Java for Game AI
3: Data Structures

Raluca D. Gaina – r.d.gaina@qmul.ac.uk

http://gameai.eecs.qmul.ac.uk

Queen Mary University of London

Outline

- Arrays, lists, queues and sets
- Maps, Hash vs Tree
- Trees and Graphs
- Visualisation and Testing

Quick Overview

- Classes are a great way to group properties and behaviours relevant to a particular object type.
- What about grouping instances of objects?
- These can quickly get unwieldy, if more than a couple are needed.
- What about keeping some structure in the data, e.g. ordering of elements, sorting based on properties?

- **Solution**: data structures!

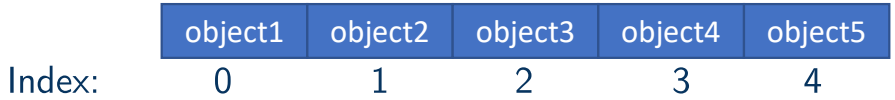
- **Disclaimer for today's exercises**: the code is not implemented/tested beforehand and instructions are written from a theoretical point of view. Therefore, as in practice, some adjustments might need to be made for everything to actually work together correctly. Updated exercises will be uploaded after the session.

3: Data Structures

Arrays, Lists, Queues and Sets

Arrays (1)

- Easy to manage list of elements of **fixed size** (an integer, largest size of array 2^{31}-1).
- In Java, treated as an object (thus passed by reference!) -> .clone() to copy.
  - **Shallow copy**: a new array object with the same contents (clone method makes a shallow copy). This is enough if your array contains *primitives*.
  - **Deep copy**: a new array object with copies of the contents all the way down (needs custom implementation).
- Can be of any type, and are declared as follows (empty brackets always indicate an **array** type):
  - Object[] array; // a variable named "array" which contains several objects of any type
  - int[] array; // a variable named "array" which contains several integers
- To create an array object, we use the new keyword and specify its size:
  - String[] stringArray = new String[5]; // an array holding 5 objects of type String, indexed from 0 to 4



- All objects in the array are given a default value: 0 for integers, false for boolean, '\0' for characters, null all others. In Java, ***null*** means the lack of an object (equivalent to ***None*** in python).
- Arrays are always indexed from 0.

Arrays (2)

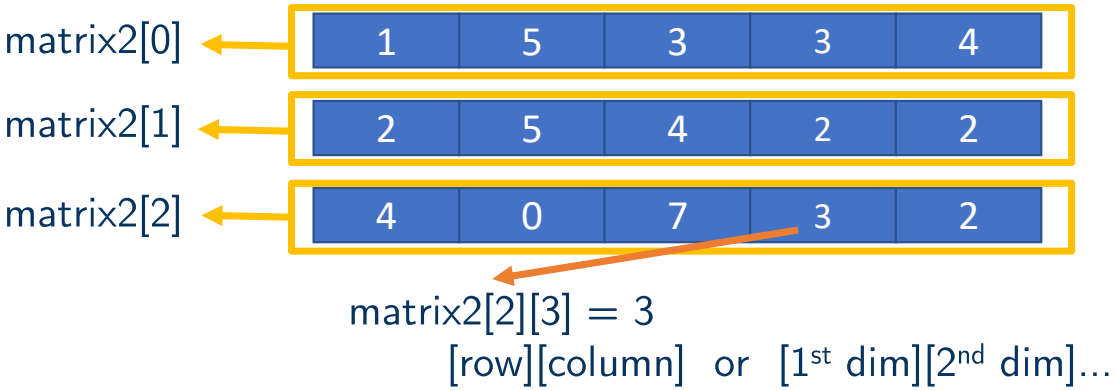
- Values can be accessed or changed by index:
  - **stringArray[0]** returns "hi"
  - **stringArray[3] = "where"**; // replace the item at index 3 (previously "who") with the new item "where"

  - stringArray[-1] or stringArray[5] would cause IndexOutOfBounds errors.
  - stringArray[4] = stringArray[3];
    - Assigns the **reference** to the object at index 3 to index 4 as well; changing either object will change the other index as well.
- Arrays can be initialised at the same time as they are created:
  - String[] stringArray = new String[] {"hi", "or", "boo", "who", "me"};
  - Array size is set based on the number of values. All values must match the type specified.
- For loops can be used to iterate through an array:
  - for (int i = 0; i < stringArray.length; i++) {
    System.out.println(stringArray[i]);
    }
  - for (String element: stringArray) {
    System.out.println(element);
    }

Arrays (3)

- Useful operations with arrays:
  - Arrays.toString(arrayVariable); // returns a String in the format ["element1String", "element2String"...]
    - Printing an array variable directly would give you the object reference and not its contents.
  - arrayVariable.length; // returns an integer, the length (or size) of the array; last index is length-1
  - Arrays.sort(arrayVariable); // sorts the given array in ascending order (or use 2nd argument Collections.reverseOrder())
    - To use custom definitions for what an object means to be less or bigger than another, the types in the array must implement the **Comparable<T>** interface.

- No multi-dimensional arrays, but you can have arrays inside other arrays to simulate this:
  - double[][] matrix1 = new double[3][]; // This is a 2D matrix with 10 rows of variable length each
  - double[][] matrix2 = new double[3][5]; // This is a 2D matrix with 10 rows of fixed length 5
  - GameObject[][][] wild; // a 3D array



Generics

- Explicit specification of class types for *templated* classes, which allows for more robust and stable code, with bugs due to type mis-match caught at compilation time, rather than run time.
- A *template* class uses capital letters (most often T, K, P) in angle brackets (<T>) to indicate that it is happy to accept objects of different types, although potentially treat them differently:
  - On the previous slide, there was one example:

```
package java.lang;
import java.util.*;

public interface Comparable<T> {
    public int compareTo(T o);
}
```

  - In the case of comparable, T is going to be the class implementing the interface, for example:

```
public class Bomb extends GameObject implements Comparable<Bomb> {
    @Override
    public int compareTo(Bomb o) {
        if (life < o.life) return -1;
        else if (life > o.life) return 1;
        return 0;
    }

}
```

- More on generics: https://docs.oracle.com/javase/tutorial/extra/generics/index.html

Slide 8

RG1 Raluca Gaina, 03/11/2020

ArrayLists (1)

- Back to arrays, what if we need a list without a fixed size?
- The **ArrayList** class offers exactly this: a dynamic list where you can add and remove elements.
- To create an ArrayList object (note the use of generics to specify object type accepted)
  - ArrayList<Integer> arrayList = new ArrayList<>();
  - ArrayList<Integer> arrayList = (ArrayList<Integer>) Arrays.asList(1, 2, 3);
  - ArrayList<Integer> arrayList = new ArrayList<>(otherCollection);
  - ArrayList<Integer> arrayList = new ArrayList<Integer>() {{ add(1); add(2); add(3); }};
- To manipulate lists:
  - arrayList.add(3); // Adds at the end of the list
  - arrayList.add(0, 5); // Adds number 5 at index 0 (at the front of the list)
  - arrayList.remove(2); // Removes element at index 2
  - arrayList.remove(Integer.valueOf(2)); // Searches for and removes element "2"
  - arrayList.get(2); // Returns the element at index 2
  - arrayList.set(2, 7); // Sets the element at index 2 to be the number 7
- ArrayList cannot be used with primitives, so use object equivalents for Integer, Double, Boolean etc.

ArrayList (2)

- We can also check if the list contains an object, if the list is empty, clear the list etc.
- ArrayLists can be used with *for* loops just like arrays.
- Some commonly used extra functionality:
  - arrayList.size(); // Returns the current size of the list
  - Collections.sort(arrayList); // Returns a sorted list; like arrays, objects can have custom comparators
  - arrayList.toString(); // Returns a formatted string, similar to Arrays.toString(array)
  - arrayList.toArray(); // Transform to array; Arrays.asList(array) is the opposite.
  - arrayList.subList(0, 5); // Returns a new ArrayList containing elements from 0 (inclusive) to 5 (exclusive)

- The ArrayList class implements the **List** interface. All lists, sets and maps we'll see next implement the **Collection<T>** interface (arrays do not).
- https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html
- Arrays are cheaper to work with, because of their fixed length, but collections are more flexible.
- ArrayLists are internally implemented using arrays.

Sets

- Like an ArrayList, but:
  - Can only have 1 copy of each object
  - Elements are **not** indexed
- **`Set`** is an interface. Commonly used implementations are:
  - **`TreeSet`**: sorted (lowest to highest), objects must implement the `Comparable<T>` interface.
    - `Set<Integer> treeSet = new TreeSet<>();`
  - **`HashSet`**: unordered, objects must implement ***equals*** and ***hashCode*** methods from the Object superclass.
    - `Set<Integer> hashSet = new HashSet<>();`
- Can be used in *for-each* loops like arrays and lists, and use similar methods as lists (add, remove etc.).

- HashSets are much faster to work with than ArrayLists (speed difference noticeable even in small programs) – if you don't need duplicates of objects or a specific ordering, always use a HashSet, e.g. for keeping a list of possible actions and checking if an action played is within those possible.

Maps

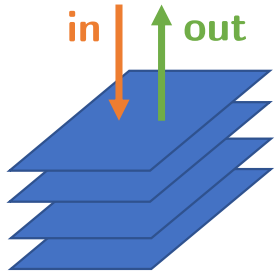
- Like dictionaries in Python.
- Store (key, value) pairs, where all keys must be unique. Can look up the key to get the value. Values can be any other types, even other maps.
- Map is an interface. Commonly used implementations:
  - **TreeMap**: sorted from lowest to highest, key types must implement the Comparable<T> interface
    - Map<Integer, String> treeMap = new TreeMap<>();
  - **HashMap**: unordered, key types must implement the ***equals*** and ***hashCode*** methods from the Object superclass
    - `Map<Integer, String> hashMap = new HashMap<>();`
    - `hashMap.put(1, "aa");` // If key 1 already exists, the value for the key is replaced with the new one
    - `hashMap.putIfAbsent(1, "aa");` // Does not replace old value if the key already exists
    - `hashMap.containsKey(1);` `hashMap.containsValue("vv");`
    - `hashMap.entrySet();` `hashMap.keySet();` `hashMap.values();`
    - `hashMap.remove(1);`
- To loop over a map:

```
for (Map.Entry<KeyType,ValueType> e: map.entrySet()) {
     System.out.println(e.getKey());
     System.out.println(e.getValue());
}
```

Other Common Collections

- **Queue** interface: first in, first out. Elements are added at the end (tail) of the queue, and the front (head) of the queue is accessed first. Methods include: `peek()`, `poll()`, `add(item)`.
  - **PriorityQueue**: orders elements in the queue in ascending order. (Often used in Dijkstra/A* implementations)
  - `Queue<Integer> queue = new PriorityQueue<>();`

- **Stack** class: last in, first out. Elements are added at the top of the stack, and the item on top is retrieved first. Methods include: `peek()`, `pop()`, `push(item)`.
  - `Stack<Integer> stack = new Stack<>();`



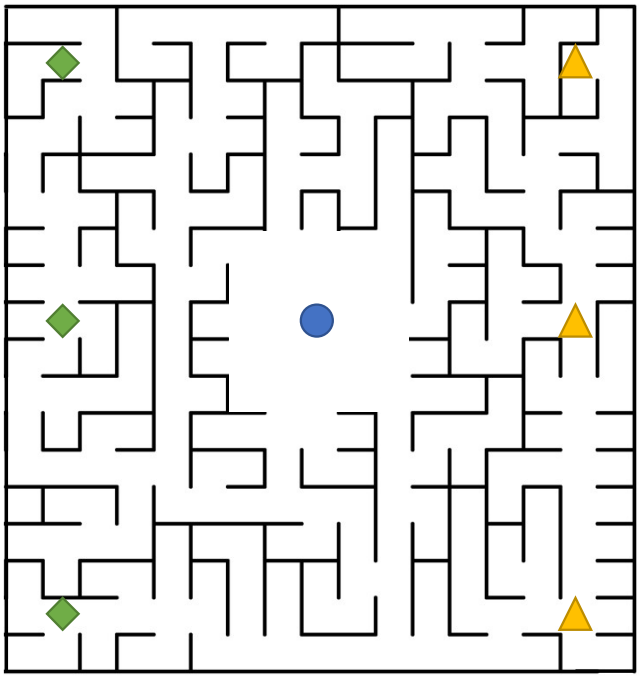
- **LinkedList** class: like ArrayList, but doubly-linked (front-back and back-front, so operations that iterate through the list will go from the end that is likely closer to the index searched for). Slower than ArrayList, but includes other useful methods, such as `getFirst()` and `getLast()`, or `addFirst()` and `addLast()` from the *Dequeue* interface (double-queue).
  - More detailed analysis of LinkedList vs ArrayList: https://stackoverflow.com/questions/322715/when-to-use-linkedlist-over-arraylist-in-java
  - `List<Integer> linked = new LinkedList<>();`

[9] Try it out! (20 minutes)

Let's continue building the maze game:

- 6 players run around in a 2D maze, as 2 teams of 3.
- They can change the team they are on at any time by stepping onto a button in the middle of the maze. Stepping on the button swaps the player's team, and has an initial 10 frames cooldown, increasing by 2 every time a players uses the button.
- Players can tag opponents by doing a "tag" action when next to 1 opponent. Tagged players are out of the game and they lose. If 2 players tag each other in the same game tick, they both lose. If player A tags player B in the same tick when they themselves were tagged, both players A and B lose. Tag actions are invalid if a player is adjacent to more than 1 opponent (adjacency does not consider diagonals).
- The last player standing wins.
- The game also ends after 1000 game ticks. If multiple players from the same team are alive at the end, but only 1 player from the other team, the 1 solo player wins and everyone else loses. If multiple players from both teams are still alive, everyone loses. If 2 players are alive at the end and on opposite teams, they tie.

[9] Try it out! (20 minutes)



[9] Try it out! (20 minutes)

1. In the core package, create a new `GameState` class, which will hold a `gameEnded` flag (boolean, true if the game has ended, false otherwise), a maximum number of game ticks constant, a game tick counter (to be initialised to 0 in the `ForwardModel.setup()`), the button object, and 6 player objects in a data structure of your choice; no values to be assigned to these variables by default. Ignore the maze (actual location of objects) for now. Add an empty default constructor.
   - Add a method named `copy` which takes no arguments and returns a deep copy of the `GameState` object. You might need to add such methods on all custom objects as well (`Player`, `Button`).
2. In the core package, create a new `ForwardModel` class, which will contain the functionality of our game in several methods:
   - `void setup (Player[] players, GameState gameState) {}`
     - Sets up the initial state of the game (within the gameState object provided), including creating the `Button` instance.
     - Assign all variables not initialised in the constructor for `Player` objects: player ID, team ID and initial positions.
     - Consider all positions to be integers going from 0 to 287 (16x18 2D grid, counting from top-left corner by rows towards bottom-right corner).
     - The `Player` class should hold a new `ForwardModel` variable, which is passed to them in this method (passing the current object, `this`).
   - `void next (int[] actions, GameState gameState) {}`
     - Ignore player actions for now. Just increase the game tick in the game state by 1, and apply the game rules:
       - Check if any player pressed the button and trigger the correct functionality.
       - Check if the game has ended. If it did, give all players their correct win status (-2 = undecided, -1 = lose, 0 = tie, 1 = win) and change the gameEnded flag in the gameState.

[9] Try it out! (20 minutes)

3. Modify the `Player` class to receive a `GameState` object in the act method (**copy** of the real one)**.**
4. Modify the `Player, Button` and `GameState` classes to implement custom toString() methods each (the Player and Button should print their current states, and the GameState class should print nicely formatted information about the state of the button and all players).
5. In the `Run` file, create an array of `Player` objects containing **6** instances of `RandomPlayer`. Then, create an instance of the `GameState` and `ForwardModel` objects. In the GameState class, set the maximum number of game ticks to **5** for this exercise.
6. Then, setup the initial game state and print it out (using the custom `toString()` method in the `GameState` class).
7. Create the main game loop, which runs until the game has ended. Within the loop:
   a) All players who are still alive are asked for an action, given a copy of the current game state
   b) Their actions are added into an array, where index in array corresponds to player ID. If the player is out of the game (was not asked for an action), add -1 in the array; otherwise, add their returned action into the array.
   c) Call the `ForwardModel.next()` method with the array of player actions and the current game state.
   d) Print the game state.
8. When the game has ended, print the win status of all players.

3: Data Structures

Graphs & Trees

Console Input/Output (IO)

- Output is simple: `System.out.println("text");`

- Input needs to *read* an *input stream* (System.in) = a stream of bytes, converted to characters:

  - `InputStream stream = System.in;`
  - `InputStreamReader inputReader = new InputStreamReader(stream);`
  - `inputReader.read();` // Can read one character input at the console at a time

  - `BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));`
  - Keeps a buffer of characters so you can read them in larger batches
  - `reader.readLine();` // Reads a full line of user console input
  - `reader.close();` // Close the reader after done to release resources

File IO (Read/Write)

- We can also read and write text from/to files, often useful in more complex applications.

- Similar to the console input.

```
try {
    BufferedReader reader = new BufferedReader(new FileReader("path to file"));
    String line = reader.readLine();
    while (line != null) {
        System.out.println(line);
        line = reader.readLine();
    }
    reader.close();
} catch (IOException e) {
    System.out.println("file not found");
}
```

File IO (Read/Write)

- A *FileNotFoundException* and *IOException* are raised at runtime if the file supplied does not exist, which crash the program. We need to state how the exceptions should be handled, in a **try/catch** block.
  - There can be multiple catch blocks. These can be empty (exception is ignored), but since exceptions occur from unexpected behaviour, you should generally know when something goes wrong and handle it appropriately.
  - `FileNotFoundException` is a subclass of `IOException`, which in turn extends superclass `Exception`, which can be used instead to catch all possible exceptions that might occur in a block of code.

- The class **`File`** can be used with a file path to obtain more information about a file, e.g.:
  - `File file = new File("myFile.txt");`
  - `file.exists();`
  - `file.getAbsolutePath();`
  - `file.isFile();`
  - `file.isDirectory();`

File IO (Read/Write)

- You can also use a simpler **try with resources:**

```
try (BufferedReader reader = new BufferedReader(new FileReader("path to file"))) {
    String line = reader.readLine();
    while (line != null) {
        System.out.println(line);
        line = reader.readLine();
    }
} catch (IOException e) {
    System.out.println("file not found");
}
```

- This handles the resource management automatically, without the need to close the reader at the end.
- To write text, use a FileWriter with a BufferedWriter instead.
- More on IO: https://docs.oracle.com/javase/tutorial/essential/io/

File IO (Read/Write)

- Some useful String operations when processing input:

- String[] splitString = String.split(" "); // Splits a text into chunks, where each chunk is separated by " "
- String s = s.trim(); // Removes leading and trailing white spaces to clean up text
- String s = s.replace('a', 'b'); // Replaces all occurrences of character 'a' with 'b' in the text.
- String s = s.substring(3, 5); // Extracts the characters from index 3 (inclusive) to 5 (exclusive)
- boolean check = s.contains("abcd"); // Checks if any of the characters in the argument appear in the text
- int aNumber = Integer.parseInt("5"); // Extracts the integer from a String, equivalents for other primitives

- In IntelliJ, type "s." to see all methods that can be applied to a String, or check out documentation (double-check your Java version): https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Recursion (1)

- The process through which a method calls itself. Such a method is called **recursive**.
- This can help with making code more compact, but also more complex to understand.
- Must be used with caution, can be deceivingly expensive to execute and can easily get stuck in infinite loops.
- Classic example, calculate factorial:

```
static int factorial(int n){
    if (n == 1)
        return 1;
    else
        return(n * factorial(n-1));
}
```

Order of computations:

```
factorial(5)
   factorial(4)
      factorial(3)
         factorial(2)
            factorial(1)
               return 1
            return 2*1 = 2
         return 3*2 = 6
      return 4*6 = 24
   return 5*24 = 120
```

Recursion (2)

- This concept can be extended to understand how an instance of an object can contain within itself other instances of the same object to represent more complex data structures (graphs and trees!).
- Recursion can also be used to iterate through graphs or trees, given a single root node (**depth-first traversal**, this can also be implemented with a *stack* data structure. **Breadth-first traversal** is implemented with a *queue* instead):

```
public static void visitNode(Node node) {
    for (Node otherNode: node.connections) {
        if (!otherNode.wasVisited) {
            visitNode(otherNode);
        }
    }
    System.out.println(node);
}

public static void main(String[] args) {
    Node root = Node.constructTree();
    visitNode(root);
}
```

Graphs

- A data structure for storing **connected** data.
- Can be used to represent city networks, social networks or even grids.
- Two elements: **vertices** (nodes) and **edges** (connections). A vertex represents the entity (or object), and the edge represents the relationship between the entities (e.g. roads with distance to travel between cities, level of friendship between people etc.).
- **Weighted graphs**: the edges have a cost associated (e.g. length of road).
- **Directed graphs**: the edges have a direction (e.g. Bob likes Ana, but Ana doesn't like Bob); adjacency matrix is no longer symmetric in directed graphs.
- **Representations:**

	Ana	Bob	Jay	Sue
Ana	0	1	1	1
Bob	1	0	0	0
Jay	1	0	0	1
Sue	1	0	1	0

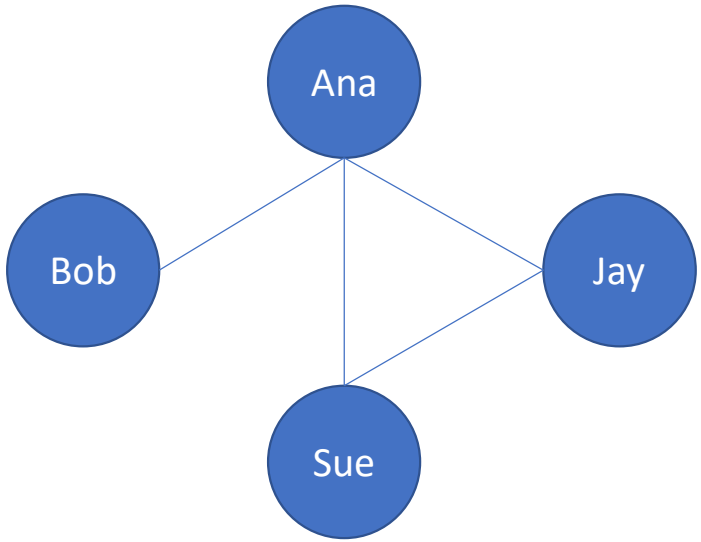
Adjacency matrix:

More space, but more efficient

Ana	->	Bob	Jay	Sue
Bob	->	Ana		
Jay	->	Ana	Sue	
Sue	->	Ana	Jay	

Adjacency list:

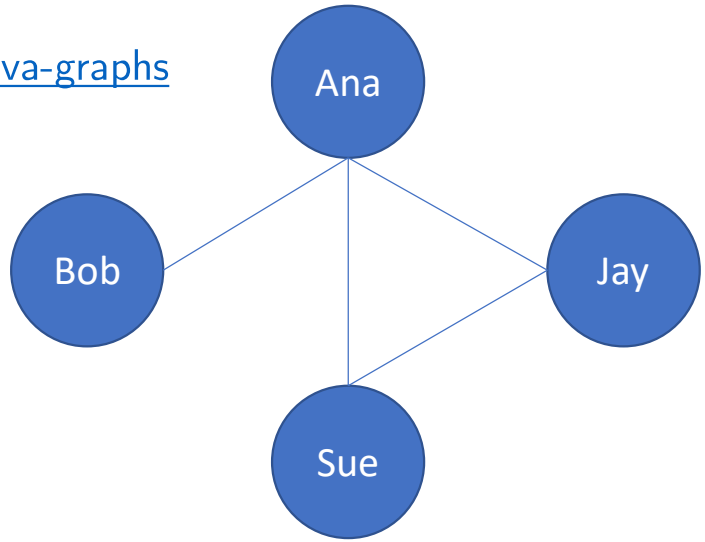
Less space, less efficient



Graphs

- A **path** through a graph is a list of nodes, where each node is connected by an edge to the nodes before and after itself in the path, e.g. Bob-Ana-Jay.
- An **entry point** is a node (vertex) which can be used as the first node in a path (by default, all nodes can be entry points in a graph).
- A **cycle** is a path where exactly 1 node appears twice, at the beginning and the end, and all others appear only once, e.g. Ana-Jay-Sue-Ana.
- A graph which contains no cycles is called *acyclic*.

- More information on graphs (in Java): https://www.baeldung.com/java-graphs



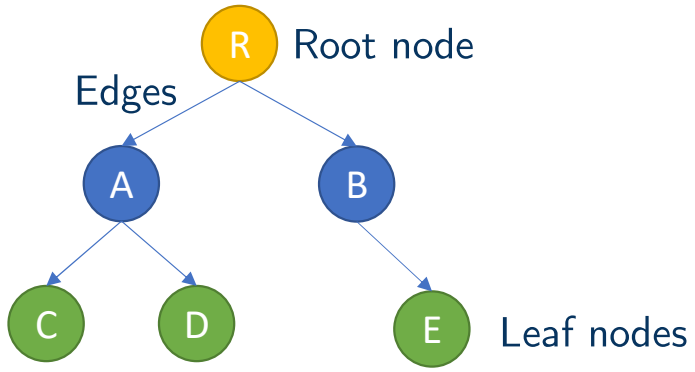
Graphs in OOP/Java

- Following OOP concepts, we only need to define one object to represent graphs: a node.
- The Node class can then hold a list of other Node objects it connects to through edges exiting the node.
  - Q: Why only exiting?
- **OR**: A Graph class contains mapping from each node object to a list of nodes it connects to.
  - Either representation is fine, and efficiency depends on specific applications and the way you need to access the data (i.e. does the node need to know who its neighbours are?)
- If edges have costs associated with them, we can instead use a map to store connections, containing all the nodes connected and costs to reach them. If more information is needed about an edge, we can add an edge object and use this instead as the value in the mapping.
- We can add access methods in the Node class to add or remove connections as well (or in the Graph class instead, depending on chosen representation).

- And that's it! We only need a list of all our nodes in the graph with connections created.
- This list of nodes can then be used in e.g. pathfinding algorithms, or other processing necessary.

Trees

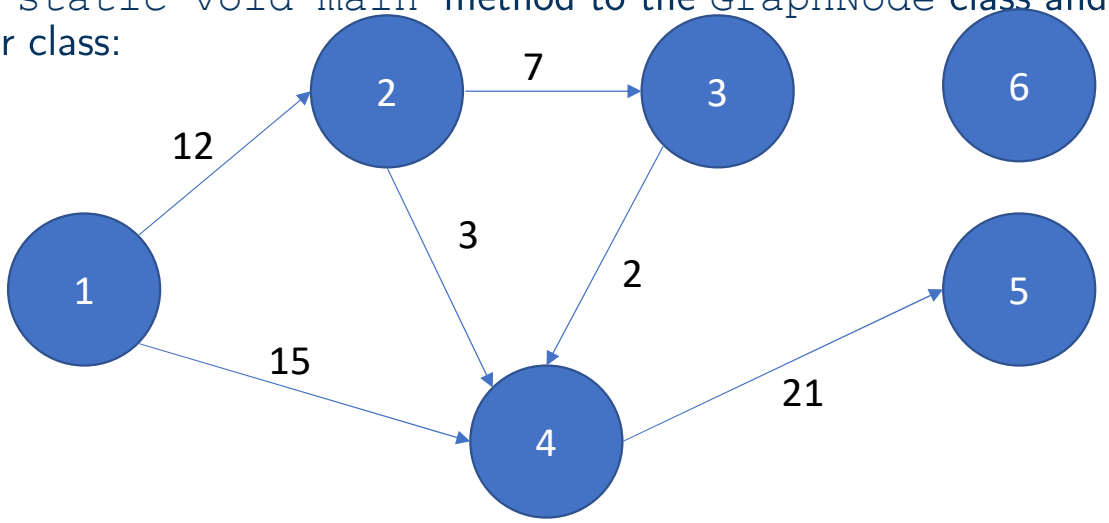
- A special type of directed graph with only 1 entry point; only 1 edge must enter any one node, but many edges can exit the node. This turns the graph into a hierarchical structure. Think of geological trees structures.



- **Root**: entry point to the tree.
- Edges: connections from one node to another, **parent** and **child**. The parent is closer to the root in the tree hierarchy (e.g. A is parent of C and D; E is child of B).
- **Leaf nodes**: nodes that do not have any children.
- **Binary trees**: a special type of tree, where each node has exactly 0 or 2 children.
- **Symmetric tree**: all nodes have the same number of children. (otherwise, asymmetric)
- To implement trees in Java, the same approach as for graphs can be followed, but we only need to store the root node (all other information can be retrieved from it).

[10] Try it out! (15 minutes)

1. Implement the `GraphNode` class to represent a graph node, with a unique ID for each node and a `toString()` method that prints out its ID and the IDs of all connecting nodes.
   - Formatted printing: `System.out.printf("Hi! I am %s and have %d years.", "Joe", 35);`
     - `%d (integer), %f (floating point), %s (String)`
2. Add a `public static void main` method to the `GraphNode` class and create the following graph using your class:



3. Keep all nodes in an ArrayList and, after creation, loop over the list to print out each node.

[11] Try it out! (10 minutes)

- Instead of manually creating all the nodes, let's instead create a method which reads the adjacency list from a file and creates all the nodes automatically. Copy and paste the following into a text ("graph.txt") file at the root of your project (**outside** of the src/ directory):
  - 1: 2-12, 4-15
  - 2: 3-7, 4-3
  - 3: 4-2
  - 4: 5-21
  - 5:
  - 6:
- Write a static method in your GraphNode class that takes as input (argument) the name of a file and returns a list of GraphNode objects created based on the file. You can assume that the ":" character separates the node ID from its edges, the "," character separates edges, and each edge is defined by a node ID and an edge cost, separated by the "-" character.
- In the `main` method of the class, call this new static method. If you print out the list of objects, it should return the exact same output as before.

[12] Try it out! (20 minutes)

1. Find the "mazeGraph.txt" file, with a similar format containing a maze game graph (but: for each node, we now specify its connection in each of the 4 directions up, right, down, left, being empty if there is no connection there; adjust your code to be able to read this format instead, and store the information appropriately in the node, e.g. mapping from direction to node ID instead). The first line in this file specifies the width and height of the grid represented.
2. Add the nodes read from the file into a `HashMap` in the `GameState` class as our maze representation, mapping from node ID to the node object.
3. In the `ForwardModel.next()` method, we can now apply the player actions (and check if a player is trying to do a valid action according to our maze graph), e.g. `mazeGraph.get(playerPosition).contains(action)` – returns true if player can move in that direction, and `nextPlayerPosition = mazeGraph.get(playerPosition).get(action);`
   a) If that can't be executed, the player might try to not move, or to tag another player. In the second case, we need to check if exactly 1 opponent is neighbouring the player, and if so, apply the tagging action).
   b) The actions array contains one action per player, where the index in the array corresponds to the player ID, e.g. player ID 0 takes action at index 0. Apply all actions in the given gameState. Remember that we defined the actions as follows previously: 0 – do nothing, 1 – move up, 2 – move right, 3 – move down, 4 – move left, 5 – tag.
   c) For now, you can allow players to move on top of each other (i.e. have the same position).
4. With this in place, we can run the game! How does it work with random players?

[12] Additional Materials

- The "mazeGraphDraw" folder is a package containing several Java classes. Running the DrawGraph class from this package draws an initially empty grid of a particular size (set in the main method).
  - You can left-click and drag your mouse to draw lines on top of the dashed ones, which are seen as walls and cut off the connections between the grid cells (the click point identification is a bit off, so the lines actually drawn might not be *quite* what you intended – drawing needs to be adjusted a bit for this to be more precise).
  - A right-click would remove the last drawn line segment (a 1-cell long part of the line you just drew).
  - Closing the drawing window (NOT stopping the program) will print to console the graph corresponding to the grid you've drawn.
- The text file suggested in the exercise uses this tool to generate it. Feel free to draw your own mazes of different sizes and experiment with those too.
- In case the display is not right in your PC, adjust the parameters at the top of DrawGraph file (offsetX, offsetY, cellSize), or the dimension of the window drawn (in the getPreferredSize method in this class).

3: Data Structures

Visualisation and Testing

Java Graphics

- You need a **JFrame** to draw on the screen. You can create your own class that extends from JFrame to customize what this window will contain. Everything is held inside a *Container*, which can be obtained through the getContentPane() method call from the super class. New elements can be added to this, for example:
  - getContentPane().add(new **JPanel**()); // Add a panel to the frame (a container itself)
  - **JComponent** – base class that can be extended for custom drawing, by overriding the public void paint(Graphics graphics) method.
  - **JButton** – a button, whose functionality can be customised by adding an ActionListener
  - Others:
- The **Graphics** object allows you to draw on the screen. Cast it to Graphics2D for more functionality. This can be accessed in the paint method from the JComponent super class. Consider Graphics2D g:
  - g.**drawRect**(0, 0, 5, 6); // Draws the outline of a rectangle of size 5x6 (widthxheight) at point (0,0) on the screen.
  - g.**fillRect**(0,0,5,6); // Draws a filled rectangle. Similar methods to draw oval.
  - g.**drawLine**(5,5,20,20); // Draws a line from point (5,5) to point (20,20).
  - g.**setColor**(Color.BLACK); // See java.awt.Color, can create any custom RGBA colors with constructor
  - g.**setStroke**(new BasicStroke(3)); // Sets the brush stroke to be width 3 for any subsequent drawings.
  - g.**drawString**("Hi", 5, 5); // Draws the text "Hi" at point (5,5) on the screen.
- More: https://books.trinket.io/thinkjava/appendix-b.html

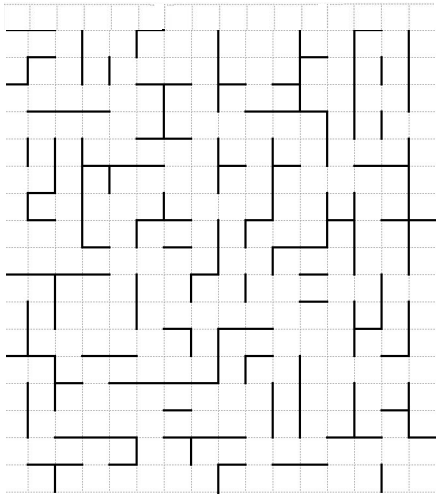
[13] Try it out! (10 minutes)

- Adapt the code found in the "graphToGridDraw/GraphToGridDraw.java" file to work with your GraphNode class and fill in all the TODOs to make the code complete and correct. Derive each node's intended position on the screen, given that its ID is encoded as follows:

$$ID = r \times w + c,$$

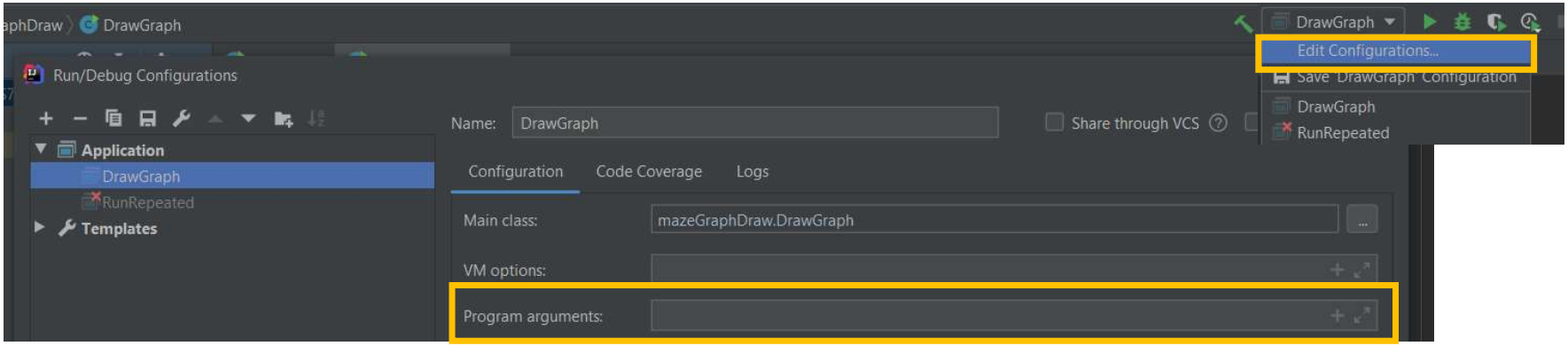
where r is the row, c is the column and w is the width of the grid

- For each node, draw its ID in the centre of its grid cell, and draw edges around the cell if a connection in that direction is missing.
- Running this with the given maze graph input should output something looking like this (with numbers in the middle of the grid cells):



Parsing & Handling Program Arguments

- We've seen this method is required in order to run a program, in at least one class:
  - `public static void main (String[] args)`
- But we've not talked about the arguments passed into this method: program arguments!
- A flexible way to parameterize a program and allow variations of it to be executed.
- These are given when first running a program. If running at command line, they are the texts separated by white space following the program name, e.g.:
  - java –jar mazeGame.jar true 2 random random
  - args[0] = "true", args[1] = "2", args[3] = "random" ...
- In IntelliJ, these can be specified in the Run/Debug configurations:



- Use String parsing operations to obtain the values (e.g. `Integer.parseInt(args[1])`).

[14] Try it out!

1. Draw a simple visualisation of the maze game state:
   - Draw the underlying grid from the graph representation.
   - Draw players that are still in the game as squares of different colours depending on their team and a number in the middle of the square to show its playerID. Ignore players that have already lost.
   - Draw the button as a circle.
   - More advanced: draw other game state information, e.g. game tick, algorithm name for each player and their win status in a separate panel in the GUI.

2. Run the game with random players with visualisation to see what happens exactly.

3. Try out a different game state representation, e.g. a 2D array. How many things would you have to modify to get things to work correctly? Which representation is easier to implement? Which one is more efficient to execute? Which one do you think is easier to work with for AI algorithms?

Acknowledgements

- Part of the material inspired by:

  MIT OpenCourseWare http://ocw.mit.edu
  6.092 Introduction to Programming in Java
  January (IAP) 2010