Lecture 18

Bits and Pieces

```
grades  = new int[5];
```

grades is a named reserved space set aside to hold exactly five **[5]** 32-bit elements all initializing to a value of zero **0**. As we have learned about programming languages, the "index" always starts at **0**, not **1**, and procedes until the size of the array is reached. In our example, since we declared **[5]** the array element index starts with **0** and ends at **4.**

	0	1	2	3	4
array element index →					
space reserved for data →	**32-bits**	**32-bits**	**32-bit**	**32-bits**	**32-bits**
value initialized in element →	**0**	**0**	**0**	**0**	**0**

index	grades
0	100
1	89
2	96
3	100
4	98

```
grades  = new int[5]; // <-- Steps 1 & 2

grades[0] = 100; // Steps 3
grades[1] = 89;
grades[2] = 96;
grades[3] = 100;
grades[4] = 98;
```

STEP 1: Declare Variable
STEP 2: Allocate Memory
STEP 3: Initialize Elements

5
grades

0	1	2	3	4
100	**89**	**96**	**100**	**98**

Off-by-one Errors

- It is very easy to be "**off-by-one**" when accessing arrays

```
// This code has an off-by-one error.
int[] numbers = new int[100];
for (int i = 1; i <= 100; i++)
      // Would work with < only
{
   numbers[i] = 99;
}
```

- Here, the equal sign allows the loop to continue on to index **100**, but **99** is the last index in the array

- This code would throw an **ArrayIndexOutOfBoundsException**

Array of Objects Declaration

Card[] deck = new Card [52]; // declares an array of type Card

- In this case, each element is initialized to null pointer.
- SO, we need to initialize each element and give it a value

```
int index = 0;
for (int suit = 0; suit < 4; suit++) {
  for (int rank = 1; rank <= 13; rank++) {
    deck[index] = new Card (suit, rank); // call the constructor
    index++;
  }
}
```

Encapsulation

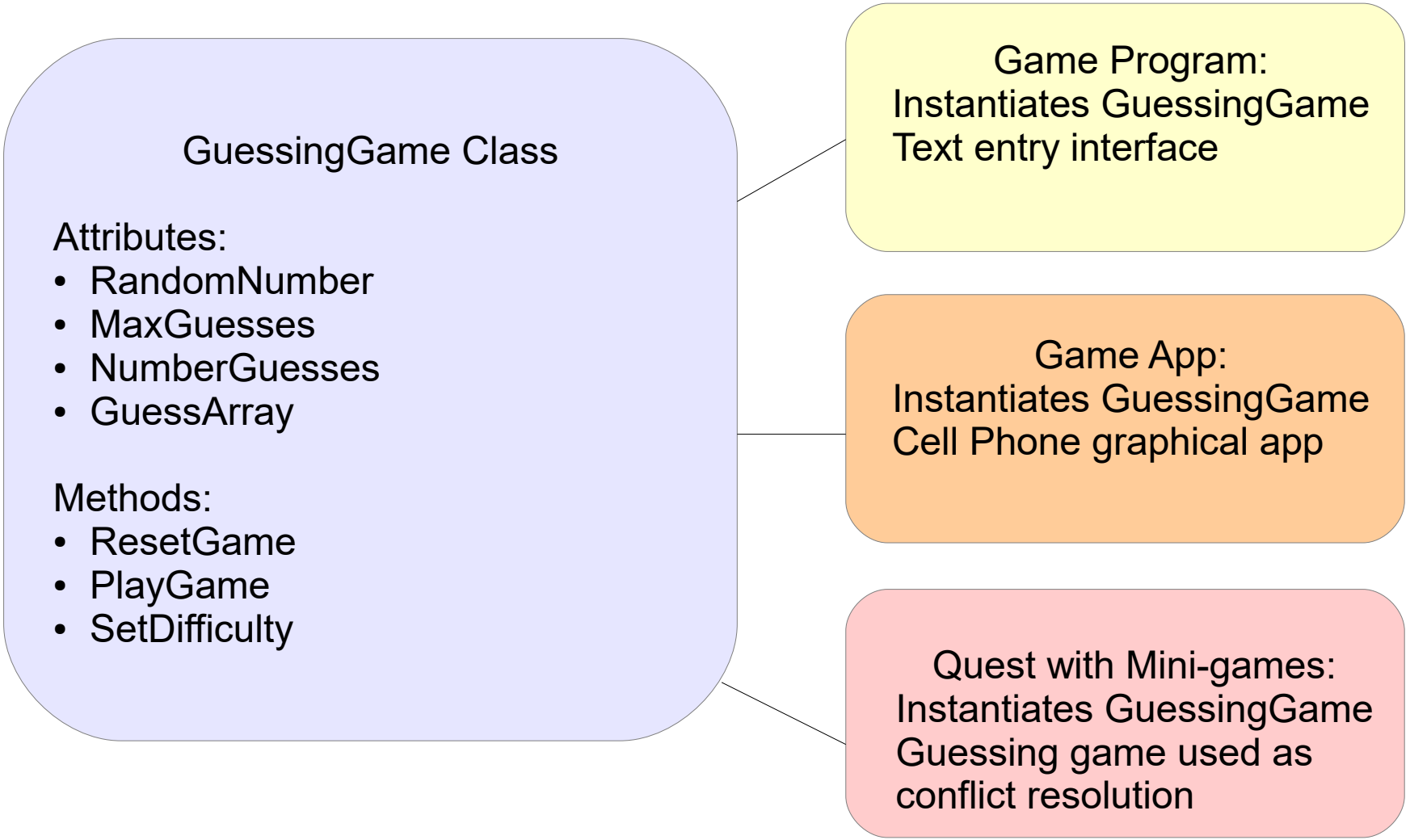
Encapsulate: to show or express the main idea or quality of (something) in a brief way,

to completely cover (something) especially so that it will not touch anything else

In programming, encapsulation refers to the bundling of data with the methods that operate on that data.

- Groups related data and methods
- Suggests protecting (making private) object attributes
- Reduces collisions of like-named variables
- Allows for refactoring by making code segments independent

Class Separation



Logical Operators

Logical Operators	
AND A && B are true only if both A and B are true	**&&**
OR A || B are true if A is true or B is true or they're both true	**||**
NOT !A is the opposite of A. If A is true, then !A is false. If A is false, !A is true.	**!**

The double ampersand **&&** and double pipe **||** are called "***short circuit" logical operators***

AND Operator

TRUE	&&	TRUE	TRUE
TRUE	&&	FALSE	FALSE
FALSE	&&	TRUE	FALSE

OR Operator

TRUE	||	TRUE	TRUE
TRUE	||	FALSE	TRUE
FALSE	||	TRUE	TRUE
FALSE	||	FALSE	FALSE

Reading in integers

Need to check for an integer

```
26        while (currentguess < MaxGuesses) {
27           while (!keyboard.hasNextInt()) {
28              System.out.println("Please enter a valid integer.");
29              keyboard.nextLine();
30           }
31           int newguess = keyboard.nextInt();
32           keyboard.nextLine();
33           if (newguess < 0 || newguess > 15) {
34              System.out.println("Please enter an integer between 0 and 15.");
```

Need to clear buffer for next entry

Should also check for the validity of the guess

Printing part of an array

"<=" means we include this guess

Note: We aren't printing the entire array.length

```
20    private void printPrevGuesses (int currentguess) {  // currentguess is 0 based
21       for (int i = 0; i <= currentguess; i++) {
22          System.out.print ("Guess " + i + ": " + guessList[i] + ", ");
23       }
24       System.out.println(""); // put in a return
25       System.out.println(MaxGuesses-currentguess-1 + " guesses left!");
26    }
```

There is some funny math here because arrays are Zero-based but our human number sense isn't!

Class / Instance Variables

```
 5 class MazeRunner extends Robot {
 6    // this is a class variable, all MazeRunners have the same name.
 7    static public String name = "Ridley";
 8
 9    //keyIn is an instance variable so any method of MazeRunner can use it without
10    //needing to create a new one each time.
11    private Scanner keyIn = new Scanner(System.in);
12    // instance variables to store step counts
13    private int numSteps;
```

In main:

```
System.out.println (MazeRunner.name);
```

Produces:

```
----jGRASP exec: java MazeWorld
  | Ridley
```

Break / Continue / Return

- A `break;` in a loop code block exits the loop and proceeds to the first line after the loop
- A `continue;` in a loop code block returns to the conditional statement to possibly restart the code block
- A `return;` exits a method and returns to the calling method
  - You send data back to the caller with `return(<something of return data type>);`

Refactoring

- Refactoring is the process of refining code, and restructuring it so it is simpler, more elegant, more efficient, easier to maintain, etc.
- Look for lines of code doing basically the same thing
  - Put them in a loop, or in a new method
- Look for two functions doing the same thing
  - Add a helper method to do that
- Look for functions that have too many jobs
  - Extract out sub jobs
- Move something to a super class
  - If you have two very similar classes, they can both inherit from one super class
- Simplify your conditionals