Topic 11.0: ADTs

Learning Goals:

- Differentiate between an abstract data type and a data structure.
- Implement Stack and Queue abstract data types using a linked list data structure.

Abstraction

- In the previous unit, we learned about interfaces, which describe what we can do with a class without knowing how it does it.

- Interfaces are one tool that allow us to focus on what’s important in a context and ignore the details. (There are others you will see next course)

- In Computer Science this strategy of ignoring the details is called **abstraction**.
-
- It allows us to build systems out of complex components without getting lost in the minutiae of technical details and implementation

Abstraction

- Actually, our ability to write programs already relies on abstractions.

- When we write code like: System.out.println(“hello”);
    - we know what will happen but we don’t necessarily know how it happens.

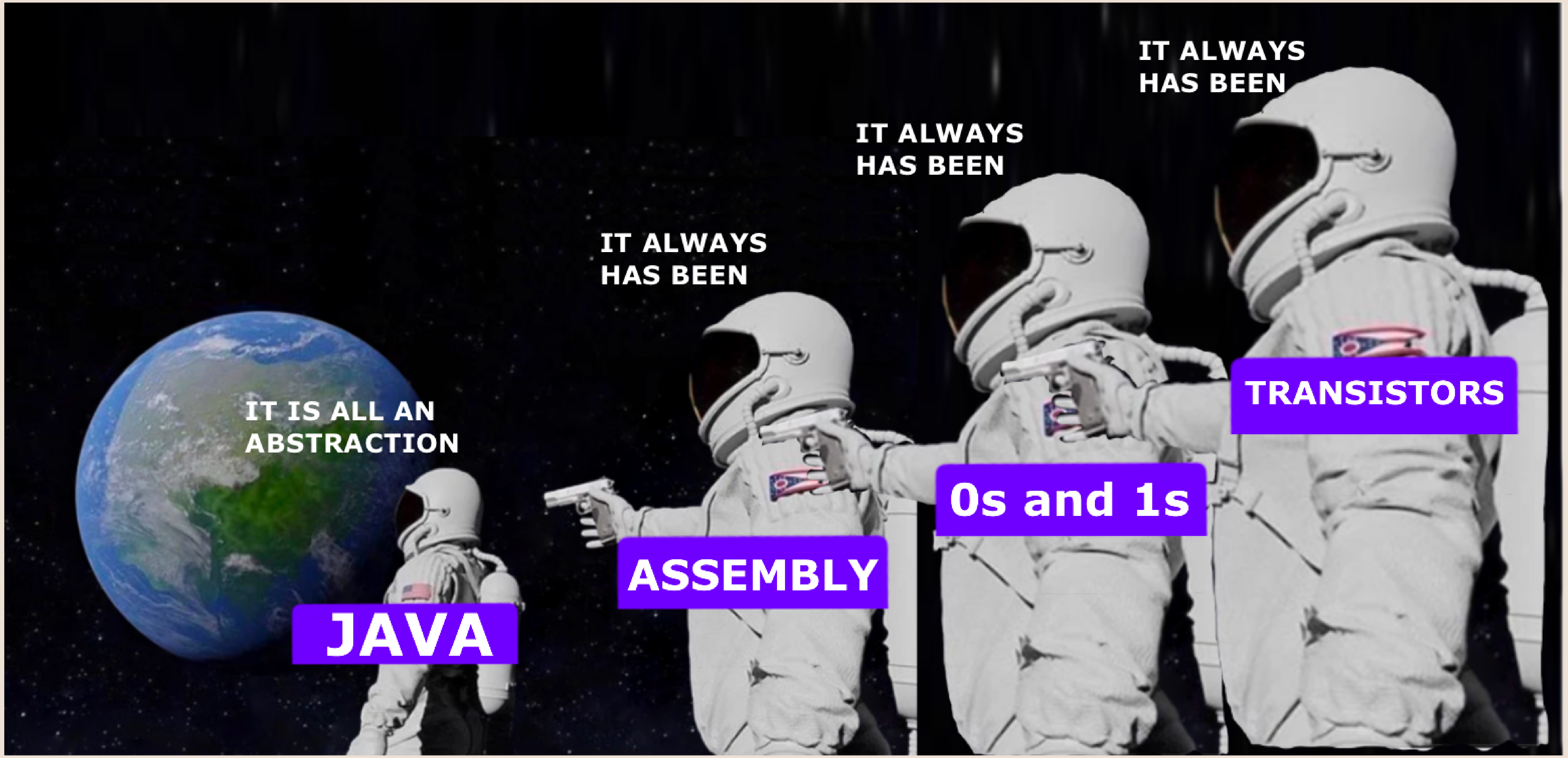
- We think that maybe the computer uses electricity and ones and zeroes and ??? (magic?)
- In the end, we see the output on our display.

Abstraction

- Another example is way back at the start, when we made a PersonList class
    - we could add and remove people from the list and whatever else
- We could use it for a ListOfStudents or a ListOfChildren or a ListOfFriends or another other List of People we could think of
- We didn’t set it up with “proper” abstraction in mind but it did work similarly.

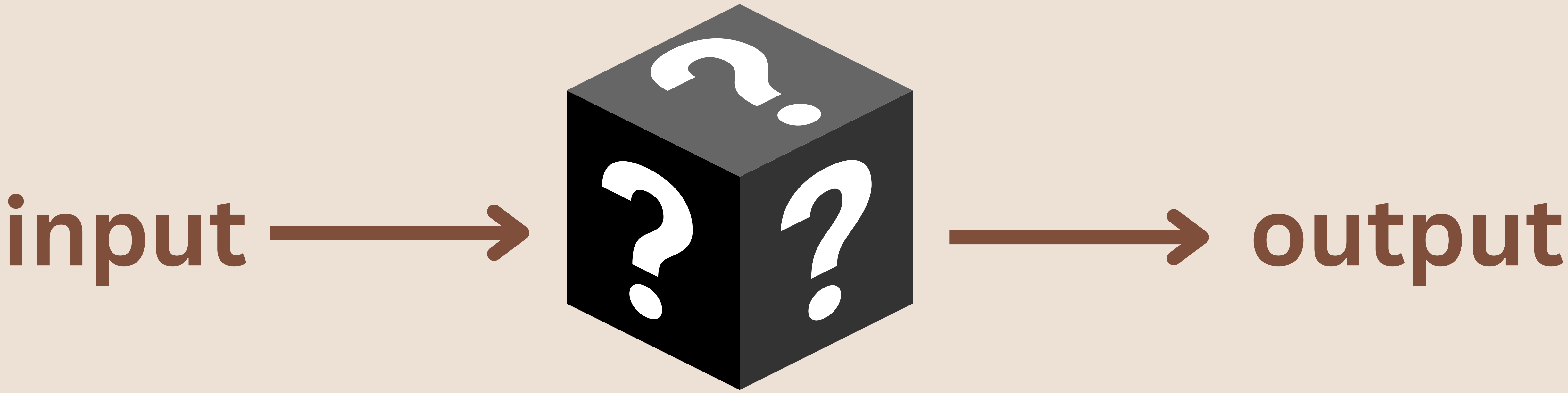
Abstraction

- Really everything we do has some level/element of abstraction to it



Abstraction

- An example of an abstraction is an “opaque box”, where data comes in, is processed, and output is produced, somehow.



input → ? → output

- We can describe the inputs and generated outputs without having to say how the box produces them.
- Example: we can know what a **sort()** function does from its name, without seeing the implementation.

Data Abstraction

- We can apply this concept of abstraction to the techniques our program uses to store data.

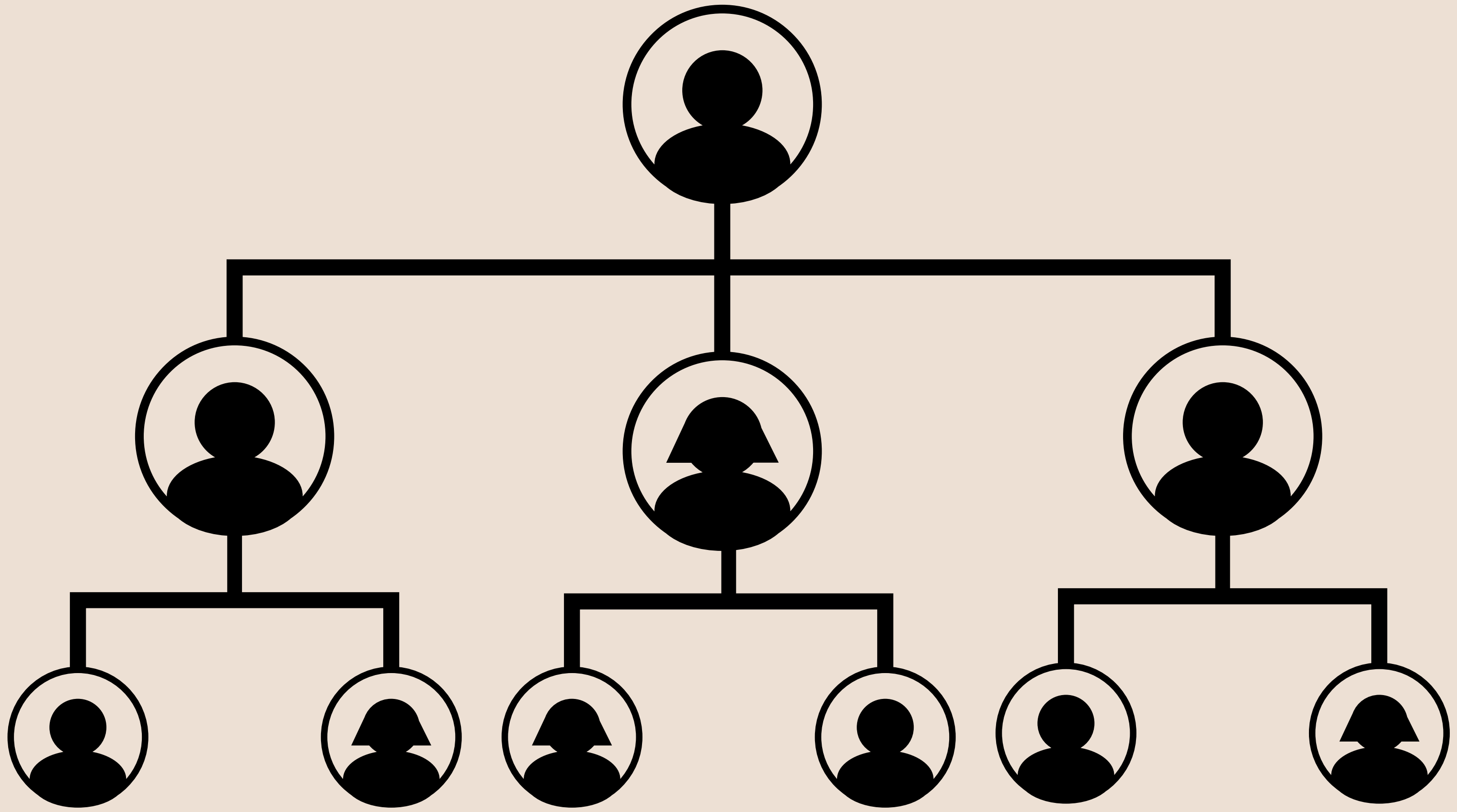
- A **data abstraction** is a description of data storage according to the organization of the data, and the operations we can perform on it.

- Basically, **a name that describes how the data is organized, and an interface.**

Data Abstraction

- e.g. **A family tree**

- How is it stored? **Who cares**
- **What can we do with it?**
  - **Add children**
  - **Remove Children (maybe ? yikes)**
  - **Connect family members**
  - **Add marriage/divorce connections**
  - **etc.**



Abstract Data Types (ADTs)

- An **abstract data type** or **ADT** is a particular example of a data abstraction.
- The ADT **List** is one that we have been using throughout this course.
    - LinkedLists and ArrayLists are private **implementations** of this ADT
- The data is in a List is organized sequentially; that is, there is a first, second, third, etc. element.
- Operations could include add, get, set, remove, size, and find.

Abstract Data Types (ADTs)

- The name and operations of the ADT form the public interface.
    - So **List** and **get(), set(), remove(), etc.**

- The private implementation uses a particular data structure.
    - **ArrayList, LinkedList**

https://docs.oracle.com/javase/8/docs/api/java/util/List.html

ADT Idea!

- An **OrderedList**
  - Guaranteed sorted order always
  - not the same as List
    - List has set() and set() could break an OrderedList implementation
- The operations are slightly different from List.
- We can still have
  - **get**, **remove**, **size**, and **find**
  - There is **no add,** but the **insert** operation will insert into a unique (ordered) position.
  - There is **no set** operation

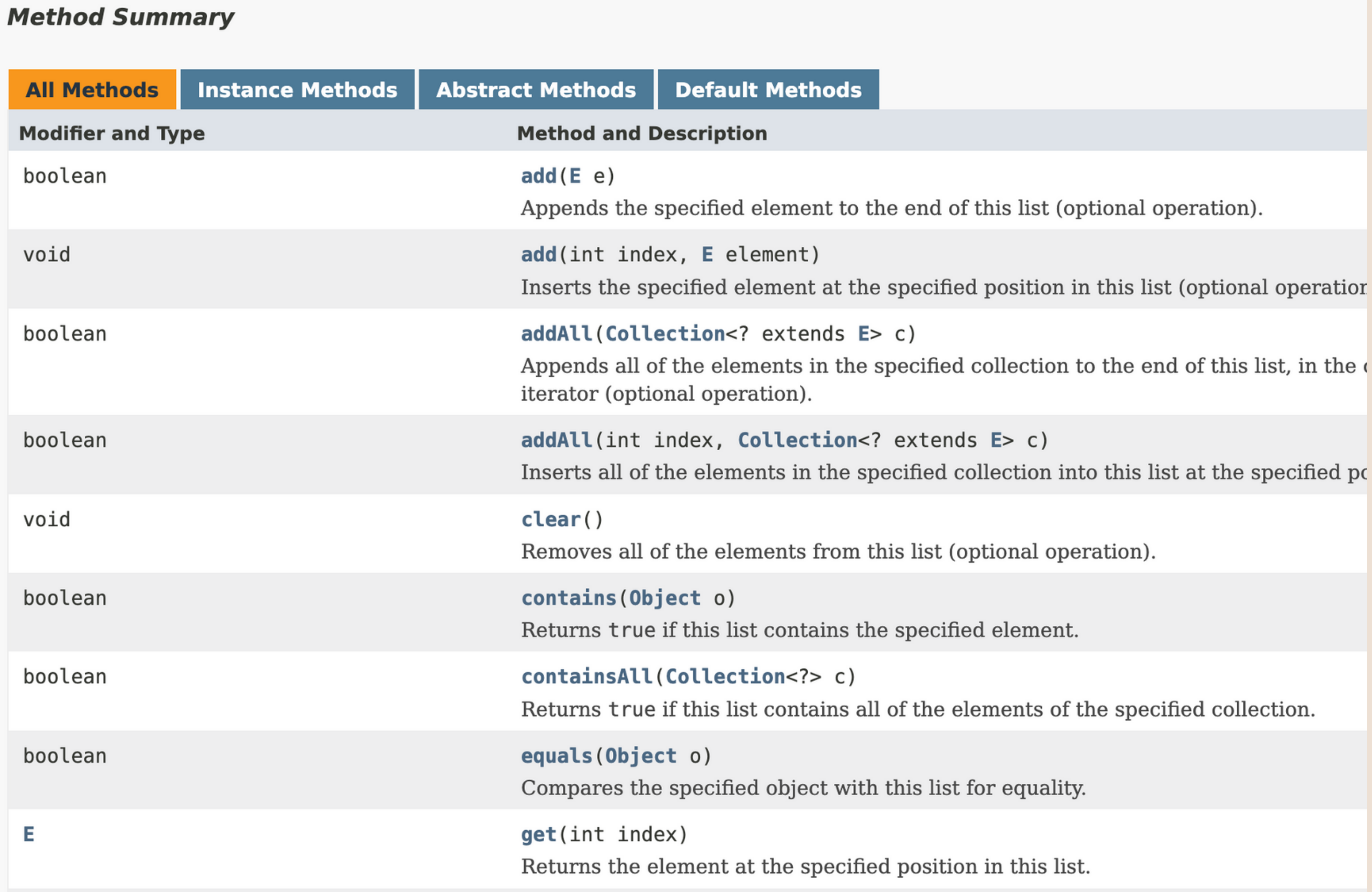
Quick Pause

- You now have an example of the ObjectList interface being implemented by my OrderedNumberArrayList implementation
- You should:
    - try to write your implementation as an OrderedNumberArray and/or OrderedNumberLinkedList
    - All you should have to do is switch the type of List that is being instantiated in main (from new OrderedNumberArrayList() to yours) and the could should still work perfectly!
- Give it a try and come back :)

ADTs and Formal Interfaces

- A programming language like Java that has formal interfaces can use them to define an ADT.

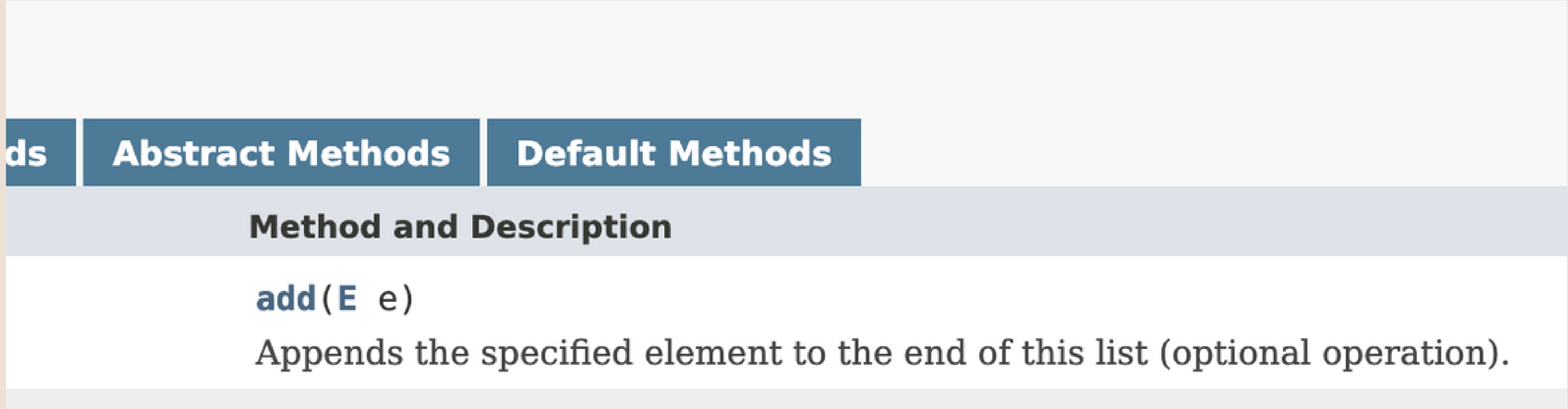
- For example, here is a snipper of Java’s built-in ADT List **interface**
    - **remember: this means we cannot make a new List() BUT we can declare variables of type List.**



Method Summary

Modifier and Type	Method and Description
`boolean`	`add(E e)` Appends the specified element to the end of this list (optional operation).
`void`	`add(int index, E element)` Inserts the specified element at the specified position in this list (optional operatior
`boolean`	`addAll(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the iterator (optional operation).
`boolean`	`addAll(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list at the specified po
`void`	`clear()` Removes all of the elements from this list (optional operation).
`boolean`	`contains(Object o)` Returns `true` if this list contains the specified element.
`boolean`	`containsAll(Collection<?> c)` Returns `true` if this list contains all of the elements of the specified collection.
`boolean`	`equals(Object o)` Compares the specified object with this list for equality.
`E`	`get(int index)` Returns the element at the specified position in this list.

A quick note on Generics



Method and Description

`add(E e)`

Appends the specified element to the end of this list (optional operation).

- You may notice Java Lists store data of type **E**
  - this is known as **Generics**
- Unfortunately these can be tricky to implement.
- Instead, we will use a simpler strategy: the Java **Object** (just like we've seen big O object before)

A bit more depth on big O Object

- An **Object** stands in for any kind of object; that is, an instance of any class.
  - all capital letter Data Types (String, Boolean, Integer, ArrayList, List, etc) fall under the **Object umbrella**
  - This means all those data types are also, technically, types of **O**bjects.
  - More on this weird umbrella hierarchy next year
- If we define our List interface to store Object so that we can put any type of object in it we like.
- Makes it a kind of General List, just like we saw with ArrayList list = new ArrayList();

Using our ObjectList class

- When we get an object out of our list, we get an **Object**, but its **actual** class is unknown.
  - We know it is under the **Object** umbrella but is it a String? A Boolean? A Student?
- We can’t call class-specific methods on obj.
  - Strings don’t have a .getGPA()
  - Booleans don’t have a .length()
  - **WHAT DO WE DO?!**

Casting Objects

- Instead, we can cast an Object to a more specific type:

```
String str = (String)list.get(0);
```

- Then we can use str like any other String.
- **But** we have to be certain that the object we got from the list is the expected type, or else this will crash with a **ClassCastException**

- Let’s make an ObjectList interface and an OrderedNumberArrayList under it (using an ArrayList<Integers> for now)
- See ObjectListExample.zip folder for complete code

Safe Casting

- One way to safely cast an object is to check using the **instanceof** operator, which gives a boolean result:

```
Object obj = list.get(0);
if (obj instanceof String) {
     String str = (String)obj;
}
```

- The **instanceof** operator produces a true result if the object is actually an instance of that class.

The Smarter Object List

- A generally better strategy is to use a generalized List of objects to build a type-specific collection class:

```
class StringList {
     private ObjectList objList;
     // constructor and other methods not shown

     public void add(String str) {
          objList.add(str);
     }
     public String get(int index) {
          return (String)(objList.get(index));
     }
}
```

OrderedList of Objects?

- We can’t easily make an OrderedList out of Object, but we can make it out of Comparable:

```
interface OrderedList {
     void insert(Comparable obj);
     Comparable get(int index);
     void remove(int index);
     int find(Comparable obj);
     int size();
     // and more …
 }
```

A general Comparable

- Because we don't use generics, **Comparable** has dropped the <Classname> part at the end of it.
- Like **Object**, this **Comparable** needs to be cast to a more specific type when we take it out of the list.
- An implementation of the **insert** method will call **compareTo**(Object other) to determine order.
- And the **OrderedList** can be implemented by an array or linked list of Comparable objects.

More ADTs (Part 2)

- There are many other ADTs used to solve problems in Computer Science.

- Next,
  - We will look at two in particular: the **stack** and the **queue**.