CSE 8B Today

MORE RECURSION!

• Exam 4 next Tuesday 27th
Review on DISCUSSION tomorrow!
(and review topics sheet will be ONLINE)

• PSA EXTRA (optional): *AI player*
- Hints today AFTER the break to give exam3 back.
- Surprise! It’s about RECURSION 😊
- It’ll replace your lowest **PSA2 to PSA7** score. BUT YOU SHOULD DO IT FOR THE SAKE OF LEARNING! and practice with recursion
- It’s very challenging BUT fun!

FINAL EXAM:
- Mandeville Auditorium
- Post request NOW if you need to change to the other section time

X to move.
Is there a way to win?
1. True or False? “If we run the main method from the class RecDemo, we will create one stack frame for each method of the class RecDemo.”

A. True, we create one stack frame for the methods in that class only.

B. True, we create one stack frame for the methods in that class and one stack frame for each method in any other class that we use.

C. False, we create one stack frame for each method that is called during that execution.

D. False, we are not allowed to create more than one stack frame.
2. How many base cases can we have in a recursive method?

A. Are you crazy? Only one!

B. Recursive methods do not have base cases.

C. We can have one base case for each stack frame that is created.

D. A recursive method needs to have at least one base case, but can have more.
3. What is the base case in the following recursive method

```java
public static void xMethod(int n) {
    if (n != 0) {
        System.out.println(n);
        xMethod(n / 10);
    }
}
```

```java
public static void main(String[] args) {
    xMethod(1234567);
}
```

A. When n != 0

B. When n == 0

C. When n == 1234567

D. When n == 1
4. How many stack frames are created when we run
RecursionDemo.recFun3( 3 )

// In the class RecursionDemo
public static void recFun3( int num ) {
    if ( num > 0 ) {
        num = num - 2;
        recFun3( num );
        System.out.println( num );
    }
}

A. 1
B. 3
C. 5
D. 7
E. Infinite
4. How many stack frames are created when we run 
RecursionDemo.recFun3( 3 )

// In the class RecursionDemo 
public static void recFun3( int num ) {
  if ( num > 0 ) {
    num = num - 2;
    recFun3( num );
    System.out.println(num);
  }
}

A. 1
B. 3
C. 5
D. 7
E. Infinite

// condition is false!
// BASE CASE!
// Start “coming back”
Problem from Thursday’s class: Additional questions...

// In the class RecursionDemo
public static void recFun3( int num ) {
    if ( num > -1 ) {
        num = num - 2;
        recFun3( num );
        System.out.println( num );
    }
}

What does the above code print?

A. Nothing, it never gets to the print statement  
B. 3 1  
C. 3 1  
D. 1  
E. 3 3

What is the base case here?

num <= 1

Can you rewrite the code in recFun2 to make the answer D (with the same method call)?

← One possible solution

RecursionDemo.recFun3( 3 )
Recursion, more *meaningful* practice: FIND!

Write a method to find an element in an array *between startIndex and the end* of the array and return its index as shown in the examples.

```java
public static int find( String[] myList, String toFind, int startIndex )
{  ???  }
```

```plaintext
>> int[] a = {"Hello", "Welcome", "Turtle", "fun"};
>> RecursionDemo.find( a, "Hello", 0 )
0
>> RecursionDemo.find( a, "Happy", 0 )
-1
>> RecursionDemo.find( a, "fun", 2 )
3
>> RecursionDemo.find( a, "Hello", 2 )
-1
```

What is the base case?
A. The element at startIndex is equal to toFind
B. The array is empty
C. startIndex is (greater than or) equal to myList.length
D. toFind is not in myList
E. More than one of these  

Actually all are correct except D!
Recursion: FIND!

FIND an element in an array between startIndex and the end of the array and return its index (or -1 if it’s NOT THERE)

```java
public static int find( String[] myList, String toFind, int startIndex )
{
    // Base case 1: I GOT TO THE END!
    if ( startIndex >= myList.length ) {
        return -1;
    }
    // Base case 2: FOUND IT!
    if ( toFind.equals( myList[startIndex] ) ) {
        return startIndex;
    }
    // Recursive step here . . .
}
```

Could we switch the order of the two base cases?

A. Yes  
B. No

CAREFUL!!! Many times the base case actually checks if the “parameters” are valid (in this case, if startIndex is too large... Out of bounds! In (if we would get to run it ... )
Complete the method

```java
public static int find( String[] myList, String toFind, int startIndex )
{
    if ( startIndex >= myList.length ) {
        return -1;
    }
    if ( toFind.equals( myList[startIndex] ) ) {
        return startIndex;
    }

    // Recursive step here . . .

    WHAT ELSE DO WE NEED once we have designed the “base case”?-
```
Good job!

Make sure results explained today make sense! Even if you scored a 10 ... review the “reasons” for each answer.

If you scored below 5, you need to start studying A LOT! (concepts do not sound clear)
- Re-read chapters 10 and 11
- Then come to office hours!
public abstract class Cake {
    protected boolean frosting; // field
    public abstract String recipe(); // abstract method
    public String toString()
    {
        return "Something sweet!";
    }
} // END CLASS CAKE

public class ChocolateCake extends Cake {
    public ChocolateCake(boolean addFrosting)
    {
        this.frosting = addFrosting;
    }
    public String recipe()
    {
        return "Prepare the dough and bake.";
    }
    public String toString()
    {
        if (this.frosting){ return super.toString()+ " with frosting and chocolate"; }
        else{ return super.toString()+ " with chocolate"; }
    }
} // END CLASS CHOCOLATECAKE

public class BirthDayChocoCake extends ChocolateCake {
    protected int nCandles; // additional field
    public BirthDayChocoCake (boolean addF, int numCandles)
    {
        super(addF); this.nCandles = numCandles;
    }
    public void putCandles()
    {
        System.out.println(“The cake has “+ nCandles +“ candles”);
    }
} // END
public abstract class Cake {
    protected boolean frosting; // field
    public abstract String recipe(); // abstract method
    public String toString()
    { return "Something sweet!"; } // END CLASS CAKE
}

public class ChocolateCake extends Cake {
    public ChocolateCake (boolean addFrosting)
    { this.frosting = addFrosting; }
    public String recipe()
    { return "Prepare the dough and bake."; }
    public String toString()
    { if (this.frosting) return super.toString()+ " with frosting and chocolate";
      else return super.toString()+ " with chocolate"; }
} // END CLASS CHOCOLATECAKE

public class BirthDayChocoCake extends ChocolateCake {
    protected int nCandles; // additional field
    public BirthDayChocoCake (boolean addF, int numCandles)
    { super(addF); this.nCandles = numCandles; }
    public void putCandles()
    { System.out.println("The cake has "+ nCandles +" candles"); } // END
}

<table>
<thead>
<tr>
<th></th>
<th>Compilation Error</th>
<th>Runtime Error</th>
<th>No error</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChocolateCake c3 = new ChocolateCake( true ); ((BirthDayChocoCake) c3).putCandles();</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cake c4 = new BirthDayChocoCake( false, 5 ); String message = c4.recipe();</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BirthDayChocoCake c = new ChocolateCake(true);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cake c2 = new Cake( false );</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
public abstract class Cake {
    protected boolean frosting; // field
    public abstract String recipe(); // abstract method
    public String toString()
    { return "Something sweet!"; } } // END CLASS CAKE

public class ChocolateCake extends Cake {
    public ChocolateCake (boolean addFrosting) {
        this.frosting = addFrosting;
    }
    public String recipe() {
        return "Prepare the dough and bake.";
    }
    public String toString() {
        if (this.frosting) {
            return super.toString() + " with frosting and chocolate";
        } else {
            return super.toString() + " with chocolate";
        }
    } // END CLASS CHOCOLATECAKE
}

public class BirthDayChocoCake extends ChocolateCake {
    protected int nCandles; // additional field
    public BirthDayChocoCake ( boolean addF, int numCandles) {
        super(addF); this.nCandles = numCandles;
    }
    public void putCandles() {
        System.out.println("The cake has " + nCandles + " candles");
    } // END
}

<table>
<thead>
<tr>
<th>Compilation Error (A)</th>
<th>Runtime Error (B)</th>
<th>No error (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChocolateCake c3 = new ChocolateCake( true );</td>
<td>((BirthDayChocoCake) c3).putCandles();</td>
<td></td>
</tr>
<tr>
<td>Cake c4 = new BirthDayChocoCake( false, 5 );</td>
<td>String message = c4.recipe();</td>
<td></td>
</tr>
<tr>
<td>BirthDayChocoCake c = new ChocolateCake(true);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cake c2 = new Cake( false );</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
public abstract class Cake {
    protected boolean frosting; // field
    public abstract String recipe(); // abstract method
    public String toString()
    { return "Something sweet!"; } // END CLASS CAKE
}

public class ChocolateCake extends Cake {
    public ChocolateCake (boolean addFrosting)
    { this.frosting = addFrosting; }
    public String recipe()
    { return "Prepare the dough and bake."; }
    public String toString()
    {
        if (this.frosting){ return super.toString() + " with frosting and chocolate";}
        else{ return super.toString() + " with chocolate"; }
    } // END CLASS CHOCOLATECAKE
}

public class BirthDayChocoCake extends ChocolateCake {
    protected int nCandles; // additional field
    public BirthDayChocoCake ( boolean addF, int numCandles)
    { super(addF); this.nCandles = numCandles; }
    public void putCandles()
    { System.out.println("The cake has "+ nCandles +" candles"); } // END
}

<table>
<thead>
<tr>
<th></th>
<th>Compilation Error (A)</th>
<th>Runtime Error (B)</th>
<th>No error (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChocolateCake c3 = new ChocolateCake(true); ((BirthDayChocoCake) c3).putCandles();</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cake c4 = new BirthDayChocoCake(false, 5); String message = c4.recipe();</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BirthDayChocoCake c = new ChocolateCake(true);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cake c2 = new Cake(false);</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We lied to the compiler!
public abstract class Cake {
    protected boolean frosting; // field
    public abstract String recipe(); // abstract method
    public String toString()
    {
        return "Something sweet!";
    } // END CLASS CAKE
}

public class ChocolateCake extends Cake {
    public ChocolateCake (boolean addFrosting)
    {
        this.frosting = addFrosting;
    }
    public String recipe()
    {
        return "Prepare the dough and bake.";
    }
    public String toString()
    {
        if (this.frosting){ return super.toString()+ " with frosting and chocolate";}
        else{ return super.toString()+ " with chocolate"; }
    } // END CLASS CHOCOLATECAKE
}

public class BirthDayChocoCake extends ChocolateCake {
    protected int nCandles; // additional field
    public BirthDayChocoCake ( boolean addF, int numCandles)
    {
        super(addF); this.nCandles = numCandles;
    }
    public void putCandles()
    { System.out.println("The cake has " + nCandles + " candles"); }
} // END

<table>
<thead>
<tr>
<th></th>
<th>Compilation Error</th>
<th>Runtime Error</th>
<th>No error</th>
</tr>
</thead>
</table>
| ChocolateCake c3 = new ChocolateCake( true );
  ((BirthDayChocoCake) c3).putCandles(); |                  | X             |          |
| Cake c4 = new BirthDayChocoCake( false, 5 );
  String message = c4.recipe(); |                  | X             |          |
| BirthdayChocoCake c = new ChocolateCake(true); |                  |               |          |
| Cake c2 = new Cake( false ); |                  |               |          |
public abstract class Cake {
    protected boolean frosting; // field
    public abstract String recipe(); // abstract method
    public String toString()
    { return "Something sweet!"; } } // END CLASS CAKE

public class ChocolateCake extends Cake {
    public ChocolateCake (boolean addFrosting)
    { this.frosting = addFrosting; }
    public String recipe()
    { return "Prepare the dough and bake."; }
    public String toString()
    { if (this.frosting) { return super.toString() + " with frosting and chocolate"; }
      else { return super.toString() + " with chocolate"; }
    } } // END CLASS CHOCOLATECAKE

public class BirthDayChocoCake extends ChocolateCake {
    protected int nCandles; // additional field
    public BirthDayChocoCake ( boolean addF, int numCandles)
    { super(addF); this.nCandles = numCandles; }
    public void putCandles()
    { System.out.println("The cake has " + nCandles + " candles"); } } // END

<table>
<thead>
<tr>
<th></th>
<th>Compilation Error (A)</th>
<th>Runtime Error (B)</th>
<th>No error (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChocolateCake c3 = new ChocolateCake( true );</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>((BirthDayChocoCake) c3).putCandles();</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cake c4 = new BirthDayChocoCake( false, 5 );</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>String message = c4.recipe();</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BirthDayChocoCake c = new ChocolateCake(true);</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Cake c2 = new Cake( false );</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
public abstract class Cake {  
    protected boolean frosting; // field
    public abstract String recipe(); // abstract method
    public String toString() {  
        return “Something sweet!”;  
    } // END CLASS CAKE

public class ChocolateCake extends Cake {  
    public ChocolateCake (boolean addFrosting)
    {  
        this.frosting = addFrosting;  
    }
    public String recipe()
    {  
        return “Prepare the dough and bake.”;  
    }
    public String toString() {  
        if (this.frosting){ return super.toString()+ “ with frosting and chocolate”;}  
        else{ return super.toString()+ “ with chocolate”; }  
    } // END CLASS CHOCOLATECAKE

public class BirthDayChocoCake extends ChocolateCake {  
    protected int nCandles; // additional field
    public BirthDayChocoCake ( boolean addF, int numCandles)
    {  
        super(addF); this.nCandles = numCandles;  
    }
    public void putCandles()  
    {  
        System.out.println(“The cake has “+ nCandles +” candles”);  
    } } // END

<table>
<thead>
<tr>
<th>Compilation Error (A)</th>
<th>Runtime Error (B)</th>
<th>No error (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChocolateCake c3 = new ChocolateCake( true ); ((BirthDayChocoCake) c3).putCandles();</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Cake c4 = new BirthDayChocoCake( false, 5 ); String message = c4.recipe();</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>BirthDayChocoCake c = new ChocolateCake(true);</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Cake c2 = new Cake( false );</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

By default, we only get “empty” constructor Cake().

By default, we only get “empty” constructor Cake().

Cake (boolean…) Does NOT exist! + we can NOT instantiate abstract class

+ we can NOT instantiate abstract class
public abstract class Cake {
    protected boolean frosting; // field
    public abstract String recipe(); // abstract method
    public String toString()
    { return "Something sweet!"; } // END CLASS CAKE
}

public class ChocolateCake extends Cake {
    public ChocolateCake (boolean addFrosting)
    { this.frosting = addFrosting; }
    public String recipe()
    { return "Prepare the dough and bake."; }
    public String toString()
    {
        if (this.frosting){ return super.toString()+ " with frosting and chocolate";} else{ return super.toString()+ " with chocolate"; }
    } // END CLASS CHOCOLATECAKE
}

public class BirthDayChocoCake extends ChocolateCake {
    protected int nCandles; // additional field
    public BirthDayChocoCake ( boolean addF, int numCandles)
    { super(addF); this.nCandles = numCandles; }
    public void putCandles()
    { System.out.println("The cake has "+ nCandles +" candles"); } // END
}

class MyTester {
    public static void main( String[] args ) {
        Cake c1 = new ChocolateCake( true );
        Cake c2 = new BirthDayChocoCake( false, 3 );
        ( (BirthDayChocoCake) c2).putCandles();
        System.out.println( c2 );
        System.out.println( c1 );
    }
}

From which type of object can I call putCandles()?

A. Cake, ChocolateCake or BirthDayChocoCake, they are all fine
B. ChocolateCake or BirthDayChocoCake
C. Only BirthDayChocoCake
public abstract class Cake {
    protected boolean frosting; // field
    public abstract String recipe(); // abstract method
    public String toString()
    {     return “Something sweet!”; } } // END CLASS CAKE

public class ChocolateCake extends Cake {
    public ChocolateCake (boolean addFrosting)
    { this.frosting = addFrosting; }
    public String recipe()
    {     return “Prepare the dough and bake.”; }
    public String toString() {
        if (this.frosting) { return super.toString()+ “ with frosting and chocolate”;}  
        else { return super.toString()+ “ with chocolate”;}  
    } } // END CLASS CHOCOLATECAKE

public class BirthDayChocoCake extends ChocolateCake {
    protected int nCandles; // additional field
    public BirthDayChocoCake ( boolean addF, int numCandles)
    { super(addF); this.nCandles = numCandles; }
    public void putCandles()
    { System.out.println(“The cake has “+ nCandles +“ candles”);} } // END

class MyTester {
    public static void main( String[] args ) {
        Cake c1 = new ChocolateCake( true );
        Cake c2 = new BirthDayChocoCake( false, 3 );
        (BirthDayChocoCake) c2).putCandles();
        System.out.println( c2 );
        System.out.println( c1 );
    }
}

Which toString method it’s actually called when I print c2?
(REMEMBER print command AUTOMATICALLY calls toString)

A. None, BirthDayChocoCake does NOT have that method
B. toString method from ChocolateCake
C. toString method from Cake
public abstract class Cake {
    protected boolean frosting; // field
    public abstract String recipe(); // abstract method
    public String toString()
    {        return “Something sweet!”;    } // END CLASS CAKE
}

public class ChocolateCake extends Cake {
    public ChocolateCake (boolean addFrosting)
    {        this.frosting = addFrosting;    }
    public String recipe()
    {        return “Prepare the dough and bake.”;    }
    public String toString() {
        if (this.frosting){ return super.toString()+ “ with frosting and chocolate”;}
        else{ return super.toString()+ “ with chocolate”;    }
    } // END CLASS CHOCOLATECAKE

public class BirthDayChocoCake extends ChocolateCake {
    protected int nCandles; // additional field
    public BirthDayChocoCake ( boolean addF, int numCandles)
    {        super(addF); this.nCandles = numCandles;    }
    public void putCandles()
    {        System.out.println(“The cake has ”+ nCandles +” candles”);    } // END
}

class MyTester {
    public static void main( String[] args ) {
        Cake c1 = new ChocolateCake( true );
        Cake c2 = new BirthDayChocoCake( false, 3 );
        ( (BirthDayChocoCake) c2).putCandles();
        System.out.println( c2 );
        System.out.println( c1 );
    } }
import javax.swing.*;
import java.awt.*;

public class MyPanel extends JPanel {

  protected void paintComponent( Graphics g ) {
    super.paintComponent(g);
    g.drawLine(10, 10, 100, 100);
    g.drawLine(100, 100, 190, 10);
  }

  public static void main( String[] args ) {
    JFrame f = new JFrame();
    f.setSize( 200, 200 );
    JPanel p = new MyPanel();
    f.add( p );
    f.setVisible( true );
  }
}

What does super.* mean? (* being any method or variable name that exists)
A. Access something in the calling object
B. Access something in the parent class
C. Override something in the parent class

This code will display the second window (the empty one), because only the JFrame is visible.

At runtime, Java will detect that the object referenced by p is actually a MyPanel object and will use the paintComponent method in the MyPanel class.

If we remove the line f.add( p ); the JPanel p won’t be added to of our JFrame f and therefore we won’t see the V drawing in the window displayed after running the main method.

The line super.paintComponent(g) included in MyPanel’s paintComponent method tells Java that this method overrides the paintComponent method from JPanel. i.e., without that line, in this code Java would use the paintComponent method from the JPanel class instead of the one from the MyPanel class.
import javax.swing.*;
import java.awt.*;
public class MyPanel extends JPanel{
    protected void paintComponent( Graphics g ) {
        super.paintComponent(g);
        g.drawLine(10, 10, 100, 100);
        g.drawLine(100, 100, 190, 10);
    }
}

public static void main( String[] args ) {
    JFrame f = new JFrame();
    f.setSize( 200, 200 );
    JPanel p = new MyPanel();
    f.add( p );
    f.setVisible( true );
}

True  False  This code will display the second window (the empty one), because only the JFrame is visible.
True  False  At runtime, Java will detect that the object referenced by p is actually a MyPanel object and will use the paintComponent method in the MyPanel class.
True  False  If we remove the line f.add( p ); the JPanel p won’t be added to of our JFrame f and therefore we won’t see the V drawing in the window displayed after running the main method.
True  False  The line super.paintComponent(g) included in MyPanel’s paintComponent method tells Java that this method overrides the paintComponent method from JPanel. i.e., without that line, in this code Java would use the paintComponent method from the JPanel class instead of the one from the MyPanel class.
False!!! that line does not mean “override”. line super.paintComponent just tells to call the parent (JPanel) paintComponent method.
In the sample code from previous question, if we want to assign something to the variable \( p \), we can assign a reference to an object of the same class than \( p \), or to an object of any \textit{SUBCLASS} of JPanel and it will be correct; this is known as \textbf{POLYMORPHISM} (or any close “spelling” ;-

\textbf{INHERITANCE} enables you to define a general class (e.g., a superclass) and later extend it to more specialized classes (e.g., subclasses). – \textit{directly from the book}...

In the code from previous question, MyPanel is a \textit{SUBCLASS} of JPanel, and JPanel is the \textbf{SUPERCLASS} of MyPanel. This implies that the class \textbf{MYPANEL} inherits fields and methods from the class JPanel.

Every class in Java has a common superclass, the \textbf{OBJECT} class. (\textit{Java.lang.Object is also ok!})

\* \textit{OR ANY SYNONYM from the BOOK}
Intelligent CS 8B?

An object is structured data that is alive, responsible, and intelligent.

This week’s objects and classes will be clever than ever . . .

This part and assignment are OPTIONAL!

- DUE ON TUESDAY (no additional slip day, everyone due on TUESDAY)
- INDIVIDUAL
- If you get better score here: REPLACE your lowest PSA score so far (PSA2 – PSA7)

An object is structured data that is alive, responsible, and intelligent.

To move.
Is there a way to win?

Slides from Zach Dodds at Harvey Mudd College
Deep Blue was a chess-playing computer developed by IBM. On May 11, 1997, the machine, with human intervention between games, won the second six-game match against world champion Garry Kasparov by two wins to one with three draws. Kasparov accused IBM of cheating and demanded a rematch, but IBM refused and dismantled Deep Blue. Kasparov had beaten a previous version of Deep Blue in 1996.
Strategic thinking!

Java *had* no Connect-four datatype…

I feel ahead of the game here...

May the best alien win

How many moves ahead might we have to look?

… but you've already corrected that!

DEMO!

Now, *may the best machine win!*
The **ConnectFourPlayer** class

What data does a computer AI player need?

- **char checker**
- **String tieBreakType**
- **int ply**

'C' or 'X' checks, O or X

How about knowledge about its opponent?
The **C4Player** class

**Board**

- `Board(int width, int height)`
- `boolean allowsMove(int col)`
- `void addMove(int col, char checker)`
- `void delMove(int col)`
- `String toString()`
- `boolean isFull()`
- `boolean winsFor(char checker)`
- `void hostGame()`

**C4Player**

- `C4Player(char ch, String tbt, int plyln)`
- `String toString()`
- `char oppCh()`
- `double scoreBoard(C4Board b)`
- `double[] scoresFor(C4Board b)`
- `int tiebreakMove(double[] scores)`
- `int nextMove(C4Board b)`

---

**Board**

- `Board(int width, int height)`
- `boolean allowsMove(int col)`
- `void addMove(int col, char checker)`
- `void delMove(int col)`
- `String toString()`
- `boolean isFull()`
- `boolean winsFor(char checker)`
- `void hostGame()`
The **C4Player** class

### C4Player

- **C4Player(char ch, String tbt, int plyIn)**
- **String toString()**
- **double scoreBoard(C4Board b)**
- **double[] scoresFor(C4Board b)**
- **int tiebreakMove(double[] scores)**
- **int nextMove(C4Board b)**

### Board

- **Board(int width, int height)**
- **boolean allowsMove(int col)**
- **void addMove(int col, char checker)**
- **void delMove(int col)**
- **String toString()**
- **boolean isFull()**
- **boolean winsFor(char checker)**
- **void hostGame()**

**HOW DO I DECIDE THAT!!??!!?!**
The **C4Player** class

**Board**

Board(int width, int height)

boolean allowsMove(int col)

void addMove(int col, char checker)

void delMove(int col)

String toString()

**C4Player**

C4Player(char ch, String tbt, int plyIn)

String toString()

char oppCh() // opposite checker than current turn

double scoreBoard(C4Board b)

double[] scoresFor(C4Board b)

int tiebreakMove(double[] scores)

int nextMove(C4Board b) // gives the column index where the computer wants to play given a board b ...

HOW DO I DECIDE THAT!!?!?!
Why AI is challenging:

Make no mistake about it: computers process numbers - not symbols

(We need to get a “number” to represent each of our “possible choices” in this PSA)

Computers can only help us to the extent that we can arithmetize an activity.

- paraphrasing Alan Perlis
We need to EXPLORE the options . . . It means, “analyze” current status of the board

scoreBoard vs. scoresFor

LET’s START WITH THE EASY CASE . . .

double scoreBoard(C4Board b) looks ahead 0 moves

it scores the board as it stands now

0-ply only

-----------------------------------------------

double[] scoresFor(C4Board b) returns a LIST of scores: one for each column a player might move next…

0-ply

it gives a separate score for each column and considers future moves up to n-ply

1-ply

2-ply

...
double scoreBoard(C4Board b)

Assigns a score to any board, b

A simple system:

<table>
<thead>
<tr>
<th>Score for 'x'</th>
<th>Score for 'o'</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>50.0</td>
</tr>
</tbody>
</table>

for a win      for anything else

0.0 for a loss

What are the scores for X and O at the board on the left?
A. 100.0, 100.0  B. 100.0, 50.0  C. 100.0, 0.0  D. 50.0, 50.0  E. None of these
double scoreBoard(C4Board b)

Assigns a score to any board, b

A simple system:

<table>
<thead>
<tr>
<th>Score for X</th>
<th>Score for O</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>50.0</td>
<td>50.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>for a win</td>
<td>for anything else</td>
</tr>
</tbody>
</table>

What are the scores for X and O at the board on the right?

A. 100.0, 100.0  B. 100.0, 50.0  C. 100.0, 0.0  D. 50.0, 50.0  E. None of these
Assigns a *score* to any board, \( b \)

A simple system:

- **100.0** for a win
- **50.0** for anything else
- **0.0** for a loss

Implementing…

**double scoreBoard(C4Board b)**

How can there be no 'X' or 'O' input?

What methods that already exist will come in handy?

This isn't looking very far ahead!
We need to EXPLORE the options . . . It means, “analyze” current status of the board.

**scoreBoard** vs. **scoresFor**

```java
double scoreBoard(C4Board b) looks ahead 0 moves
```

- It scores the board as it stands *now*.

```java
double[] scoresFor(C4Board b) returns a LIST of scores: *one for each column a player might move next* . . .
```

- It gives a separate **score for** each column and considers **future moves** up to *n- ply*.

"**Ply**" is the number of moves to look ahead...
Scoring *moves* at 0 ply

"Ply" is the number of moves to look ahead...

What would the 0-ply scores be for each *column* (move)?

0 ply is a Zen-like approach: *exist only in the present*

0-ply scores for ⫿

0-ply means 0 moves are made!

We use $-1$ as the score into a full column.
Scoring *moves* at 0 ply

"Ply" is the number of moves to look ahead...

0 ply is a Zen-like approach: *exist only in the present*

What would the 0-ply scores be for each *column* (move)?

0-ply scores for ♜

-1  50  50  50  50  50  50

0-ply means 0 moves are made!

We use -1 as the score into a full column.

`o0.scoreBoard(b42)`

`o0.scoresFor(b42)`
Scoring *moves* at 1 ply

What would the 1-ply scores be for each *column* (move)?

A 1-ply lookahead player will "see" an impending victory.

"Gotcha!"

1-ply scores for ○

1-ply means 1 move is made!

I try 1 ply!
Scoring *moves* at 1 ply

What would the 1-ply scores be for each *column* (move)?

A 1-ply look-ahead player will be able to "see" an impending victory.

"Gotcha!"

1-ply scores for

<table>
<thead>
<tr>
<th></th>
<th>50</th>
<th>50</th>
<th>50</th>
<th>100</th>
<th>50</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Seeing the recursive structure

If I do that... what happens to my opponent scores...

1-ply scores for 🟠

-1

1-ply means 1 move is made!

0-ply scores for ⚫

-1 50 50 50 50 50 50

0-ply means 0 moves are made!

to move
to move
Seeing the recursive structure

If I do that... what happens to my opponent scores...

1-ply scores for ○

| -1 | 50 | 50 | 50 | 100 |

1-ply means 1 move is made!

0-ply scores for ●

0-ply means 0 moves are made!

to move

x x x x x x x x

to move
Seeing the recursive structure

If I do that... what happens to my opponent scores...

1-ply scores for ●

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>100</th>
</tr>
</thead>
</table>

-1 50 50 50

1-ply means 1 move is made!

0-ply scores for ●

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

-1 0 0 0 0 0 0 0

0-ply means 0 moves are made!
Scoring *moves* at 2 ply

What would the 2-ply scores be for each *column* (move)?

*(If I do this... what the other player do? What score will I get then?)*

A 2-ply lookahead player will be able to see a way to win or block the opponent's win

"Gotcha!" + "Uh Oh..."

2-ply scores for ♦️

A. -1, 0, 0, 0, 100, 0, 0
B. 0, 50, 50, 50, 50, 50, 50
C. 50, 50, 50, 50, 100, 50, 50
D. -1, 0, 0, 0, 50, 0, 0
E. None of these

What about 3-ply?
Scoring *moves* at 2 ply

What would the 2-ply scores be for each *column* (move)?

*(If I do this... what the other player do? What score will I get then?)*

A 2-ply lookahead player will be able to see a way to win or block the opponent's win

"Gotcha!" + "Uh Oh..."

2-ply scores for ●

-1 50 50 50 100 50 50

2-ply means 2 moves are made!

What about 3-ply?

A. -1, 0, 0, 0, 100, 0, 0
B. 0, 50, 50, 50, 50, 50, 50
C. 50, 50, 50, 50, 100, 50, 50
D. -1, 0, 0, 0, 50, 0, 0
E. None of these
scoreBoard vs. scoresFor

double scoreBoard(C4Board b) looks ahead 0 moves

0-ply only

it **scores** the **board** as it stands **now**


double[] scoresFor(C4Board b) returns a LIST of scores: **one for each column a player might move next**…

0-ply

it gives a separate **score for** each column and considers **future** moves up to **n-ply**

1-ply

2-ply

...
Fill in the score for each column at each ply up to 3.

The same move is evaluated at each ply... it's just evaluated farther into the future!

Each row is different in at least 1 score…

<table>
<thead>
<tr>
<th>col 0</th>
<th>col 1</th>
<th>col 2</th>
<th>col 3</th>
<th>col 4</th>
<th>col 5</th>
<th>col 6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0-ply scores for 'O':
Looks 0 moves into the future

1-ply scores for 'O':
Looks 1 move into the future

2-ply scores for 'O':
Looks 2 moves into the future

3-ply scores for 'O':
Looks 3 moves into the future
Fill in the score for each column at each ply up to 3.

*The same move is evaluated at each ply...* it's just evaluated farther into the future!

Each row is different in at least 1 score...

### 0-ply scores for 'O':

<table>
<thead>
<tr>
<th>col 0</th>
<th>col 1</th>
<th>col 2</th>
<th>col 3</th>
<th>col 4</th>
<th>col 5</th>
<th>col 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

Looks 0 moves into the future

### 1-ply scores for 'O':

<table>
<thead>
<tr>
<th>col 0</th>
<th>col 1</th>
<th>col 2</th>
<th>col 3</th>
<th>col 4</th>
<th>col 5</th>
<th>col 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>50</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

Looks 1 move into the future

### 2-ply scores for 'O':

<table>
<thead>
<tr>
<th>col 0</th>
<th>col 1</th>
<th>col 2</th>
<th>col 3</th>
<th>col 4</th>
<th>col 5</th>
<th>col 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>50</td>
</tr>
</tbody>
</table>

Looks 2 moves into the future

### 3-ply scores for 'O':

<table>
<thead>
<tr>
<th>col 0</th>
<th>col 1</th>
<th>col 2</th>
<th>col 3</th>
<th>col 4</th>
<th>col 5</th>
<th>col 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

Looks 3 moves into the future
(0) Suppose you're playing at 2 ply...

(1) Make ALL moves!

scoresFor's idea

Make ALL moves!
(0) Suppose you're playing at 2 ply...

(1) Make ALL moves!
(0) Suppose you're playing at 2 ply...
(1) You need to make ALL moves!
AND FOR EACH MOVE:

(2) Ask OPPONENT its scoresFor at ply-1

these are all of the opponent's calls to scoresFor!
(0) Suppose you're playing at 2 ply...

(1) You need to make ALL moves!

AND FOR EACH MOVE:

(2) Ask OPPONENT its scoresFor at ply-1

*This means:
- You do the move
- The opponent runs its scoreFor method (with ply-1)

these are all of the opponent's calls to scoresFor!
(0) Suppose you're playing at 2 ply...

(1) You need to make ALL moves!

AND FOR EACH MOVE:

(2) Ask OPPONENT its scoresFor at ply-1

(3) Compute which score the opp. will take

(4) Compute what score you get...

(5) Clean up before analyzing another option

\[
\begin{bmatrix}
0, 0, 0, 0, 0, 0, 0 \\
0, 0, 0, 0, 0, 0, 0 \\
0, 0, 0, 0, 0, 0, 0 \\
0, 0, 0, 0, 0, 0, 0 \\
0, 0, 0, 0, 0, 0, 0 \\
50, 50, 50, 50, 100, 50, 50
\end{bmatrix}
\]

(3) What score will the opponent choose? (tie-break?)

(4) What score does this get you as a result?

max(S) = 0

max(S) = 0

max(S) = 0

max(S) = 0

max(S) = 0

max(S) = 0

max(S) = 100

max(S) = 0

max(S) = 100

max(S) = 100

max(S) = 100

max(S) = 50

These are all of the opponent's calls to scoresFor!
**The C4Player class: where to start!!!**

(specially if you still feel you need to study about recursion)

**Board**

Board(int width, int height)

boolean allowsMove(int col)

void addMove(int col, char checker)

---

**C4Player**

C4Player(char ch, String tbt, int plyIn)

String toString()

char oppCh() // opposite checker than current turn

double scoreBoard(C4Board b)

double[] scoresFor(C4Board b)

int tiebreakMove(double[] scores)

int nextMove(C4Board b) // gives the column index where the computer wants to play given a board b ...

---

**HOW DO I DECIDE THAT!!?!?!?!
MAKE A “DUMMY” decision to START to SET UP EVERYTHING:**
- No need of oppCh scoreBoard and scoresFor fot that
- Create the C4 Player class
- Add an instance variable to your ConnecFour class (e.g. C4Player ia_player)
- When the user “clicks” instead of making just that move, you should also find out and run the ai_player move.