INFO1113 Object-Oriented Programming

Week 4B: Collections
Usage and construction of ADTs

Copyright Warning

Topics

- ADTs (s. 4)
- Lists (s. 6)
- Maps and Sets (s. 48)
- Checked and Unchecked Operations (s. 55)

Abstract Data Type

You will be **hearing *type*** a lot in this course.

An abstract data type (ADT) is a type that is an aggregation of other types. However, we may have an **ADT** that only has methods and no data.

- **Primitive types > not an ADT**
- **Reference types > is an ADT**

The language does not afford us to primitive types in an abstract way. We would need to do the hard work for it.

Collections

Our applications will require certain needs in storing and organising data. Collections or **aggregate** types allow us to store data in the appropriate objects.

For example:

- A school contains students
- A stage has performers
- Running times from athletes for race
- TV Shows on netflix

We will be visiting:

- List Types
- Set types
- Map types

What is a list?

The most common list type that is used is an **ArrayList.** This comes from the convenience of having a **resizable Array**.

There are other list types:

- **LinkedList**
- **Vector**
- **Stack**

However depending on how we store, update and access the data will determine what type we want to use.

What is a list?

Let’s take a look at this problem and see if we can solve it just using arrays.

“We want to store all input given by the user in a collection and be able to review it.”

What is a list?

Let’s take a look at this problem and see if we can solve it just using arrays.

“We want to **store all input** given by the user in a collection and be able to review it.”

Okay… slight problem here, Arrays are fixed length.

Let’s try and solve it

ArrayList (Dynamic Array)

Not to be confused with dynamically allocated array. The ArrayList data structure is a resizable array or (performs resizing for us).

Let's look at the syntax:

Syntax:

```
List<Type> name = new ListType<Type>;
```

Most collections follow a similar format.

ArrayList (Dynamic Array)

Not to be confused with dynamically allocated array. The ArrayList data structure is a resizable array or (performs resizing for us).

Let's look at the syntax:

Syntax:

```
List<Type> name = new ListType<Type>;
```

Most collections follow a similar format.

The type, in this instance, a **interface** type for **generalisation.**

ArrayList (Dynamic Array)

Not to be confused with dynamically allocated array. The ArrayList data structure is a resizable array or (performs resizing for us).

Let's look at the syntax:

Syntax:

```
List<Type> name = new ListType<Type>;
```

Most collections follow a similar format.

The type, in this instance, a **interface** type for **generalisation.**

The **type** the List will store. We want to ensure that only one kind of object is stored in this collection.

ArrayList (Dynamic Array)

Not to be confused with dynamically allocated array. The ArrayList data structure is a resizable array or (performs resizing for us).

Let's look at the syntax:

Syntax:

```
List<Type> name = new ListType<Type>;
```

Variable name

Most collections follow a similar format.

The type, in this instance, a **interface** type for **generalisation.**

The **type** the List will store. We want to ensure that only one kind of object is stored in this collection.

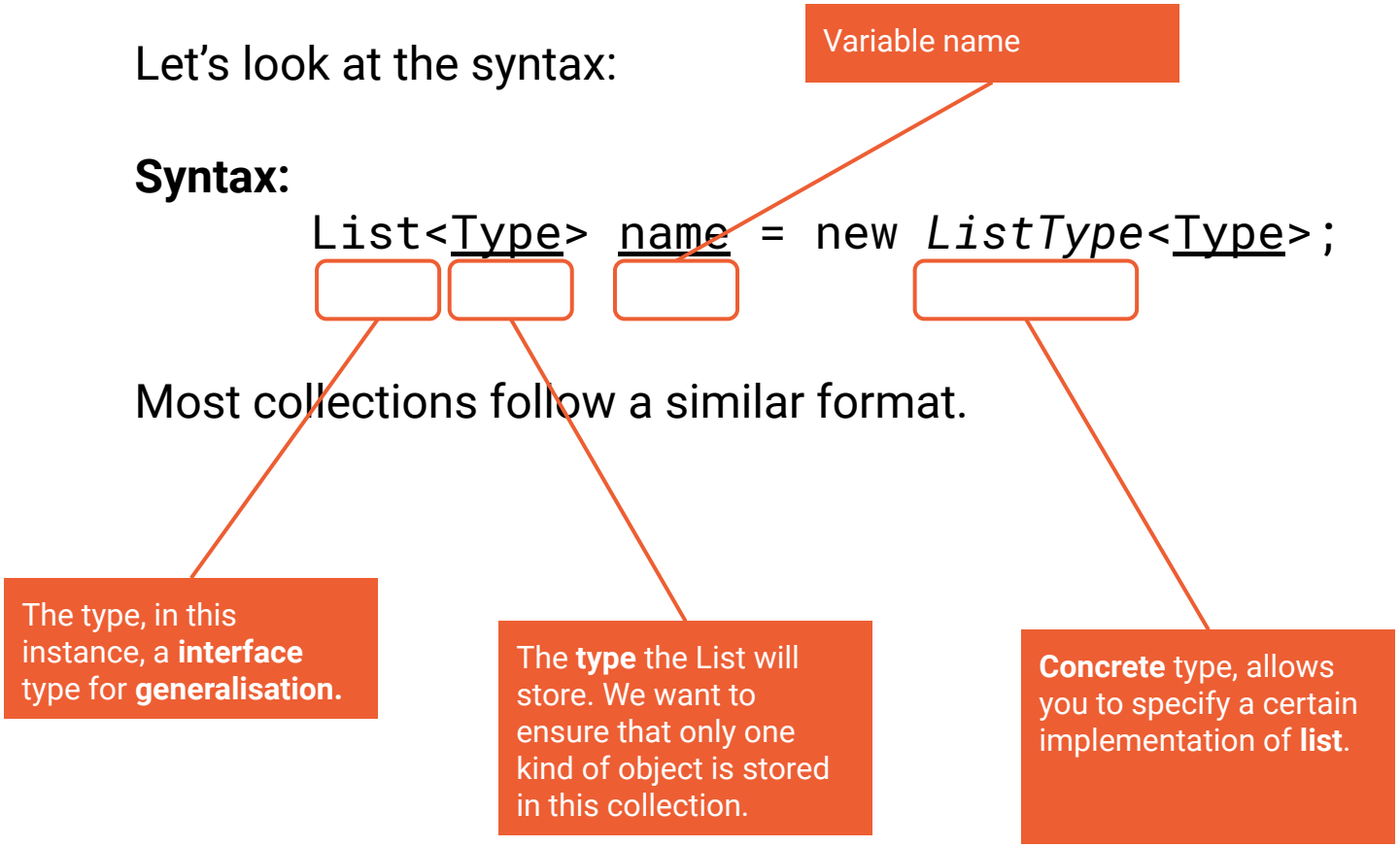
ArrayList (Dynamic Array)

Not to be confused with dynamically allocated array. The ArrayList data structure is a resizable array or (performs resizing for us).

Let's look at the syntax:

Syntax:

```
List<Type> name = new ListType<Type>;
```



Most collections follow a similar format.

ArrayList (Dynamic Array)

So let’s disassemble these operations:

```
import java.util.ArrayList;
public class Example {
    public static int main(String[] b) {
        ArrayList<String> list = new ArrayList<String>();

        list.add(“First String!”);
        list.add(“Second String!”);
        list.add(“Woof!!”);

        list.remove(1);

        list.set(1, “Meow”);
        System.out.println(list.get(0));
        System.out.println(list.get(1));

    }
}
```

Let’s fix up our code!

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)

2
8
3
4
90
12
45
32

When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)

2
8
3
4
90
12
45
32

When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)

When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

2
8
3
4
90
12
45
32

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)

2	8	3	nil	nil	nil	nil	nil

When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

2
8
3
4
90
12
45
32

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)

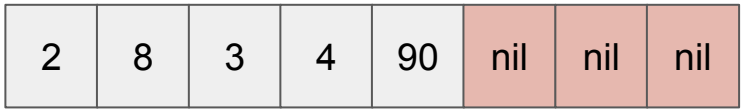
When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

2
8
3
4
90
12
45
32

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)



When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

2
8
3
4
90
12
45
32

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)

2	8	3	4	90	12	nil	nil

When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

2
8
3
4
90
12
45
32

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)

2	8	3	4	90	12	45	nil

When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

2
8
3
4
90
12
45
32

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)

2	8	3	4	90	12	45	32

2
8
3
4
90
12
45
32

When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)

2	8	3	4	90	12	45	32

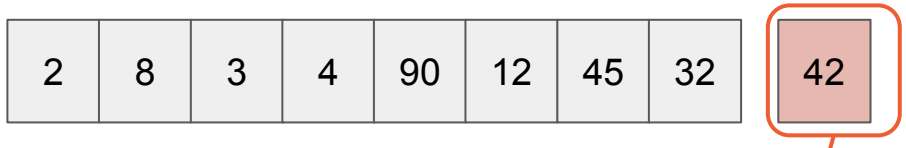
When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

Hang on! We just ran into the same problem with an Array! This is where the **resize occurs**.

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)



When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

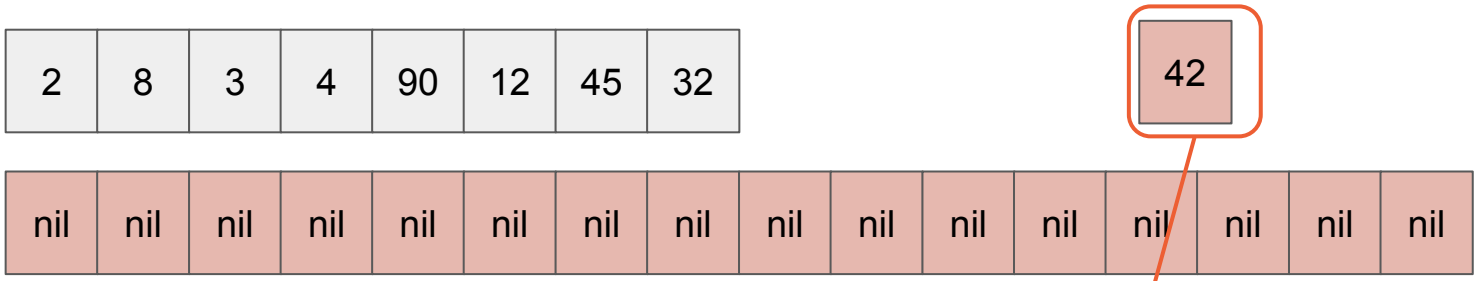
Can't simply override an element.

We will create a new array!

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)



When the **.add()** method is called, the **DynamicArray** will start adding elements into its internal array.

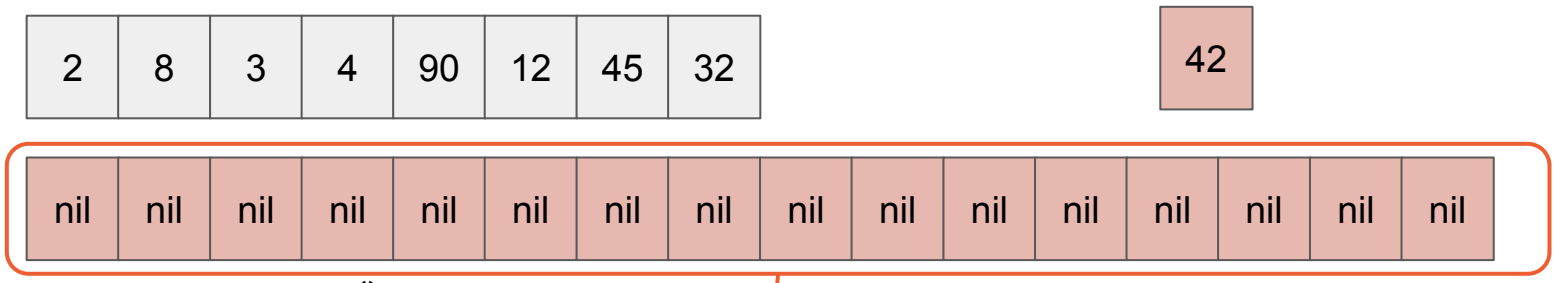
Can't simply override an element.

We will create a new array! Double the size of the old one!

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)



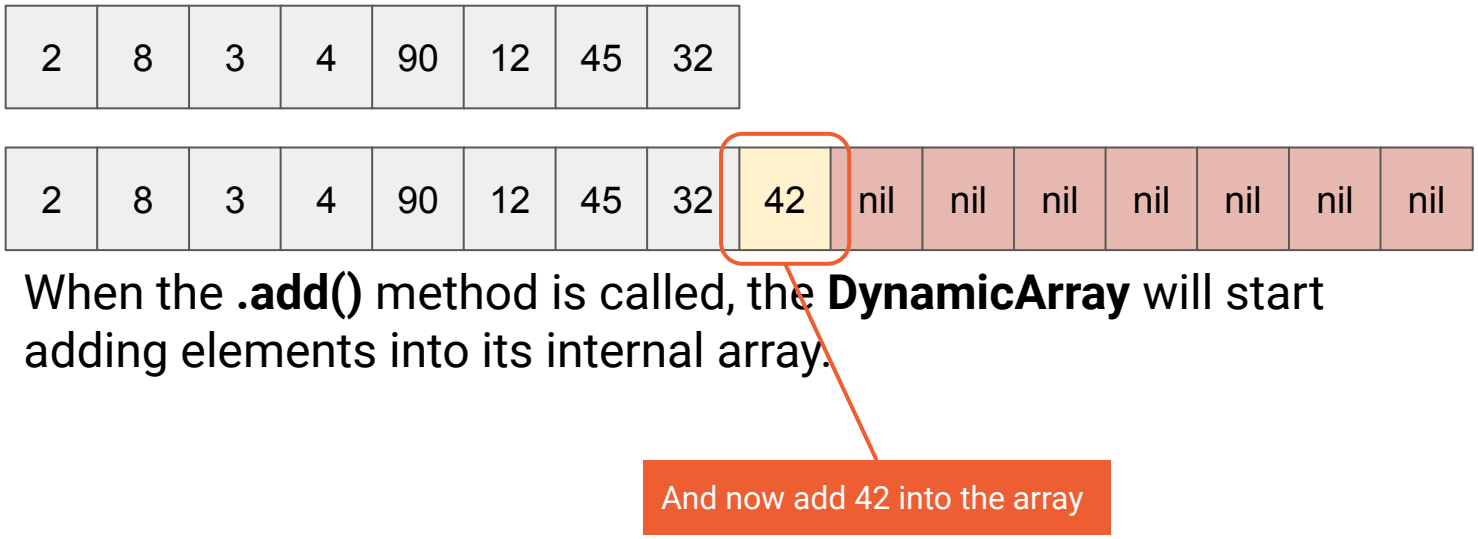
When the **.add()** method is called, the **DynamicArray** will start adding elements into its internal array.

New array with no elements!
We will copy the elements over from the previous Array to the new one.

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)

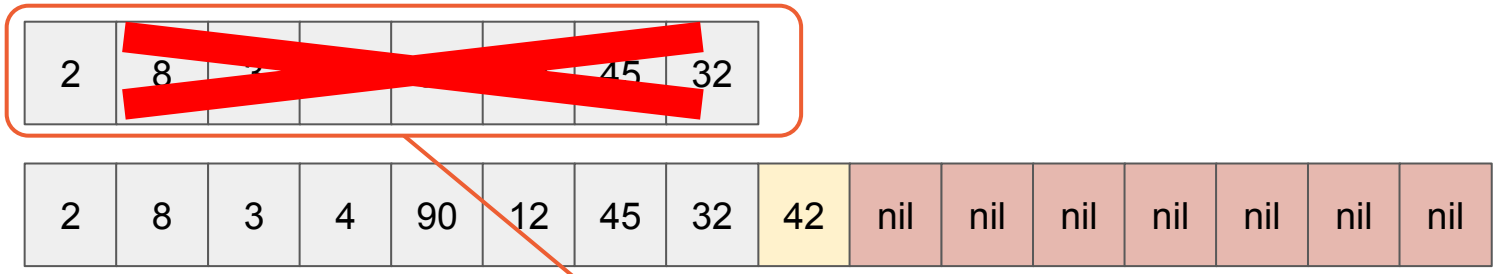


When the **.add()** method is called, the **DynamicArray** will start adding elements into its internal array.

So what is **ArrayList** doing that allows it to resize?

Let's delve into an **ArrayList** or in our version we will call it **DynamicArray**.

We're given a starting array where all elements are **null**. (nil)



When the **.add()** method is called, the **DynamicArray** will start adding elements into its internal array.

After we have our new array, copied the elements across and add the new element, we lose the reference to the old array and it is collected by the garbage collector.

Let’s make our own!

LinkedList

You can be forgiven for thinking that a **LinkedList** is another name for **ArrayList**. If we were to look at the methods between the two classes it would like we are seeing a duplicate.

Methods

Modifier and Type	Method and Description
boolean	`add(E e)` Appends the specified element to the end of this list.
void	`add(int index, E element)` Inserts the specified element at the specified position in this list.
boolean	`addAll(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the
boolean	`addAll(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list, starting at the sp
void	`addFirst(E e)` Inserts the specified element at the beginning of this list.
void	`addLast(E e)` Appends the specified element to the end of this list.
void	`clear()` Removes all of the elements from this list.
Object	`clone()` Returns a shallow copy of this `LinkedList`.
boolean	`contains(Object o)` Returns `true` if this list contains the specified element.
Iterator<E>	`descendingIterator()` Returns an iterator over the elements in this deque in reverse sequential order.
E	`element()` Retrieves, but does not remove, the head (first element) of this list.
E	`get(int index)` Returns the element at the specified position in this list.

Modifier and Type	Method and Description
boolean	`add(E e)` Appends the specified element to the end of this list.
void	`add(int index, E element)` Inserts the specified element at the specified position in this list.
boolean	`addAll(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of t
boolean	`addAll(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list, star
void	`clear()` Removes all of the elements from this list.
Object	`clone()` Returns a shallow copy of this `ArrayList` instance.
boolean	`contains(Object o)` Returns `true` if this list contains the specified element.
void	`ensureCapacity(int minCapacity)` Increases the capacity of this `ArrayList` instance, if necessary, to e
E	`get(int index)` Returns the element at the specified position in this list.
int	`indexOf(Object o)` Returns the index of the first occurrence of the specified element in t
boolean	`isEmpty()` Returns `true` if this list contains no elements.
Iterator<E>	`iterator()` Returns an iterator over the elements in this list in proper sequence.

However they differ fundamentally in their construction and behaviour. I'll save the

LinkedList

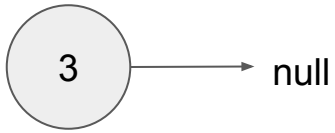
In contrast to an **ArrayList**, a **LinkedList** does not have an array containing all the elements stored. Instead, it chains elements and their values.

Each element contains a **value** and a **link** to the next element. The **link** is commonly referred to as **next** and an the elements within a **LinkedList** are commonly referred to as a **Node**.

LinkedList

In contrast to an **ArrayList**, a **LinkedList** does not have an array containing all the elements stored. Instead, it chains elements and their values.

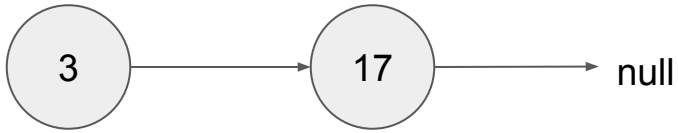
Each element contains a **value** and a **link** to the next element. The **link** is commonly referred to as **next** and an the elements within a **LinkedList** are commonly referred to as a **Node**.



LinkedList

In contrast to an **ArrayList**, a **LinkedList** does not have an array containing all the elements stored. Instead, it chains elements and their values.

Each element contains a **value** and a **link** to the next element. The **link** is commonly referred to as **next** and an the elements within a **LinkedList** are commonly referred to as a **Node**.



LinkedList

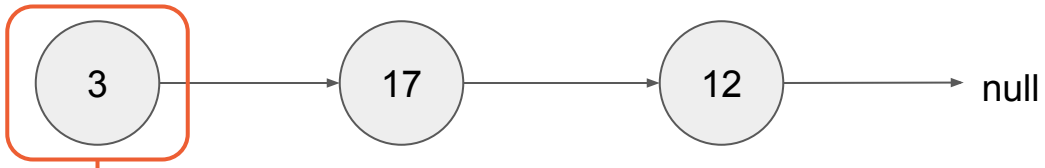
In contrast to an **ArrayList**, a **LinkedList** does not have an array containing all the elements stored. Instead, it chains elements and their values.

Each element contains a **value** and a **link** to the next element. The **link** is commonly referred to as **next** and an the elements within a **LinkedList** are commonly referred to as a **Node**.

LinkedList

In contrast to an **ArrayList**, a **LinkedList** does not have an array containing all the elements stored. Instead, it chains elements and their values.

Each element contains a **value** and a **link** to the next element. The **link** is commonly referred to as **next** and an the elements within a **LinkedList** are commonly referred to as a **Node**.



The first element of our **LinkedList** is referred to as the **Head** of the list.

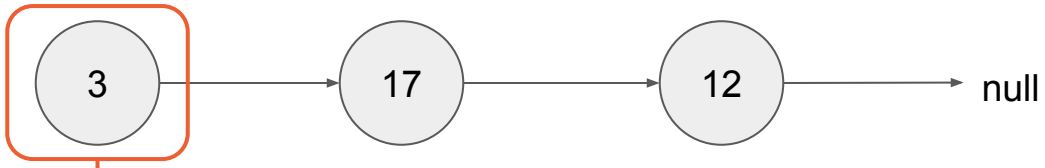
```
public class Node {
        int value;
        Node next;
}
```

Each **Node** follows a class definition similar to the following.

LinkedList

In contrast to an **ArrayList**, a **LinkedList** does not have an array containing all the elements stored. Instead, it chains elements and their values.

Each element contains a **value** and a **link** to the next element. The **link** is commonly referred to as **next** and an the elements within a **LinkedList** are commonly referred to as a **Node**.



The first element of our **LinkedList** is referred to as the **Head** of the list.

```
public class Node {
        int value;
        Node next;
}
```

Each **Node** follows a class definition similar to the following.

Wait.. Node inside a Node?

LinkedList

In contrast to an **ArrayList**, a **LinkedList** does not have an array containing all the elements stored. Instead, it chains elements and their values.

Each element contains a **value** and a **link** to the next element. The **link** is commonly referred to as **next** and an the elements within a **LinkedList** are commonly referred to as a **Node**.


```
public class Node {
        int value;
        Node next;
}
```

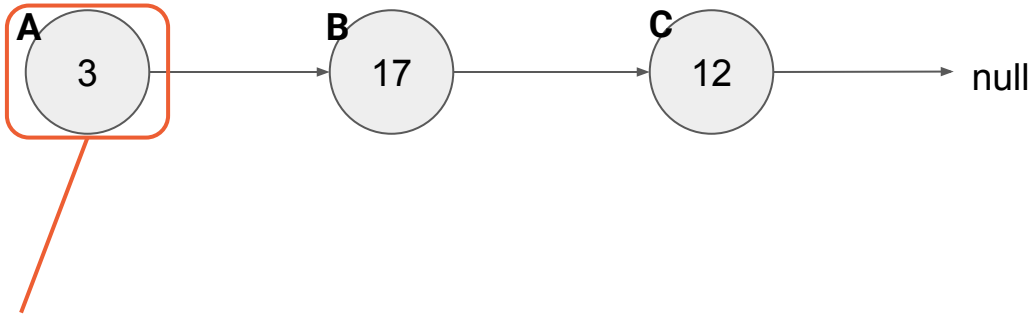
Reference types are an address to an allocation. So if the matter is “How does the Node know the size of itself?”

The answer is: “We are storing an address, so the compiler will only factor in the address size”

LinkedList

In contrast to an **ArrayList**, a **LinkedList** does not have an array containing all the elements stored. Instead, it chains elements and their values.

Each element contains a **value** and a **link** to the next element. The **link** is commonly referred to as **next** and an the elements within a **LinkedList** are commonly referred to as a **Node**.

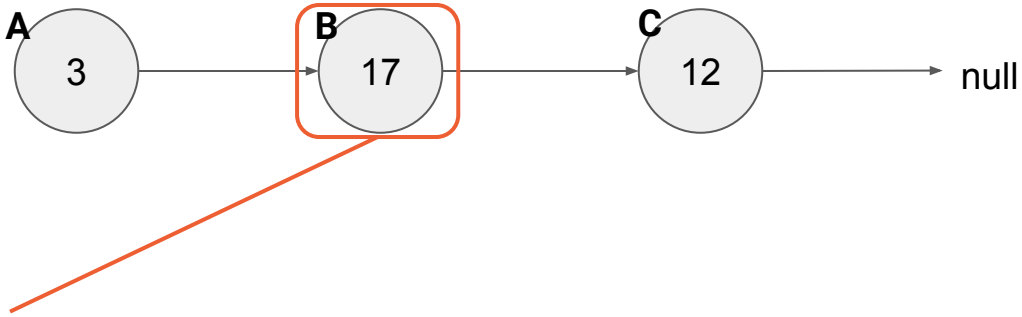



```
public class Node {
      int value = 3
      Node next = B;
}
```

LinkedList

In contrast to an **ArrayList**, a **LinkedList** does not have an array containing all the elements stored. Instead, it chains elements and their values.

Each element contains a **value** and a **link** to the next element. The **link** is commonly referred to as **next** and an the elements within a **LinkedList** are commonly referred to as a **Node**.

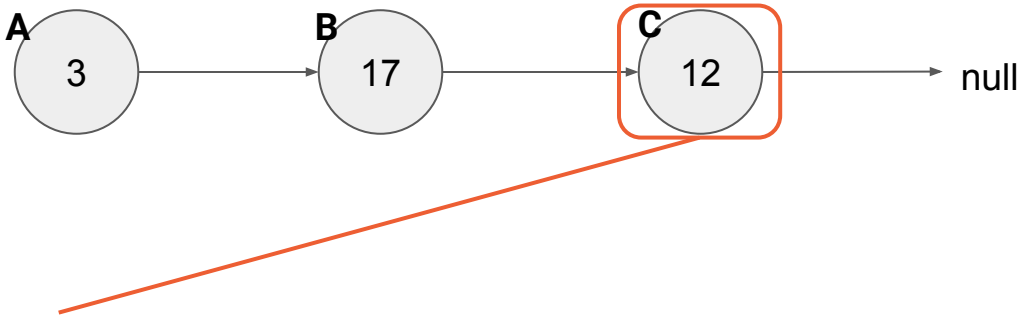



```
public class Node {
      int value = 17;
      Node next = C;
}
```

LinkedList

In contrast to an **ArrayList**, a **LinkedList** does not have an array containing all the elements stored. Instead, it chains elements and their values.

Each element contains a **value** and a **link** to the next element. The **link** is commonly referred to as **next** and an the elements within a **LinkedList** are commonly referred to as a **Node**.




```
public class Node {
        int value = 12;
        Node next = null;
}
```

Let’s make our own!

Is this it?

No!

You will always encounter a problem which doesn't fit nicely and may require extra work.

We can always combine collections within each other as they are just **another type**.

Example:

ArrayList<ArrayList<Integer>> dynamicMatrix;

Maps and Sets

We want to be able to use non-integer objects for storage. This where we bring in two different types that allow this.

Map and **Set**

Not to be confused with a **map** method. A **Map** and **Set** provides a **mapping** of an object to **location** where that element is stored.

Types that are commonly used of this variety are:

- HashMap
- TreeMap
- HashSet
- TreeSet

Maps and Sets

We want to be able to use non-integer objects for storage. This where we bring in two different types that allow this.

Map and **Set**

Not to be confused with a **map** method. A **Map** and **Set** provides a **mapping** of an object to **location** where that element is stored.

Types that are commonly used of this variety are:

- HashMap
- TreeMap
- HashSet
- TreeSet

We'll focus on HashMap and HashSet.

Map and Set

Map and **Set** have similar functionality in providing a mapping between an object and its location but they do differ in a

Let's look at the syntax:

Syntax:

```
Map<KeyType,ValueType> name = new
    Map<KeyType,ValueType>;

Set<Type> name = new Set<Type>;
```

Maps

So let’s disassemble these operations:

```
import java.util.HashMap;
public class Example {
    public static int main(String[] b) {
        HashMap<String,Integer> seats = new HashMap<String,Integer>();

        seats.put(“Bus”, 30);
        seats.put(“Car”, 5);
        seats.put(“Bike”,1);
        seats.put(“Truck”, 2);

        seats.remove(“Truck”);

        System.out.println(seats.get(“Bike”));
        System.out.println(seats.get(“Car”));

    }
}
```

Maps

So let's disassemble these operations:

```
import java.util.HashMap;
public class Example {
    public static int main(String[] b) {
        HashMap<String,Integer> seats = new HashMap<String,Integer>();

        seats.put("Bus", 30);
        seats.put("Car", 5);
        seats.put("Bike",1);
        seats.put("Truck", 2);

        seats.remove("Truck");

        System.out.println(seats.get("Bike"));
        System.out.println(seats.get("Car"));

    }
}
```

Instead of add() we have **put** instead. This is because **Key's** are **unique.**

Maps

So let's disassemble these operations:

```
import java.util.HashMap;
public class Example {
    public static int main(String[] b) {
        HashMap<String,Integer> seats = new HashMap<String,Integer>();

        seats.put("Bus", 30);
        seats.put("Car", 5);
        seats.put("Bike",1);
        seats.put("Truck", 2);

        seats.remove("Truck");

        System.out.println(seats.get("Bike"));
        System.out.println(seats.get("Car"));

    }
}
```

Remove elements based on the key.

Maps

So let's disassemble these operations:

```
import java.util.HashMap;
public class Example {
    public static int main(String[] b) {
        HashMap<String,Integer> seats = new HashMap<String,Integer>();

        seats.put("Bus", 30);
        seats.put("Car", 5);
        seats.put("Bike",1);
        seats.put("Truck", 2);

        seats.remove("Truck");

        System.out.println(seats.get("Bike"));
        System.out.println(seats.get("Car"));

    }
}
```

For Python programmers, you can think of Maps as **dictionary** type.

Checked Operations

We have a compiler to check that the types we are using are correct and we are not assigning data to the wrong variable. This is referred to as a **checked operation**.

This concept is important for collections since we have the concept of **two kinds of types**.

- Reference types
- Value types

With collection types, we only care about **Reference types** as they are the only type that can be used with the standard library collections (with **generics**).

Unchecked Operations

If you are familiar with another language such as C, Python, Javascript, Ruby
(Where either language is weakly typed or is dynamically typed).

You may have encountered the situation where you provided arbitrary types to a list or array and you as a programmer know the order.

However, this is an **assumption** and is considered an **unsafe** operation. In Java, the compiler likes to provide feedback that you are using types correctly in your code. It warns you when you have an **Unchecked Operation**.

So what does an unchecked warning look like?

We'll take a look at the syntax for List again.

Syntax:

```
List<Type> name = new ListType<Type>;
```

Why is this important for compiler?

Specifically, we omit the **Type** information from the list.

So what does an unchecked warning look like?

We'll take a look at the syntax for List again.

Syntax:

```
List      name = new ListType      ;
```

Specifically, we omit the **Type** information from the list.

ArrayList (Dynamic Array)

Using our previous example, what if we omit the type information and change what we add?

```
import java.util.ArrayList;
public class Example {
    public static int main(String[] b) {
        ArrayList<String> list = new ArrayList<String>();

        list.add(“First String!”);
        list.add(“Second String!”);
        list.add(“Woof!!”);

        list.remove(1);

        list.set(1, “Meow”);

    }
}
```

ArrayList (Dynamic Array)

Using our previous example, what if we omit the type information and change what we add?

```
import java.util.ArrayList;
public class Example {
    public static int main(String[] b) {
        ArrayList list = new ArrayList();

        list.add(“First String!”);
        list.add(new Integer(5));
        list.add(new DefinitelyNotAString());

        list.remove(1);

        list.set(1, “Meow”);

    }
}
```

By omitting the **type information** from the **ArrayList** we are able to add any object of any type to this collection.

ArrayList (Dynamic Array)

Using our previous example, what if we omit the type information and change what we add?

```
import java.util.ArrayList;
public class Example {
    public static int main(String[] b) {
        ArrayList list = new ArrayList();

        list.add("First String!");
        list.add(new Integer(5));
        list.add(new DefinitelyNotAString());

        list.remove(1);

        list.set(1, "Meow");

    }
}
```

By omitting the **type information** from the **ArrayList** we are able to add any object of any type to this collection.

What dangers are present when retrieving elements from this collection?

ArrayList (Dynamic Array)

Using our previous example, what if we omit the type information and change what we add?

```
import java.util.ArrayList;
public class Example {
    public static int main(String[] b) {
        ArrayList list = new ArrayList();

        list.add("First String!");
        list.add(new Integer(5));
        list.add(new DefinitelyNotAString());

        list.remove(1);

        list.set(1, "Meow");

    }
}
```

By omitting the **type information** from the **ArrayList** we are able to add any object of any type to this collection.

What dangers are present when retrieving elements from this collection?

- We may use the wrong object (may assume it is a string when it is an integer)
- We have no idea what is stored there and therefore functionally useless.

See you next time!