worst-case time. v is not in b[0..n-1], so linear search has to look at every element. Takes time proportional to n.

expected (average) case time. If you look at all possibilities where v could be and average the number of elements linear search has to look at, you would get close to n/2. Still time proportional to n.

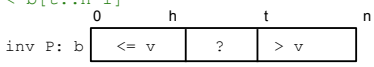
Binary search time (b[0..n-1] is sorted)

```
h= -1; t= n;
// invariant: P (below)
while (h < t-1) {
   int e= (h+t)/2;
   if (b[e] <= v) h= e;
   else  t= e;
}
// b[0..h] <= v < b[t..n-1]
```

b[h+1..t-1] starts out with n elements in it.

Each iteration cuts size of b[h+1..t-1] in half.

worst-case and expected case time: log n



Insertion sort of b[0..n-1]

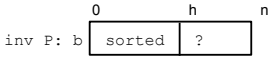
```
h= 0;
// invariant: P (below)
while (h < n) {
   Push b[h] down into
     its sorted position
     in b[0..h];
   h= h+1;
}
```

Worst-case time for Push: h swaps

Average case time for Push: h/2 swaps

1 + 2 + 3 + … + n-1 = n (n-1) / 2

Worst-case and average case time: proportional to n^2



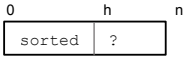
Selection sort of b[0..n-1]

```
h= 0;
// invariant: P (below)
while (h < n) {
   Swap b[h] with min
      value in b[h..n-1];
   h= h+1;
}
```

To find the min value of b[h..n-1] takes time proportional to n - h.

n + (n-1) + … + 3 + 2 + 1 = n (n-1) / 2

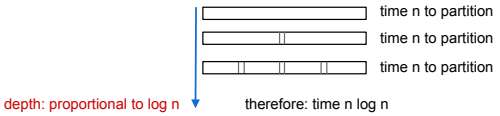
Worst-case and average case time: proportional to n^2



Quicksort of b[0..n-1]

partition(b, h, k) takes time proportional to size of b[h..k]

Best-case time: partition makes both sides equal length



Quicksort of b[0..n-1]

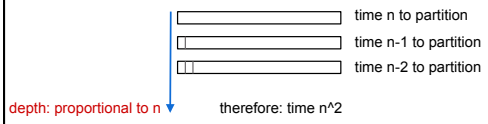
```
/** Sort b[h..k] */
void QS(int[] b, int h, int k) {
  if (b[h..k] size < 2)
     return;
  j= partition(b, h, k);
  // b[h..j-1] <= b[j] <= b[j+1..k]
  QS(h, j-1);
  QS(j+1, k)
}
```

Someone proved that the average or expected time for quicksort is n log n

Quicksort of b[0..n-1]

partition(b, h, k) takes time proportional to size of b[h..k]

Worst-case time: partition makes one side empty



What method calls are legal

```
Animal an; …  an.m(args);
```

legal ONLY if Java can guarantee that method m exists. How to guarantee?

m must be declared in Animal or inherited.

Java Summary

- On the "Resources" tab of the course website
- We have selected some useful snippets
- We recommend going over all the slides

Casting among types

(**int**) 3.2 casts **double** value 3.2 to an **int**

any number type

any number expression

narrow — may be automatic cast → wider

byte **short** **int** **long** **float** **double**

must be explicit cast, may truncate

char is a number type: (**int**) **'V'** (**char**) 86

Unicode representation: 86 **'V'**

Page A-9, inside back cover

48

Declaration of class Circle

Multi-line comment starts with /* ends with */

```
/** An instance (object) represents a circle */
public class Circle {
}
```

Precede every class with a comment

Put declarations of fields, methods in class body: { … }

Put class declaration in file Circle.java

public: Code everywhere can refer to Circle. Called **access modifier**

Page B-5

49

Overloading

Possible to have two or more methods with same name

```
/** instance represents a rectangle */
public class Rectangle {
  private double sideH, sideV; // Horiz, vert side lengths

  /** Constr: instance with horiz, vert side lengths sh, sv */
  public Rectangle(double sh, double sv) {
    sideH= sh; sideV= sv;
  }

  /** Constructor: square with side length s */
  public Rectangle(double s) {
    sideH= s; sideV= s;
  }
  …
}
```

Lists of parameter types must differ in some way

50

Use of this

this evaluates to the name of the object in which is appears

Memorize this!

```
/** Constr: instance with radius radius*/
public Circle(double radius) {
    this.radius= radius;
}
```

Page B-28

51

```
/** An instance represents a shape at a point in the plane */
public class Shape {
  private double x, y; // top-left point of bounding box
  /** Constructor: a Shape at point (x1, y1) */
  public Shape (double x1, double y1) {
    x= x1;  y= y1;
  }
  /** return x-coordinate of bounding box*/
  public double getX() {
    return x;
  }
  /** return y-coordinate of bounding box*/
  public double getY() {
    return y;
  }
}
```

Class Shape

52

Object: superest class of them all

Class doesn't explicitly extend another one? It automatically extends class Object. Among other components, Object contains:

Constructor: **public** Object() {}

```
/** return name of object */
public String toString()
```

c.toString() is "Circle@x1"

```
/** return value of "this object and ob
    are same", i.e. of this == ob */
public boolean equals(Object ob)
```

Page C-18

53

Java has 4 kinds of variable

```
public class Circle {
  private double radius;

  private static int t;

  public Circle(double r) {
    double r1= r;
    radius= r1;
  }
```

Field: declared non-static. Is in every object of class. Default initial val depends on type, e.g. 0 for **int**

Class (static) var: declared **static**. Only one copy of it. Default initial val depends on type, e.g. 0 for **int**

Parameter: declared in () of method header. Created during call before exec. of method body, discarded when call completed. Initial value is value of corresp. arg of call. Scope: body.

Local variable: declared in method body. Created during call before exec. of body, discarded when call completed. No initial value. Scope: from declaration to end of block.

54

Basic class Box

```
public class Box {
  private Object object;
  public void set(Object ob) {
    object = ob;
  }
  public Object get() {
    return object;
  }
}
```

New code

```
Box<Integer> b= new Box<Integer>();
b.set(new Integer(35));
Integer x= b.get();
```

parameter T (you choose name)

Written using generic type

```
public class Box<T> {
  private T object;
  public void set(T ob) {
    object = ob;
  }
  public T get() {
    return object;
  }
} …
```

Replace type Object everywhere by T

55

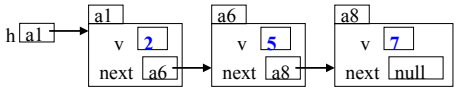
Linked Lists

(These slides are from the class lectures and available on the website as well)

Linked Lists

57

Idea: maintain a list (2, 5, 7) like this:

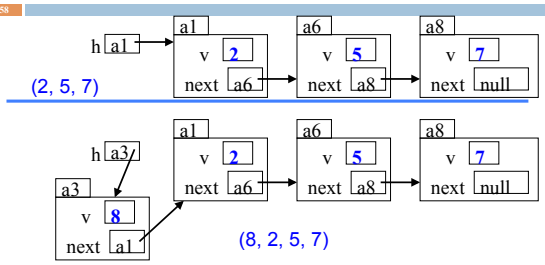


This is a singly linked list

To save space we write names like a6 instead of N@35abcd00

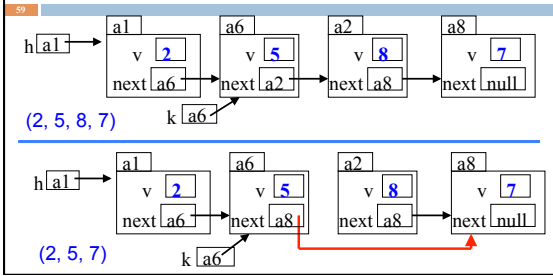
Easy to insert a node in the beginning!

58



Easy to remove a node if you have its predecessor!

59



Recursion

Sum the digits in a non-negative integer

61

```
/** return sum of digits in n.
  * Precondition:  n >= 0 */
public static int sum(int n) {
    if (n < 10) return n;

    // { n has at least two digits }
    // return first digit + sum of rest
    return sum(n/10)  +  n%10 ;
}
```

sum calls itself!

E.g. sum(7) = 7

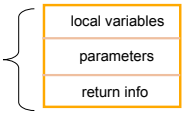
E.g. sum(8703) = sum(870) + 3;

Stack Frame

62

A "frame" contains information about a method call:

At runtime, Java maintains a stack that contains frames for all method calls that are being executed but have not completed.



Method call: push a frame for call on stack, assign argument values to parameters, execute method body. Use the frame for the call to reference local variables, parameters.

End of method call: pop its frame from the stack; if it is a function, leave the return value on top of stack.