Review Session

CS2110 Prelim #1

Primitive types vs classes

- Variable declarations:
  - 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"

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), index0f, charAt
- Useful methods:
  - lastIndexOf, contains, compareTo

1D Array Review

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

Why is the following illegal?
```java
pets[1] = new Object();
```

Java arrays do not change size!

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

Java arrays do not change size!

```java
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.
```java
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?
```java
b.length
```

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

How many columns in row 1?
```java
b[1].length
```
2D arrays: An array of 1D arrays.

```java
int[][] b = new int[2][];
The elements of b are of type int[].
```

A Throwable instance: ArithmeticException

```java
Exception
Throwable
ArithmeticException
```

Try-catch blocks

```java
class Ex {
  void first() {
    second();
  }
  void second() {
    third();
  }
  void third() {
    int c = 5/0;
  }
}
```

Exception: ArithmeticException

```java
try
  System.out.println("in");
  third();
  System.out.println("out");
catch (Exception e) {
  System.out.println("error");
}
```

Console:

```
Exception Type
ArithmeticException!
```
### How to write an exception class

```java
/** 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!

#### Abstract Classes

<table>
<thead>
<tr>
<th>Shape</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangle</td>
<td>base</td>
<td>height</td>
</tr>
<tr>
<td>Circle</td>
<td>radius</td>
<td></td>
</tr>
</tbody>
</table>

### A Partial Solution:

**Add method area to class Shape:**

```java
public abstract 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

```java
public abstract class Shape {
    public double area() { return 0; }
}
```

### Solution: Abstract methods

```java
public abstract class Shape {
    public abstract double area();
}
```

- 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...)

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

class Human extends Mammal implements Whistler {
    // Must implement all methods in the implemented interfaces
}
```

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

class Human extends Mammal implements Whistler, Singer {
    // Must implement singTo(Human h) and whistle()
}
```

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

class Human extends Mammal implements Whistler, Singer {
    // Must implement singTo(Human h) and whistle()
}
```

```
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 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));
    }
}

Interfaces

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

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

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

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

\[
\begin{array}{c|c}
 b & ??? \\
\end{array}
\]

Invariant

\[
\begin{array}{c|c|c}
 b & ??? & s = \text{sum} \\
 h & & \\
\end{array}
\]

Postcondition

\[
\begin{array}{c|c|c}
 b & s = \text{sum} \\
\end{array}
\]

Add elements backwards

\[
\begin{array}{c|c|c}
 0 & h & ??? \\
& & s = \text{sum} \\
\end{array}
\]

inv: \quad \begin{align*}
\text{INV: } & b \\
\text{Precondition: } & \\
\text{Invariant: } & b \quad ??? \quad s = \text{sum} \\
\text{Postcondition: } & \\
\text{Loop Invariants: } & 0 \quad h \\
\text{Postcondition: } & b \quad s = \text{sum} \\
\end{align*}

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

Does it start right?  Does it stop right?  Does it keep the invariant true?  Does it make progress toward termination?

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

\[
\begin{array}{cccc}
\text{any number type} & \text{any number expression} & \text{narrow} & \text{wider} \\
\text{may be automatic cast} & \text{may be explicit cast, may truncate} & \\
\text{byte short int long float double} & \\
\end{array}
\]

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

Unicode representation: \ 86 'V'

Declaration of class Circle

public class Circle {
    // multi-line comment starts with /* ends with */
    /** An instance (object) represents a circle */
    public Circle () {
        // put declarations of fields, methods in class body: ...
    }
    // public: Code everywhere can refer to Circle. Called access modifier
    // Precede every class with a comment
    // Put class declaration in file Circle.java
}

Page A-9, inside back cover
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;
    }
    ...
}

Use of this
this evaluates to the name of the object in which it appears
Memorize this!
/** Constr: instance with radius radius*/
public Circle(double radius) {
    this.radius= radius;
}

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) {

Java has 4 kinds of variable
public class Circle {
    private double radius;
    private static int t; // Class (static) var: declared static. Only one copy of it. Default initial val depends on type, e.g. 0 for int
    public Circle(double r) { double r1= r; radius= r1; }
}

Basic class Box
Written using generic type
public class Box<T> {
    private T 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(15));
    Integer x = b.get();
    ...
}

Parameter T (you choose name)
parameter T (you choose name)
Linked Lists

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

```
  h
  ^
 |  v
a1 -> a5 -> a8 -> null
```

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!

```
  h
  ^
 |  v
a1 -> a5 -> a8
  ^
 |  v
a6
```

8

(8, 2, 5, 7)

Easy to remove a node if you have its predecessor!

```
  h
  ^
 |  v
a1 -> a5 -> a8
  ^
 |  v
a6
```

(2, 5, 7)

Recursion

Sum the digits in a non-negative integer

```java
/** return sum of digits in n. 
* Precondition:  n >= 0 */
public static int sum(int n) {
    sum calls itself!
    if (n < 10) return n;
    // { n has at least two digits }
    // return first digit + sum of rest
    return sum(n/10) + n%10 ;
}
```

E.g. sum(7) = 7
E.g. sum(8703) = sum(870) + 3;
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 on the 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 the stack.

(some) things to know for the prelim

- Can you list the steps in evaluating a new-expression? Can you do them yourself on a piece of paper?
- Can you list the steps in executing a method call? Can you do them yourself on a piece of paper?
- Do you understand exception handling? E.g. What happens after a catch block has been executed?
- Can you write a recursive method or understand a given one?
- Abstract class and interfaces
- ArrayList, interface Comparable
- Loops invariants