

# Lecture March 26 - Dynamic Array

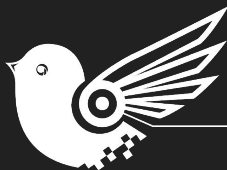
Bentley James Oakes

March 26, 2018

# COMP 101: Intro to Majoring in CS

**March 28 4:30-6pm ENGTR 1090**

**Food, course  
selection  
tips, and a  
Q&A session  
with current  
CS majors!**



**McGill**

School of Computer Science

- Monday, March 26th - Today
- From 15:00 to 17:00
- McConnell Building - Room 321

# This Lecture

1 Recap

2 Apartment Example

3 Adding a Player

4 Data Structures

# Section 1

## Recap

Might want two different methods in the same class to have the same name  
But have different parameters

For example:

- Changing a method's algorithm depending on the types
- Different constructors

# Overloading Details

- Java allows overloading based on the changes in method **parameters** (# and type)
- So `public static int add(int i, int j)` and `public static double add(double a, double b)` are okay
  - Java automatically figures out when to call the int version, and when to call the double version

```
public static int add(int a, int b){
    return a + b;
}

public static double add(double a, double b){
    return a + b;
}

public static void main(String[] args){

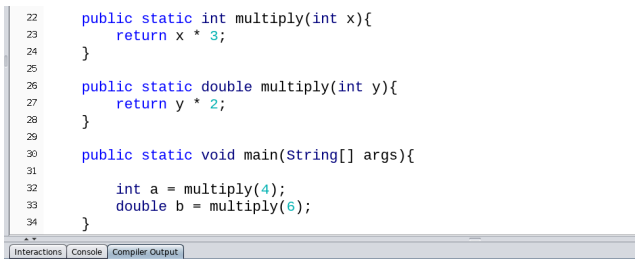
    System.out.println(add(12, 15)); //27

    System.out.println(add(15.6, 34.8)); //50.4
}
```

# Overloading Not Working

- Changing the return type and static/non-static of a method doesn't allow overloading
  - Java can't tell which version of the method to call
  - `public int add(int a, int b)` and `public String add(int i, int j)` are called the same way, so this isn't allowed

```
22     public static int multiply(int x){
23         return x * 3;
24     }
25
26     public static double multiply(int y){
27         return y * 2;
28     }
29
30     public static void main(String[] args){
31
32         int a = multiply(4);
33         double b = multiply(6);
34     }
```



**1 error found:**

**File:** /home/dcx/Dropbox/COMP 202/Lecture 17 - More OOP/Code/OverloadingExample  
[line: 26]

**Error:** method multiply(int) is already defined in class OverloadingExample



# Overloading for Constructors

```
public class Painting{
    public String artist;
    public double value; //in millions

    //constructor if we know the artist and value
    public Painting(String artist, double value){
        this.artist = artist;
        this.value = value;
    }
    //constructor if we don't know the artist
    public Painting(double value){
        this.artist = "Unknown";
        this.value = value;
    }

    public static void main(String[] args){
        Painting starryNight = new Painting("Van Gogh", 6.2);
        Painting sunset = new Painting(1);

        System.out.println("Artist: " + starryNight.artist);
        System.out.println("Value: " + starryNight.value);
    }
}
```

# The toString method

Returns a String when the instance is passed to a print method

Must have the header: `public String toString()`

```
public String toString(){  
    return "Artist: " + this.artist + " Value: " + this.value;  
}
```

//main method in Painting class

```
public static void main(String[] args){  
    Painting starryNight = new Painting("Van Gogh", 6.2);  
    Painting sunset = new Painting(1);
```

```
    System.out.println(starryNight);  
    System.out.println(sunset);
```

//before adding toString method

//Painting@27dbfe7e

//Painting@53ecb0f7

//after adding toString method

//Artist: Van Gogh Value: 6.2

//Artist: Unknown Value: 1.0

```
}
```

# Arrays of Paintings

Let's put a Painting in an array

And use `Arrays.toString()`

Calls `toString` on each element in the array

```
Painting[] pArr = new Painting[3];
System.out.println("New painting arr: " + Arrays.toString(pArr));
//New painting arr: [null, null, null]

pArr[1] = new Painting("Picasso", 10);

System.out.println("Painting arr: " + Arrays.toString(pArr));
//Painting arr: [null, Artist: Picasso Value: 10.0, null]
```

# Arrays of Paintings

- Here three Paintings are placed in an array
- Then we search the array to find the Painting with the highest value

```
public static Painting maxValue(Painting[] pArr){
    Painting bestPainting = pArr[0];
    for (int i=1; i < pArr.length; i++){
        if (pArr[i].value > bestPainting.value){
            bestPainting = pArr[i];
        }
    }
    return bestPainting;
}

public static void main(String[] args){
    Painting starryNight = new Painting("Van Gogh", 6.2);
    Painting sunset = new Painting(1);
    Painting nighthawks = new Painting("Hopper", 4);

    Painting[] collection = {starryNight, sunset, nighthawks};
    Painting mostExpensive = maxValue(collection);
    System.out.println("Most expensive worth: " + mostExpensive.value + " million");
}
```

## Section 2

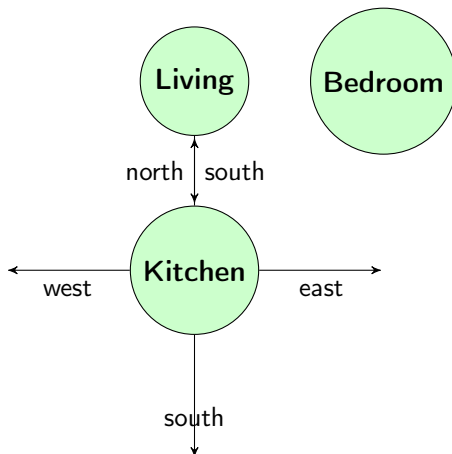
### Apartment Example

# Connected Rooms Example

- We're going to represent an apartment with a few rooms
- We'll have an `Apartment` class with `Rooms` connected to each other
  - For example, the living room will be to the north of the kitchen
- We'll store the length and width for each room, so that we can calculate the apartment's total area

# Room Connections

- We are going to connect instances of these rooms together.
- For example, we might place a living room to the north of the kitchen.
- In this case, we must store the address of the living room in the kitchen, and vice versa.



# Room Start

Here we define the room name, width and length, as well as the Room constructor.

```
public class Room{
    //name of the room
    private String name;

    //private dimensions
    private double width;
    private double length;

    //this constructor takes a room name,
    //the width and the length
    public Room(String name, double width, double length){
        this.name = name;
        this.width = width;
        this.length = length;
    }

    //return the name of the room
    public String getName(){
        return name;
    }
}
```



This method returns the area of a room.

```
//returns the area of the room
public double getArea(){
    return width * length;
}
```

- For your future programs, think about the units
- Is the area in metres? feet? yards?
- Need to write documentation so everyone uses the same units

In the Room class, have attributes for the other rooms connected to this Room instance

```
//connections to other rooms
//default value is null
private Room north;
private Room east;
private Room south;
private Room west;
```

We also need methods to set the Rooms in the four directions

```
//set the north of this room
//and set the south attribute of the other room
public void setNorth(Room other){
    this.north = other;
    other.south = this;
}

public void setEast(Room other){
    this.east = other;
    other.west = this;
}
```

# Room toString

The toString method prints a few details about the room. As well, if there is a room in a direction (as in the room is not null), then that room's name is printed

```
public String toString()
{
    String s = "";
    s += "Room: " + name + " Area: " + getArea();

    //check to see if there's a room to the north
    //if so, print north's name
    if (north != null){
        s += "\nRoom to the north: " + north.name;
    }
    if (east != null){
        s += "\nRoom to the east: " + east.name;
    }
    if (south != null){
        s += "\nRoom to the south: " + south.name;
    }
    if (west != null){
        s += "\nRoom to the west: " + west.name;
    }
    return s;
}
```

In a main method, let's start by creating three rooms

```
public static void main(String[] args)
{
    //create a room named kitchen
    Room kitchen = new Room("Kitchen", 10, 12);
    System.out.println("Area of the kitchen: " + kitchen.getArea());

    //create more rooms
    Room living = new Room("Living", 20, 40);
    Room bedroom = new Room("Bedroom", 100, 3000);
}
```

Next, let's connect the rooms together

And to test, print out the kitchen

```
kitchen.setNorth(living);  
living.setEast-bedroom);  
  
System.out.println(kitchen);
```

*Room: Kitchen Area: 120.0*

*Room to the north: Living*

# Placing Rooms in an Array

These rooms can then be placed in an array. We then iterate through the array and print out each room. Make sure to check to see if an entry in the array is *null*

```
//an array that stores rooms
Room[] rooms = new Room[4];
rooms[0] = kitchen;
rooms[1] = living;
rooms[2] = bedroom;

//print out the details of all rooms
for (int i = 0; i < rooms.length; i++)
{
    Room r = rooms[i];
    if (r != null){
        System.out.println(r);
    }
}
```

# Apartment Area

It is also easy to iterate through the array and calculate the total area for all the rooms

```
//sum up the area for all rooms
double apartmentArea = 0;
for (int i = 0; i < rooms.length; i++)
{
    Room r = rooms[i];
    if (r != null){
        apartmentArea += r.getArea();
    }
}
System.out.println("Total square metres: " + apartmentArea);
```



## Section 3

### Adding a Player

# Adding a Player

- To make our apartment more interesting, let's add a player to walk around
- The Player will have a current room
- The user will be able to input commands using a Scanner
- This will be very similar to Assignment 5

# Player Start

Let's have the Player with a name and their current room

```
import java.util.Scanner;

public class Player
{

    //simple class with a name and currentRoom
    private String name;
    private Room currentRoom;

    public Player(String name, Room currentRoom)
    {
        this.name = name;
        this.currentRoom = currentRoom;
    }
}
```

# Player Move

Let's add methods to travel between rooms

We'll need getter methods in our Room class to access north, south, east, west

```
//get the room that is to the north of the currentRoom
//then set our currentRoom to be the northRoom
public void goNorth()
{
    Room northRoom = currentRoom.getNorth();
    this.currentRoom = northRoom;
}
|
//this method is safer than goNorth()
//as first we check to make sure that we
//are not travelling to a null room
public void goSouth()
{
    if (currentRoom.getSouth() == null)
    {
        System.out.println("You can't go south");
    }
    else
    {
        Room southRoom = currentRoom.getSouth();
        this.currentRoom = southRoom;
    }
}
```

Let's add a look command, to see which room we're currently in

```
//print the name of the currentRoom
//note that this assumes that the currentRoom is never null
//(which might not be a correct assumption)
public void look()
{
    System.out.println("I am in the " + currentRoom.getName());
    System.out.println(currentRoom);
}
```

# Player Input

A non-static method within the Player class that asks for commands, and executes them

```
public void getInput(){
    //set up a scanner that gets user input
    Scanner sc = new Scanner(System.in);
    String input = "";
    //keep looping until quit is entered
    while (!input.equals("quit"))
    {
        System.out.println("Enter a command");
        input = sc.nextLine();
        //call various methods based on the input
        //we have a game now!
        if (input.equals("look")){
            this.look();
        }
        else if (input.equals("north")){
            this.goNorth();
        }
        else if (input.equals("south")){
            this.goSouth();
        }
        else{
            System.out.println("That is not a command.");
        }
    }
}
```

# Ways to Extend This

- Many ways to extend this example
- Making a game is a great way to learn
- Things to try:
  - Add items to the room that you can pick up and use
  - Example: Read and write notes that can be dropped in each room
  - Have a monster to fight, where both characters deal randomly-calculated damage
  - Have the player head through a maze, where they have to visit different locations before they win

## Section 4

# Data Structures



In computer science, a data structure is a specialized format for organizing and storing data.

- Arrays are a data structure
  - Can store one type of elements, for a fixed size
- We'll talk about more flexible data structures
  - Dynamic-sized array class
  - Built-in `ArrayLists`
- Future course talk more about data structures
  - Stack, queue, tree

## Section 5

# Dynamic Arrays

- It's very annoying to have arrays of a fixed size
- We'll start working on a `DynamicArray`
- We'll store ints in our array
- And it'll automatically resize as we add elements

In our `DynamicArray` class, we'll have two attributes/variables:

- The int array itself
- The size (the number of elements in the array)

```
public class DynamicIntArray{  
    private int[] arr;  
    private int numElements;
```

Let's discuss the idea of the dynamic array

- 1 Create an array with a guess for the size (eg, 10).
- 2 Keep track of the number of elements that have been inserted
- 3 If the array is full:
  - 1 Create a new, larger array
  - 2 Copy all the elements over
  - 3 Set the array to be the larger array

# Dynamic Array

We'll have two constructors in the `DynamicArray`:

- A constructor that initializes the array length 10
- A constructor that takes a size as input and uses that to initialize the array

```
//constructor if the size isn't known
public DynamicArray(){
    this.arr = new int[10];
    this.numElements = 0;
}

//constructor for when there's an initial size
public DynamicArray(int size){
    if (size < 1){
        throw new IllegalArgumentException("Invalid size");
    }

    this.arr = new int[size];
    this.numElements = 0;
}
```

Let's add toString, so that we can easily print out our DynamicArray

```
public String toString(){
    String s = "[";
    for (int i=0; i < numElements; i++){
        s += arr[i] + ", ";
    }
    return s + "]";
}
```



We'll need a public method for adding elements:

- `add` - appends the element to the end of the array

And we need a private method:

- `resizeArray` - if space needs to be added
  - We'll double the size of the array
  - And copy over all the elements

```
public void add(int x){  
    //if we've run out of space  
    if (numElements >= arr.length){  
        //we need to resize the inner array  
        this.resize();  
    }  
  
    //place the new element and increase the count  
    arr[numElements] = x;  
    numElements ++;  
}
```

```
//resize method: will double the reserved space
private void resize(){
    //create a new array of double the space
    int[] newArr = new int[arr.length * 2];

    //copy over all the existing elements
    for (int i=0; i < arr.length; i++){
        newArr[i] = arr[i];
    }
    //store the new array
    this.arr = newArr;
}
```

Here are some other public methods we need:

- `public int getAtIndex(int i)` - returns the element at a particular index
- `public void replaceAtIndex(int i)` - replaces the element at an index
- `public int size()` - return the number of elements in the array
- `public boolean contains(int x)` - true/false whether x is in the array
- `public void removeAtIndex(int i)` - remove an element at a particular index
  - Requires a helper method `private void shiftOver(int i)` - shifts array to the left, starting at position i
- Methods left as an exercise to the reader - code is available on MyCourses

## Section 6

# ArrayList

- There's a built-in data structure that is essentially a `DynamicArray`
- This is an `ArrayList`
- Added bonus: an `ArrayList` can hold any reference type
  - Talk more about this next week

- Important points for ArrayLists:
  - ArrayLists don't have a fixed size
  - Can store any reference type
  - Can add, delete, and check if an object exists

- Let's create an ArrayList which stores Strings
- We'll indicate the type the ArrayList stores with <String>

```
ArrayList<String> list = new ArrayList<String>();  
System.out.println("List size: " + list.size()); //0
```



- To add elements, just use the add method

```
ArrayList<String> list = new ArrayList<String>();  
System.out.println("List size: " + list.size()); //0  
  
list.add("Hello");  
list.add("World");  
System.out.println("List size: " + list.size()); //2
```

# ArrayList Printing and Empty

- There's a toString method in ArrayLists
- We can also check if it's empty

```
System.out.println(list);  
//[Hello, world]
```

```
System.out.println("List is empty: " + list.isEmpty());  
//List is empty: false
```

# ArrayList Get and Search

- We have the get method to get an element of the ArrayList
- And the contains method to search for an element

```
String s = list.get(0);  
System.out.println("First element: " + s);  
//First element: Hello
```

```
System.out.println("Contains Hello: " + list.contains("Hello"));  
//Contains Hello: true
```

- The remove method can remove an element at a particular index
- We also have clear to remove all elements

```
list.remove(0);  
System.out.println("List size: " + list.size()); //1  
  
list.clear();  
System.out.println("List size: " + list.size()); //0
```

Just to have it in the slides, here are some useful methods for an ArrayList

- `add(Object element)`  $\leftarrow$  this method appends the element to the end of the list
- `get(int index)`  $\leftarrow$  this method returns the element at the specific index
- `isEmpty()`  $\leftarrow$  returns *true* if there are no elements in the list and *false* otherwise
- `indexOf(Object o)`  $\leftarrow$  returns the index of *o* in the list, or -1 if *o* is not in the list
- `size()`  $\leftarrow$  returns the number of elements in the list
- `contains(Object o)`  $\leftarrow$  returns *true* if *o* is in the list and *false* otherwise

This documentation will be on the final

An `ArrayList` also has `add` and `set` methods for an index:

- `add(int index, Object element)`  $\leftarrow$  adds the element into the list at the specified index. The length of the list will be one larger after
- `set(int index, Object element)`  $\leftarrow$  sets the element in the list at the specified index to the new value. The length of the list will be unchanged

An `ArrayIndexOutOfBoundsException` can be thrown if the index is invalid

## Section 7

### ArrayList Exercises

- Write a method `public static ArrayList listUnion(ArrayList a, ArrayList b)`
- The returned `ArrayList` should have all elements that are in either `ArrayList`
- Do not have duplicates in the returned `ArrayList`



- Write a method `public static ArrayList listIntersection(ArrayList a, ArrayList b)`
- The returned `ArrayList` should have all elements that are in both `ArrayLists`
- Do not have duplicates in the returned `ArrayList`

## Section 8

### CatSpa Example

Let's create a business that offers spa services for cats



We'll create an `ArrayList` of Cats, and iterate through the list

We'll create cats, who have a name, and whether they've been pampered (treated very well)

```
public class Cat{

    private String name;
    private boolean pampered;

    public Cat(String name){
        this.name = name;
    }

    public String toString(){
        String s = "Name: ";
        s += this.name;
        s += " Pampered: ";
        s += this.pampered;
        return s;
    }
}
```

# Cat ArrayList

- We can put Cats in an ArrayList
- When the ArrayList is printed, the toString method is called on each Cat

```
import java.util.ArrayList;

public class CatSpa{

    public static void main(String[] args){

        //create an ArrayList of Cats
        ArrayList<Cat> cats = new ArrayList<Cat>();

        Cat a = new Cat("Alice");
        cats.add(a);

        Cat b = new Cat("Bob");
        cats.add(b);

        System.out.println(cats);
        //[Name: Alice Pampered: false, Name: Bob Pampered: false]
    }
}
```

# Cat Pampering

- Very easy to iterate through the ArrayList
- And call methods on the elements in the ArrayList

```
for (int i=0; i < cats.size(); i++){  
    //get each cat in the list  
    Cat c = cats.get(i);  
  
    System.out.println(c.getName() + " is getting the spa treatment!");  
    c.pamper(); //set each cat to be pampered  
}  
  
System.out.println(cats);  
//[Name: Alice Pampered: true, Name: Bob Pampered: true]
```