Introduction to Objects and Abstract Data Types in Java

Objectives

- Java terms and their behaviour:
  - static
  - Access Modifiers (e.g., public, private)
  - Constructors
  - Getters and Setters
  - Overriding methods
- Identify what an Abstract Data Type (ADT) is.
- Use the List<T> ADT and some of it's implementations.
- Use the filter pattern for instances of List<T>.
- Convert instances of Integer to strings.

static in the main method

```
class Foo{

    public static void main(String [] argv){
        System.out.println("Hello, world!");
    }

}
```

- Why is this static!?
- **Definition:** static allows for a method or attribute to be used without needing to first instantiate a class
  - We don't have to crate an instance of Foo foo = new Foo(); in order to call main.
- The main method is the starting point for your program and the Java compiler needs to call it before instantiating any methods.

static in the main method

Off to the worksheet to compare the following classes!

```
class ComputeSumWithStatic{

  public static int sumNums(Integer[]
       nums){
    int total = 0;
    for(Integer num: nums){
      total += num;
    }
    return total;
  }
}

Integer[] n = {1, 2, 3, 4, 5};
int s = ComputeSumWithStatic.sumNums(n);
System.out.println(s);
```

```
class ComputeSumWithoutStatic{

  public int sumNums(Integer[] nums){
    int total = 0;
    for(Integer num: nums){
      total += num;
    }
    return total;
  }
}

Integer[] n = {1, 2, 3, 4, 5};
ComputeSumWithoutStatic nonStaticSum =
     new ComputeSumWithoutStatic();
int s = nonStaticSum.sumNums(n);
System.out.println(s);
```

Access Modifiers

- The follow
  - public: The attribute and method is accessible to any other class that instantiates, implements, or extends the class in which it resides.
  - private: The method or attribute is only accessible within the class.
  - protected and the default permissions serve other purposes.
- For this class we primarily care about public and private.

Access Modifiers

```
class AnInt{

  public int num = 5;

  //..

}

AnInt x = new AnInt();
System.out.println(x.num); // You can read from it
x.num = 3;                 // You can write to it
```

- By declaring the attribute as public we can access them by using the format:

  <instanceVar>.<attr>

```
class AnInt{

  private int num = 5;

  //..

}

AnInt x = new AnInt();
System.out.println(x.num); // "read" error
x.num = 3;                 // "write" error
```

- By declaring it as private we no longer have read/write access to the attribute.
- How do we fix this?
  - getters and setters

Getters and Setters

```
class AnInt{

  private int num = 5;

  public void setNum(int newNum){
    num = newNum;
  }

  public int getNum(){
    return num;
  }

}

AnInt x = new AnInt();
System.out.println(x.getNum());
x.setNum(3);
```

- Setters are used to change the values of private variables.
  - Good practice dictates we use the form set<VarName>.
- Getters are used to retrieve the values of private variables.
  - Good practice dictates we use the form get<VarName>.

Why use Getters and Setters?

```
class AnInt{

  private int num = 5;

  public void setNum(int newNum){
    num = newNum;
  }

  public int getNum(){
    return num;
  }

}

AnInt x = new AnInt();
System.out.println(x.getNum());
x.setNum(3);
```

- Why not leave everything public? That seems simpler!
- **Encapsulation** we want to hide as much of the internal representation of a class as possible.

Why use Getters and Setters?

```
class APosInt{

  private int num = 1;

  public boolean setNum(int newNum){
    if(newNum > 0){
        num = newNum;
        return true;
    } else{
        return false;
    }
  }

  public int getNum(){
    return num;
  }

}

APosInt x = new APosInt();
System.out.println(x.getNum());
x.setNum(3);
```

- Private variables and getters/setters allows us to control access to attributes via methods. For example:
  - We can add checks to our setters to ensure what the user is setting is valid.

Why use Getters and Setters?

```
class AReadOnlyInt{

  private int num = 1;

  public int getNum(){
    return num;
  }

}

AReadOnlyInt x = new AReadOnlyInt();
System.out.println(x.getNum());
x.setNum(3);
```

- Private variables and getters/setters allows us to control access to attributes via methods. For example:
  - We can add checks to our setters to ensure what the user is setting is valid.
  - We could only include a getter so that the user doesn't have write access to an attribute.

Constructors

```
class AnInt{

  public int num; //define

  // Constructor # 1
  AnInt(){
    num = 0;
  }

  // Constructor # 2
  AnInt(int num){
    this.num = num;
  }

}

AnInt x = new AnInt();
AnInt y = new AnInt(2);
```

- Constructors are used to: 1) initialize attributes and 2) perform any other setup instructions for a class.
- If you don't define a constructor Java will provide a default one for you.
- You can define multiple constructors with method overloading.

toSting Overrides

```
class AnInt{

  private int num;

  //..

  @Override
  public String toString(){
    return String.format("Our number is
        %d", num);
  }

}

ARAnInt x = new AnInt();
System.out.println(x);
```

- The default toString produces a string of the format ClassName@memory—address.
- If we want to convert our class to a string or print it we want to override the toString method.
- String formatting is often the easiest way to accomplish this task:
  - %d - A placeholder for ints
  - %s - A placeholder for strings
  - %f - A placeholder for floats

Abstract Data Types

- ADTs are abstract!
- They define behaviour but they do not define how that behaviour should be implemented.
  - In Java this means that we usually implement them with interfaces.
  - These define the methods, their parameters, and what they return.
  - A data structure then implements the interface and *must* implement each function in the interface.
  - Interfaces will be covered more in depth in future lectures.

The List ADT

By implementing an interface we are required to provide an implemention for each method specified by the interface in order for our code to compile.

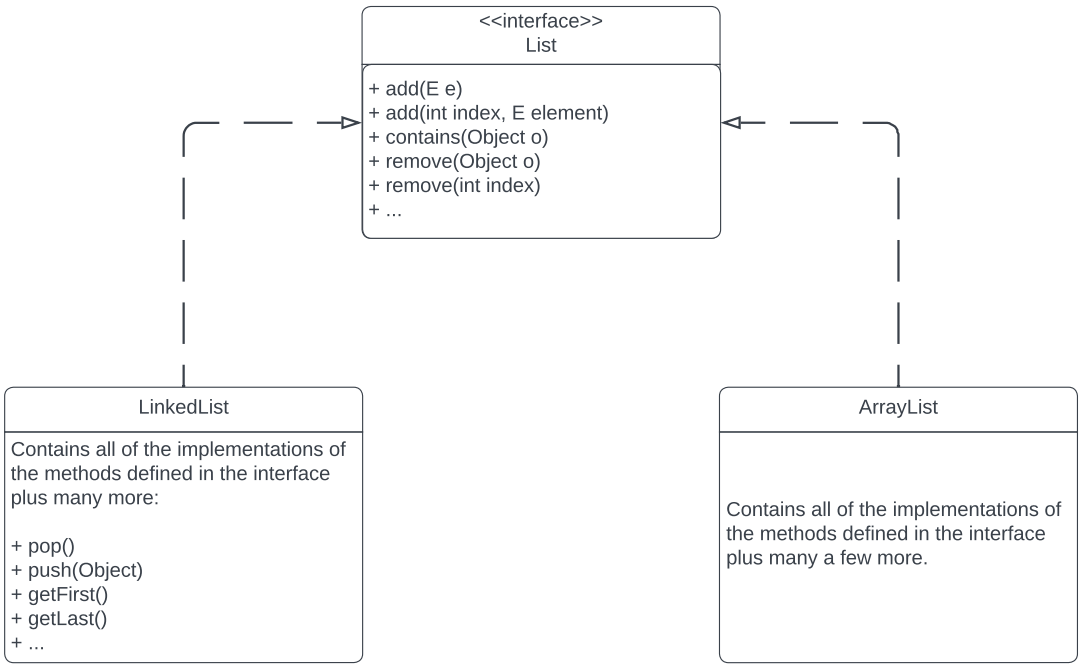
```
interface List{
    public void add(Object o);
    public void remove(int index);
    public void get(int index);
    //...
}
```

```
class ArrayList implements List{
    public void add(Object o){
        //implementation here
    }
    //...
}

class LinkedList implements List{
    public void add(Object o){
        //a different implementation here
    }
    //...
}

List<Integer> A = new ArrayList<>();
List<Integer> B = new LinkedList<>();
```

Abstract Data Types

The Listov Substitution Principle

Any subclass object should be substitutable for the superclass object from which it is dervied.



LinkedList

- When we create a linked-list via
  List<Integer> numList = new LinkedList<>() we have access to **only** the List interface's methods.
- When we create a linked-list via
  LinkedList<Integer> numList = new LinkedList<>() we have access to these methods
  - pop() - Removes an element from the top of the list.
  - push(Object o) - Adds an element to the top of the list.
  - getFirst () - Returns the first element in the list.
  - getLast() - Returns the last element in the list.
  - removeFirst()- Removes **and returns** the first element in the list.
  - removeLast() - Removes **and returns** the last element in the list.

ArrayList

- When we create an arraylist via
  List<Integer> numList = new ArrayList<>() we have access to **only** the List interface's methods.
- When we create an arraylist via
  ArrayList<Integer> numList = new ArrayList<>() we have access to some additional methods.
  - The additional methods provided by this denfition of an arraylist

Programming to Interfaces

Consider the following segment of code:

```
public float sumList(List<Integer> lst){

  int total = 0;

  for(int i = 0; i < lst.size(); i++){
    total += lst.get(i);
  }

  return total;
}

List<Integer> al = new ArrayList<>();
List<Integer> ll = new LinkedList<>();

// Fill them with numbers 1-10
for(int i = 1; i <= 10; i++){
  al.add(i);
  ll.add(i);
}

System.out.println(sumList(al));
System.out.println(sumList(ll));
```

- The two key takeaways from this segment of code:
  1. We declared these as List type rather than ArrayList or LinkedList.
     - This limits the methods we can use on al and ll to **only** those specified by the List interface.
  2. The sumList method takes a List as a parameter.

Programming to Interfaces

Consider the following segment of code:

```
public float sumList(List<Integer> lst){

  int total = 0;

  for(int i = 0; i < lst.size(); i++){
    total += lst.get(i);
  }

  return total;
}

List<Integer> al = new ArrayList<>();
List<Integer> ll = new LinkedList<>();

// Fill them with numbers 1-10
for(int i = 1; i <= 10; i++){
  al.add(i);
  ll.add(i);
}

System.out.println(sumList(al));
System.out.println(sumList(ll));
```

- **Key Takeaway #1**: Methods should take List as the parameter unless your method specifically requires methods only avaliable in the LinkedList or ArrayList classes.
- **Key Takeaway #2**: Lists should be delcared as List unless you require methods only avaliable in the LinkedList or ArrayList classes.

Algorithm: Filtering a List

Algorithm 1 Filtering a List

```
1: procedure FilterList(GivenList)
2:     NewList ← new List()
3:     for item ∈ GivenList do
4:         if item meets criteria then
5:             NewList.add(item)
6:         end if
7:     end for
8:     return NewList
9: end procedure
```

Another Example of Programming to Interfaces

```
public List<Integer> filterPosNums(List<Integer> lst){

    List<Integer> posLst = new ArrayList<>();

    for(int i = 0; i < lst.size(); i++){
        Integer item = lst.get(i);
        if(item > 0){
            posLst.add(item);
        }
    }

    return posLst;
}
```

- Allows us to filter the positive integers from any `List`.
- This algorithm doesn't require any methods specific to `ArrayList` or `LinkedList`.

Worksheet: Filtering Lists

In breakout rooms, work through the inclass activities for filtering lists for 15 minutes. We'll cover them once you get back.

Integer vs int

- int is a primative data type meaning
- Integer is a wrapper class that contains the value and methods for working with integers.
- Integer has useful static fields:
  - MAX_VALUE - The largest value an int can be.
  - MIN_VALUE - The minimum value an int can be.
- Integer has useful static methods:
  - Integer.parseInt(String str) - Attempts to convert a string representation of an integer (e.g., "123") to an integer.
- and even some useful non-static methods. If we declare an integer as Integer integerInstance = new Integer(3):
  - integerInstance.intValue() - Returns the value of the integer as a primative int (i.e., 3).
  - integerInstance.toString() - Returns the value of the integer as a string (i.e., "3").

Integer to String Conversation and Excpetions

```
String str = // something...
try{
    Integer convInt = Integer.parseInt(str);
} catch(NumberFormatException e){
    System.out.printf("%s cannot be converted to an Integer");
}
```

- We use the Integer.parseInt(str) method to convert a given string (str) to an Integer.
  - If str = "123" this works fine!
  - If str = "123asdf" this produces a NumberFormatException :(
- An exception is an error (you've likely seen these).
- We use a try-catch block to manage exception
  - We "try" the code in the try block
  - If a specified exception occurs while trying the code we "catch" that exception and run the code in the catch block