INFO1113 / COMP9003
Object-Oriented Programming

Lecture 4

THE UNIVERSITY OF
SYDNEY

Contents developed by Tyson Thomas

Acknowledgement of Country

I would like to acknowledge the Traditional Owners of Australia and recognise their continuing connection to land, water and culture. I am currently on the land of the Gadigal people of the Eora nation and pay my respects to their Elders, past, present and emerging.

I further acknowledge the Traditional Owners of the country on which you are on and pay respects to their Elders, past, present and future.

Copyright Warning

Topics: Part A

- **Binary Input/Output**
- **Garbage Collector**

Binary IO

Last week we saw how we could read data from text files. We will start this lecture by reading a binary file and how we can store data in a non-readable format.

Applications

Why would we want to do this?

There are going to be applications where the data itself will not be presented in a textual representation.

- Interacting with devices and peripherals (gamepads, midi devices)
- Modifying executables
- Encoding images
- Writing your own file format for your programs
- Writing software for automotive applications
- Video and audio streams
- Networking application

Binary I/O

.class files are not human readable, just like image formats and executables but how does the **JVM** understand and interpret this file?

Simply, there is some method of interpreting binary files and being able to read them.

We can use **DataInputStream, DataOutputStream** and use the available methods such as **readChar, writeInt** to read or write binary file.

Binary I/O: Writing

```
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;


public class BinaryWriter {

    public static void main(String[] args) {

        try {

            FileOutputStream f =  new FileOutputStream("newfile.bin");

            DataOutputStream output = new DataOutputStream(f);

            output.writeInt(50);

            output.close();

        } catch (FileNotFoundException e) {

            e.printStackTrace();

        } catch (IOException e){

            e.printStackTrace();

        }

    }

}
```

Binary I/O: Reading

```java
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;


public class BinaryReader{

    public static void main(String[] args) {

        try {

            FileInputStream f =  new FileInputStream("newfile.bin");

            DataInputStream input = new DataInputStream(f);

            int n = input.readInt();

            System.out.println(n);

        } catch (FileNotFoundException e) {

              e.printStackTrace();

        } catch (IOException e){

              e.printStackTrace();

        }

    }

}
```

Demonstration

Garbage Collector

Java employs the use of a **garbage collector** to free memory. Any memory that is allocated using the **new** keyword will be kept on the heap.

The garbage collector will subsequently free any allocation that no longer has a reference during execution.

The garbage collector will **stop-the-world** and act on allocations without a reference.

All I have known is garbage collecting.. what was it like before?

What does manual look like?

```
Person createPerson() {
  Person p = new Person();
  //things
  return p;
}

//<other bits of code>

void deletePerson(Person p) {
  delete p;
}
```

What does manual look like?

```
Person createPerson() {
  Person p = new Person();
  //things
  return p;
}

//<other bits of code>

void deletePerson(Person p) {
  delete p;
}
```

← We have allocated a new Person object

What does manual look like?

```
Person createPerson() {
  Person p = new Person();
  //things
  return p;
}

//<other bits of code>

void deletePerson(Person p) {
  delete p;
}
```

← We have returned p to the calling method

What does manual look like?

```
Person createPerson() {
  Person p = new Person();
  //things
  return p;
}

//<other bits of code>

void deletePerson(Person p) {
  delete p;
}
```

Once the allocation has been used and the programmer wants it deleted it is sent to the **deletePerson** method

What does manual look like?

```
Person createPerson() {
  Person p = new Person();
  //things
  return p;
}

//<other bits of code>

void deletePerson(Person p) {
  delete p;
}
```

← At this point allocation is deleted

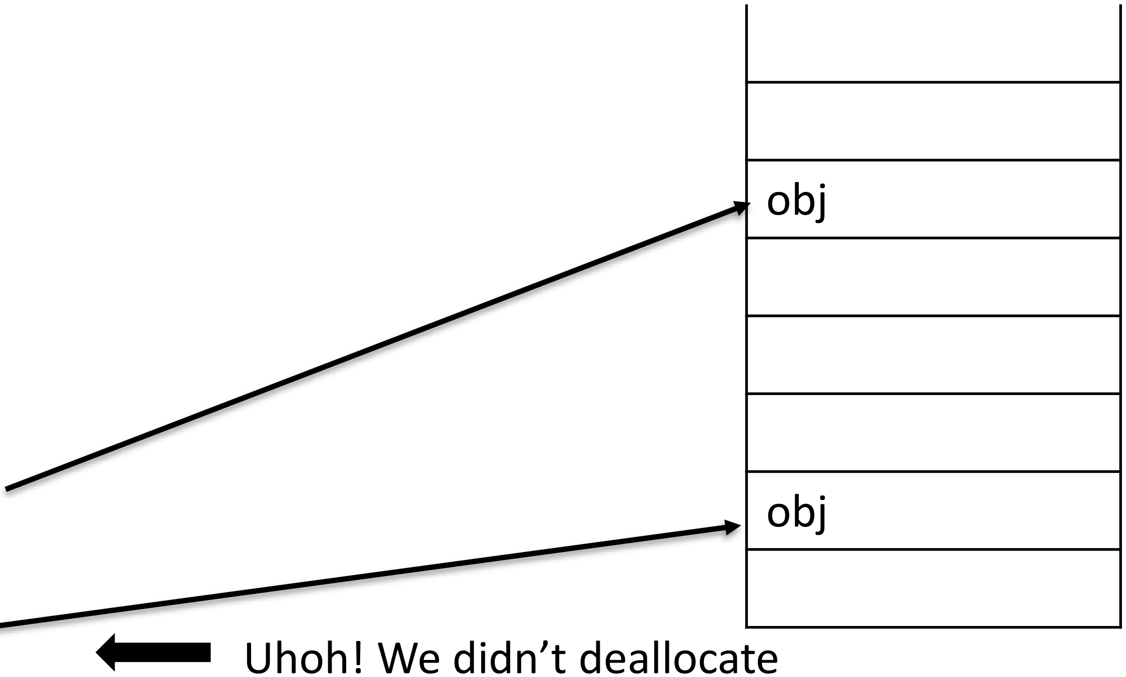
Scenario 1: Forgot to free

When we forget to deallocate:

```
Person createPerson() {
  Person p = new Person();
  return p;
}

//<other bits of code>

void main(String[] args) {
  Person p = createPerson();
  someWork(p);
  p = createPerson();
}
```



Uhoh! We didn't deallocate the previous one!

How does the garbage collector handle this?

Scenario 1: Forgot to free

When we forget to deallocate:

```
Person createPerson() {
  Person p = new Person();
  return p;
}

//<other bits of code>

void main(String[] args) {
  Person p = createPerson();
  someWork(p);
  p = createPerson();
}
```

← Since the original allocation has no references to it, when the GC passes it, it will be marked for deletion.

Scenario 2: Oops! I thought it was still allocated

```
Person createPerson() {
  Person p = new Person();
  return p;
}

//<other bits of code>

void main(String[] args) {
  Person p = createPerson();
  Person c = p;
  delete p;
  someWork(c);
}
```

←

Uhoh! If the memory has been freed then potentially we could be using memory that we don't own anymore

Scenario 2: Oops! I thought it was still allocated

```
Person createPerson() {
  Person p = new Person();
  return p;
}

//<other bits of code>

void main(String[] args) {
  Person p = createPerson();
  Person c = p;
  ~~delete p;~~
  someWork(c);
}
```

We don’t do this anymore

Simply: There is no manual deallocation

Why?

Shouldn’t you just be more careful?

It’s about simplifying writing software for everyone. It allows you as a programmer to be able to write software without needing to worry about memory management.

We can minimise errors and make programming **safer** through this method.

Garbage Collector

How does the garbage collector clean up?

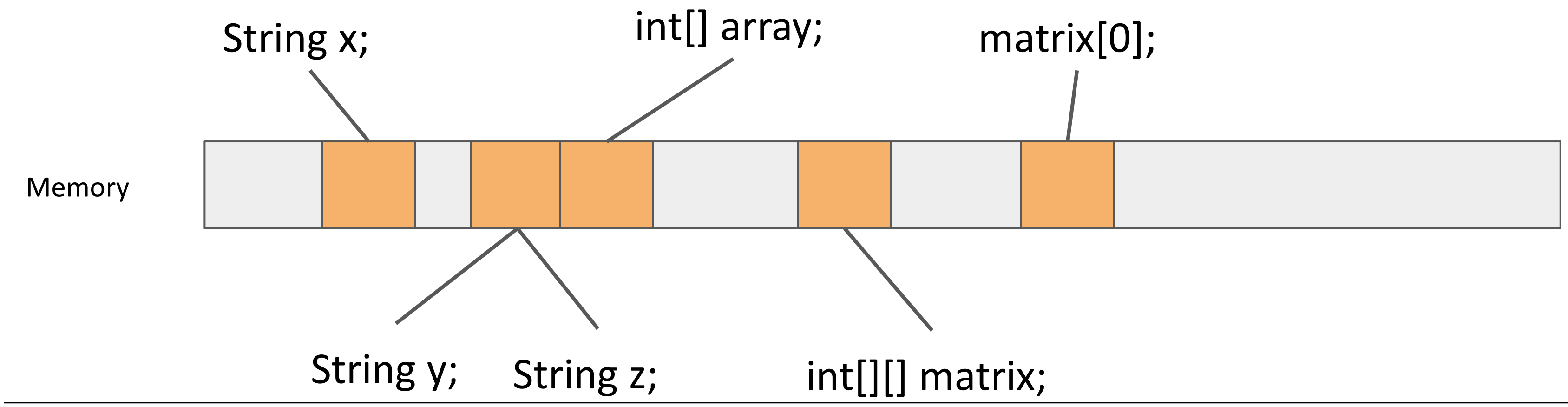
When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

Garbage Collector

How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

Let's see how the following works:

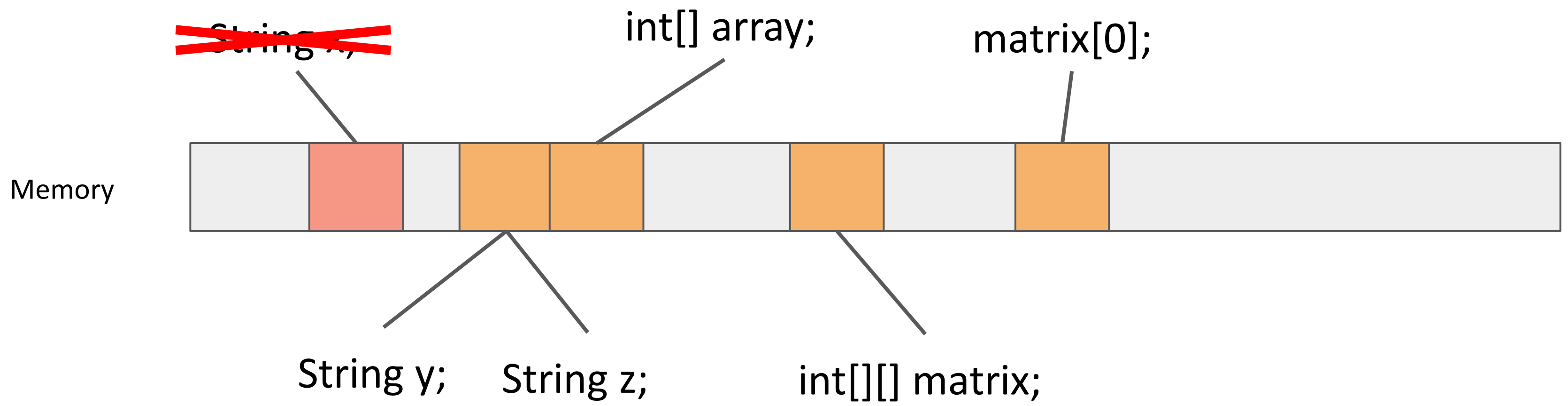


Garbage Collector

How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

Let's see how the following works:

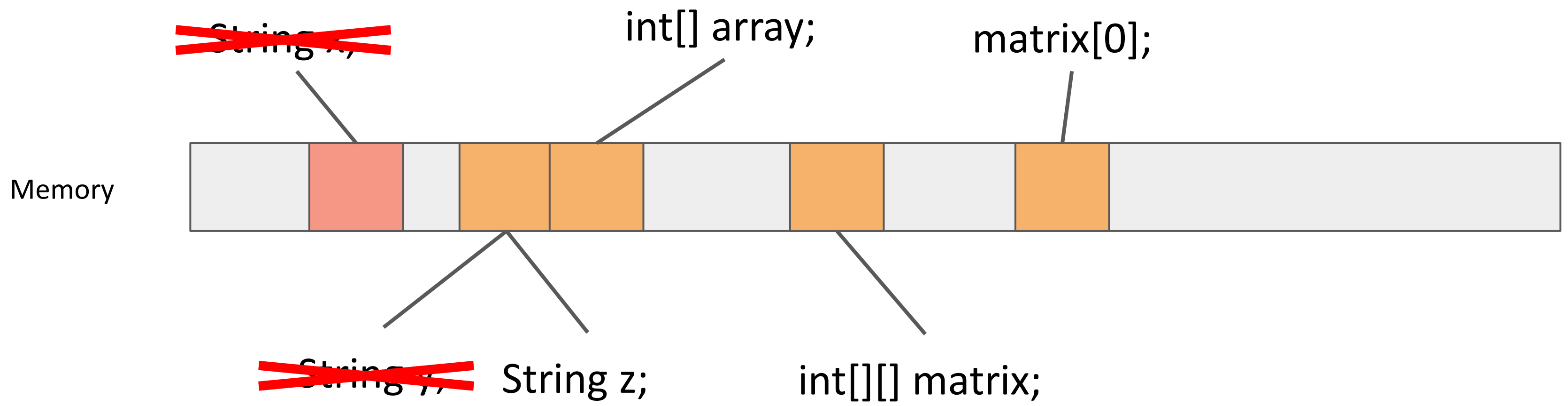


Garbage Collector

How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

Let's see how the following works:

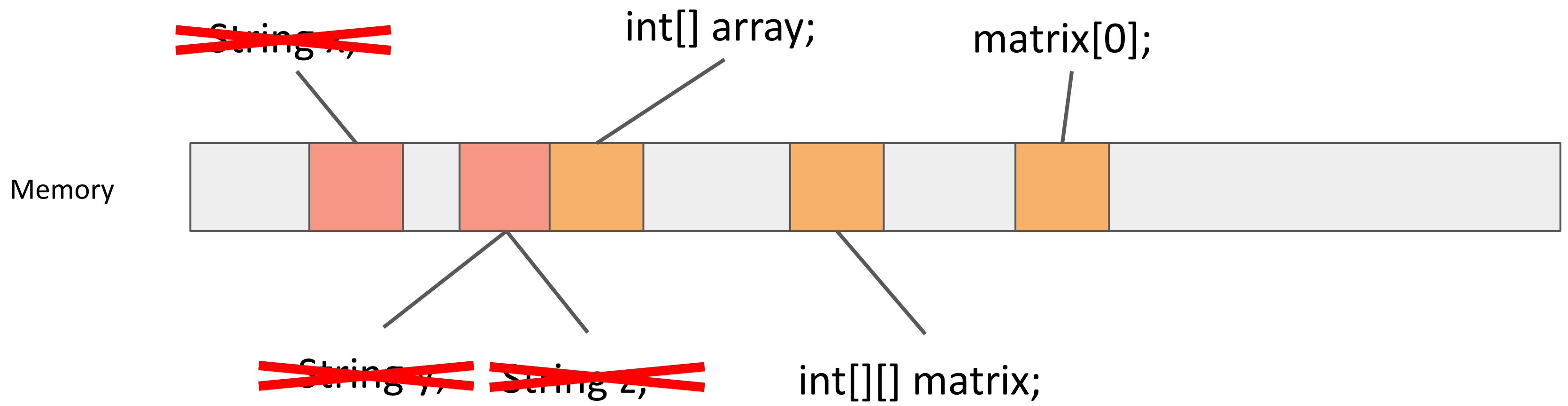


Garbage Collector

How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

Let's see how the following works:

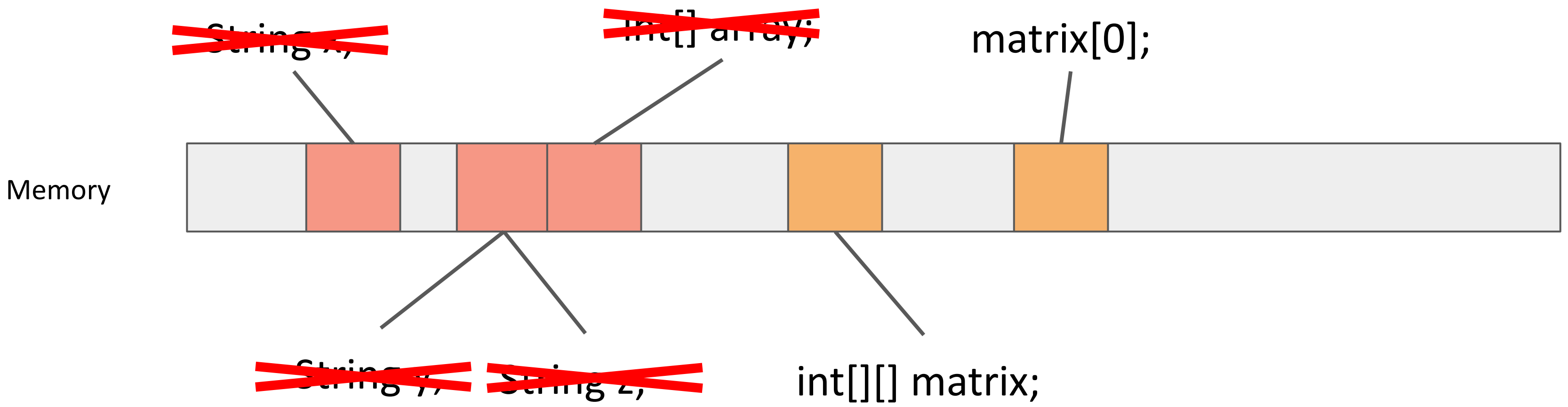


Garbage Collector

How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

Let's see how the following works:

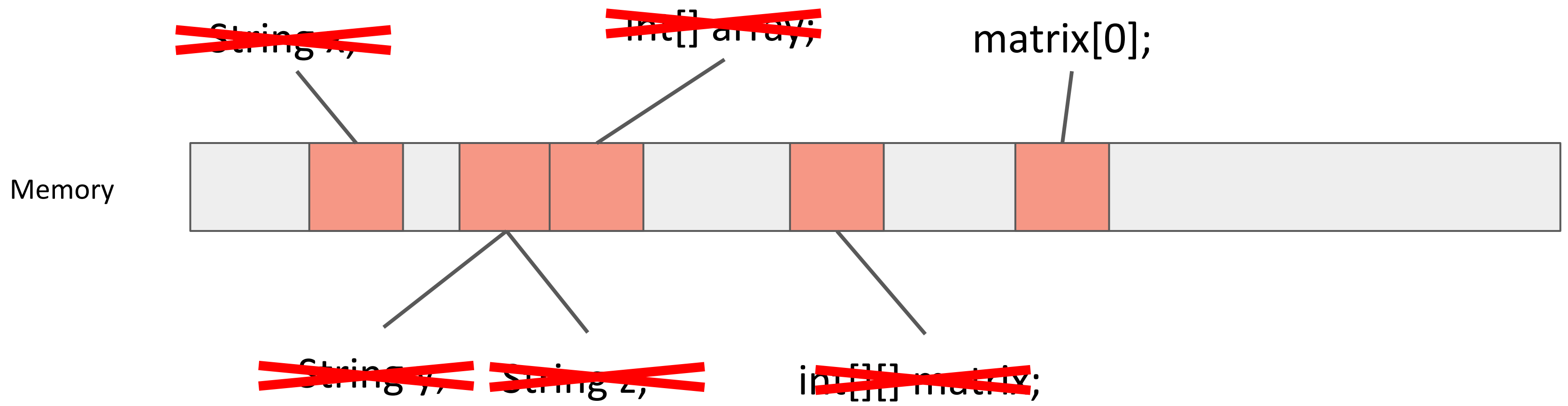


Garbage Collector

How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

Let's see how the following works:

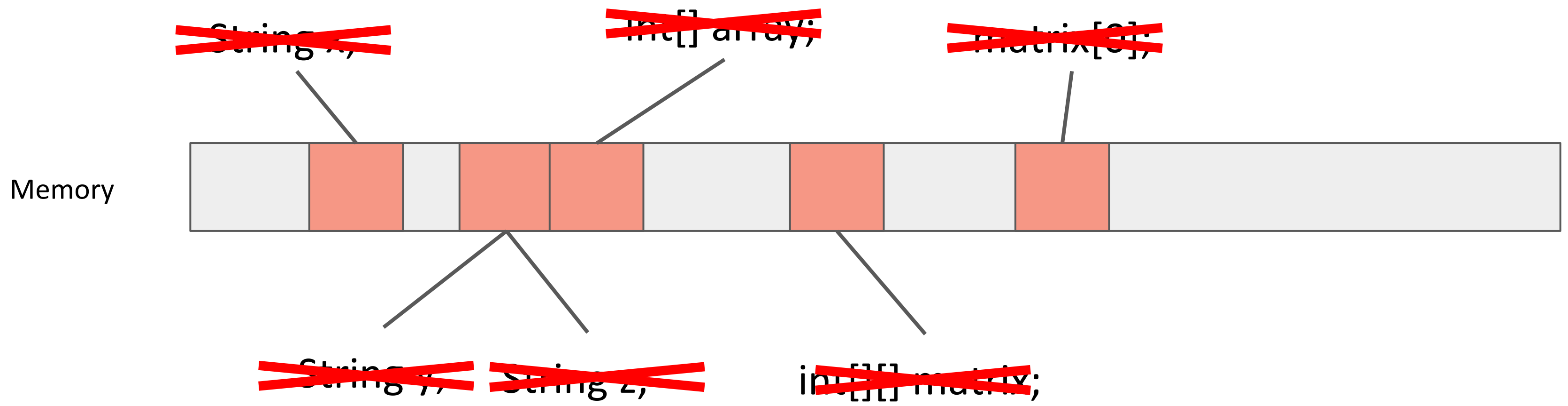


Garbage Collector

How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

Let’s see how the following works:



Let’s take a break!

Topics: Part B

- **ADTs**
- **Lists**
- **Maps and Sets**
- **Checked and Unchecked Operations**

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 allow us to use 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

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.

List

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

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

List

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

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

Okay… slight problem here, Arrays are fixed length.

Let’s try to 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<<u>Type</u>> <u>name</u> = new *ListType*<<u>Type</u>>;

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:

Variable name

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.

Concrete type, allows you to specify a certain implementation of **list**.

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 our code

Data Structures

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)

nil	nil	nil	nil	nil	nil	nil	nil

2
8
3
4
90
12
45
32

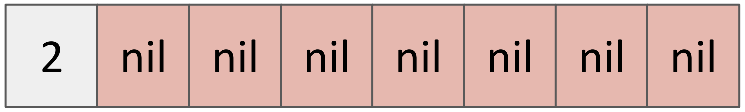
When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

Data Structures

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.

Data Structures

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	nil	nil	nil	nil	nil	nil

2
8
3
4
90
12
45
32

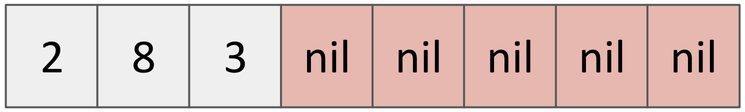
When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

Data Structures

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

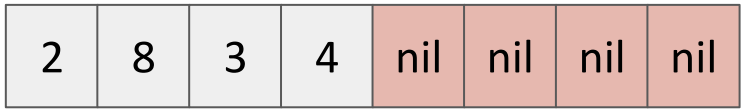
When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

Data Structures

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.

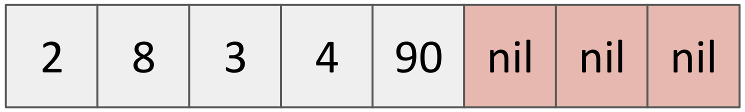
Data Structures

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



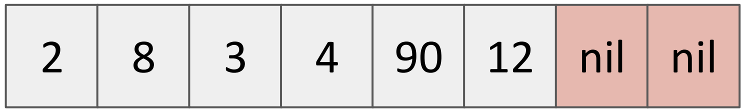
When the **.add()** method is called the **DynamicArray** will start adding elements into the Array.

Data Structures

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.

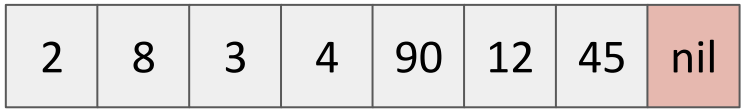
Data Structures

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.

Data Structures

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.

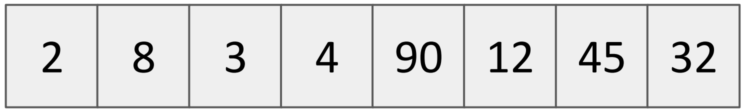
2
8
3
4
90
12
45
32

Data Structures

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.

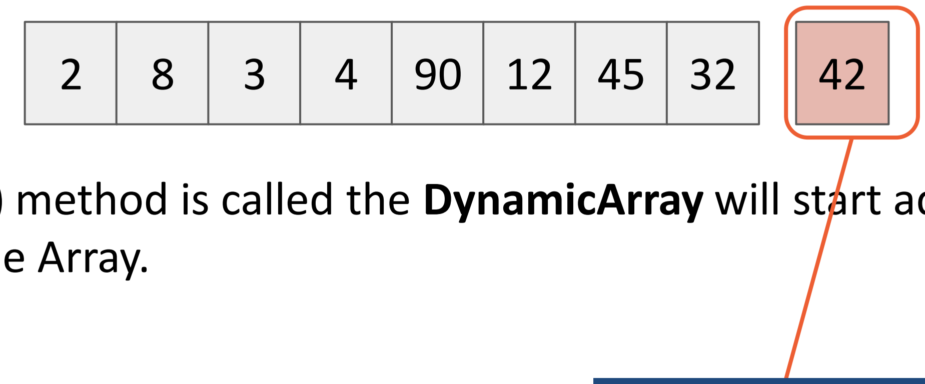
Hang on! We just ran into the same problem with an Array!
This is where the **resize occurs**.

Data Structures

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.

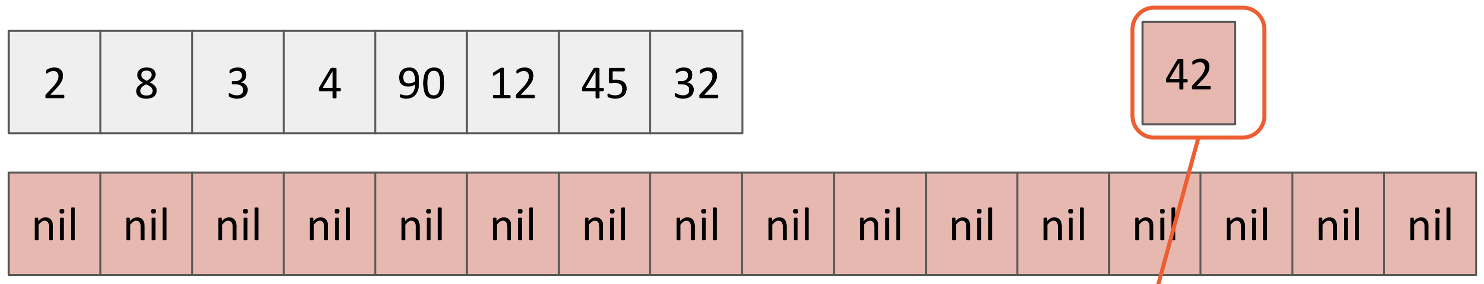
We will create a new array!

Data Structures

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
42
nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil

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

Can’t simply override an element.

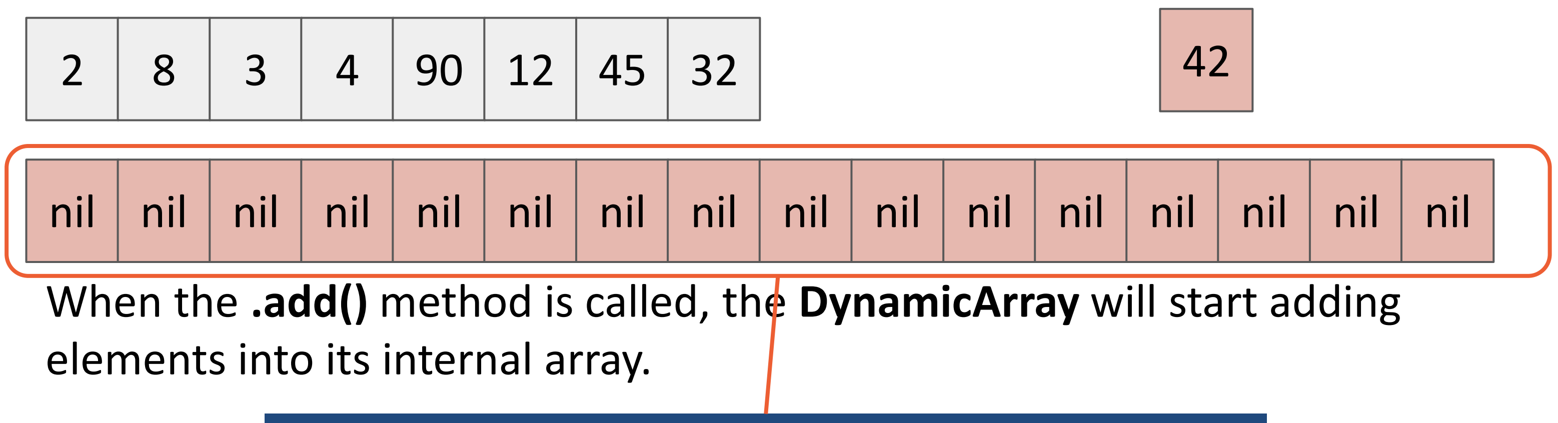
We will create a new array! Double the size of the old one!

Data Structures

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.

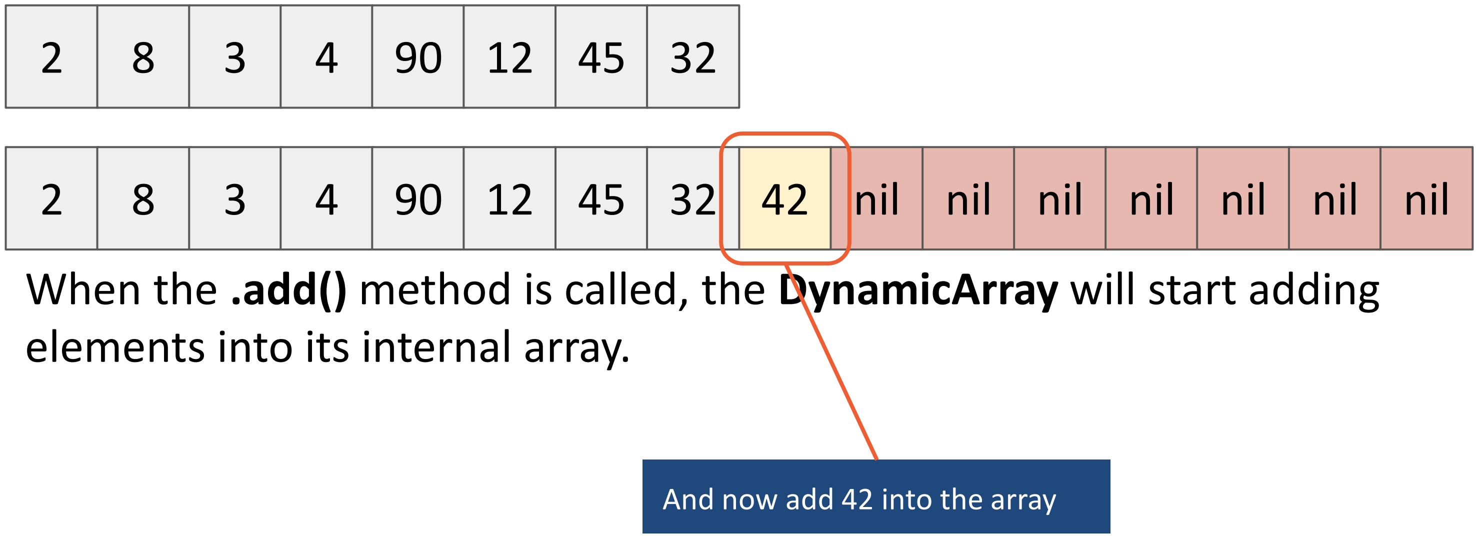
New array with no elements!
We will copy the elements over from the previous Array to the new one.

Data Structures

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)



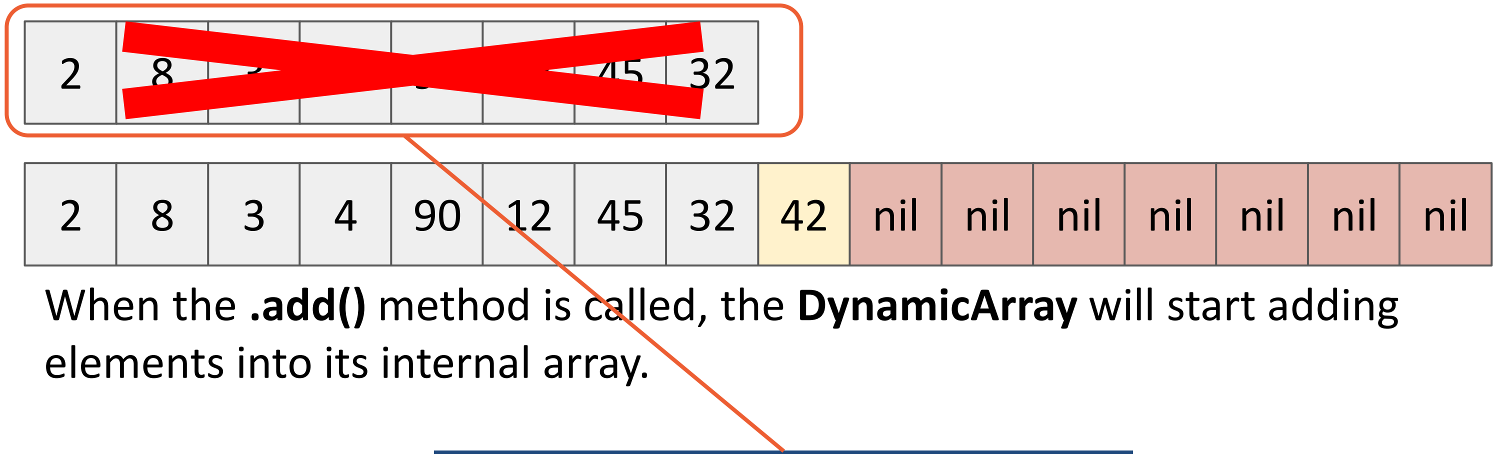
When the **.add()** method is called, the **DynamicArray** will start adding elements into its internal array.

Data Structures

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)



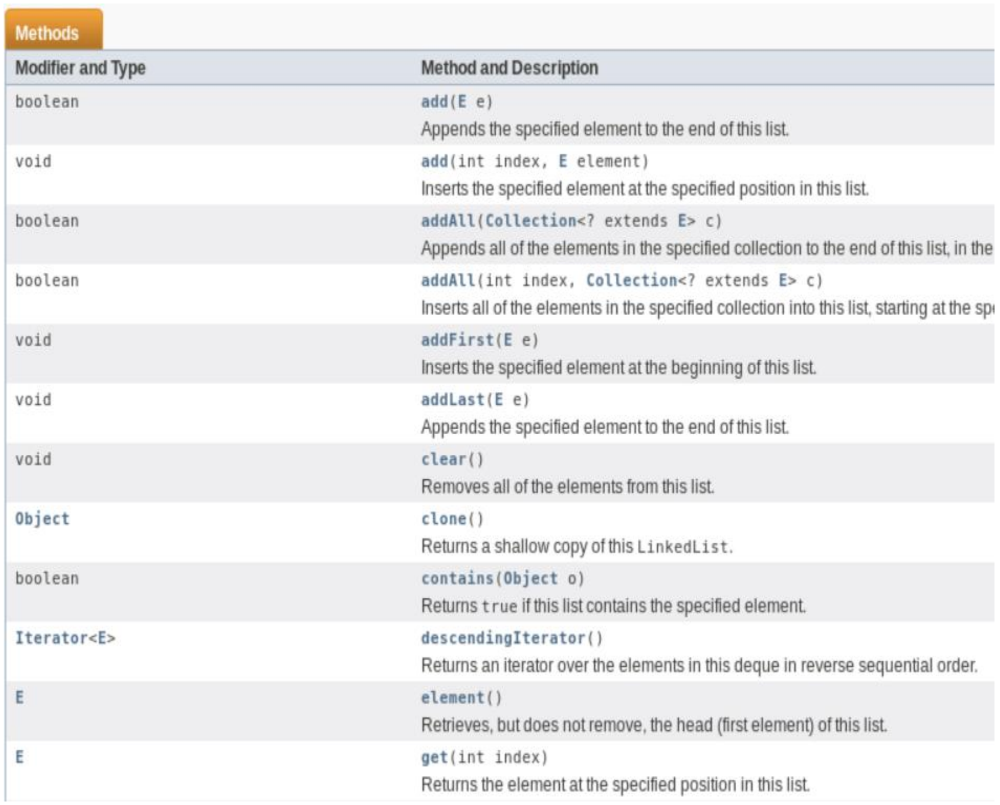
When the **.add()** method is called, the **DynamicArray** will start adding elements into its internal array.

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!

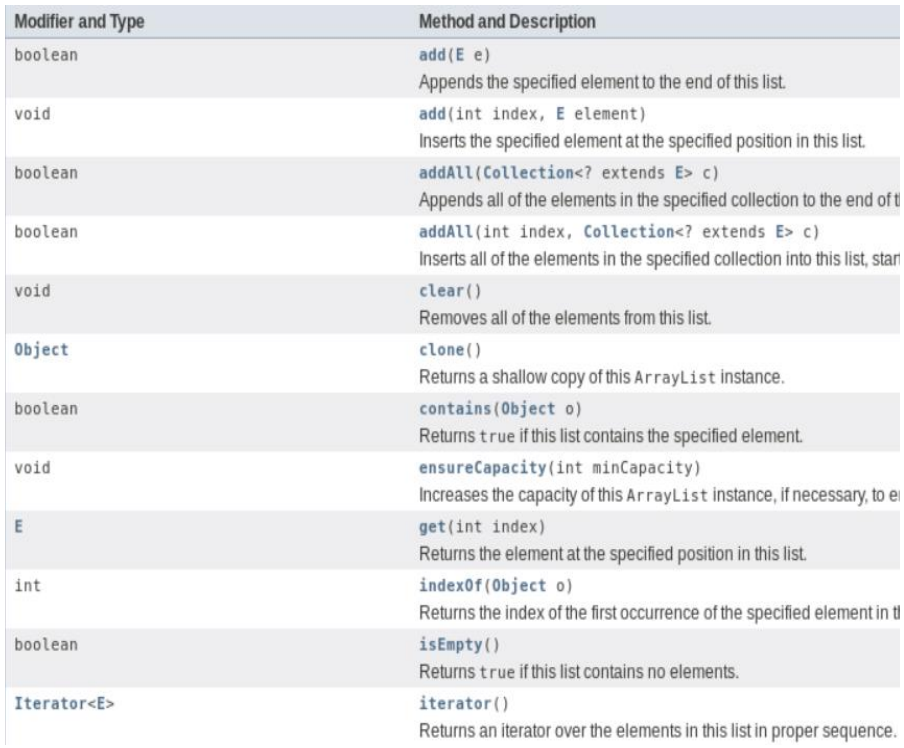
LinkedList

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

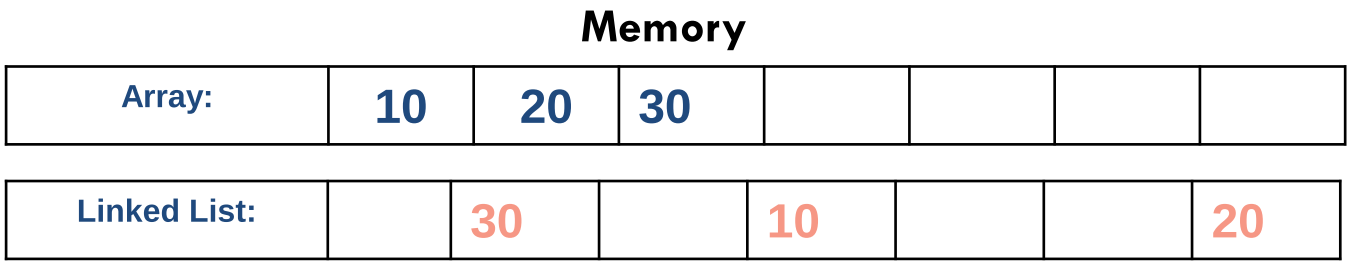
Modifier and Type	Method and Description
`boolean`	`add(E e)` Appends the specified element to the end of this list.
`void`	`add(int index, E element)` Inserts the specified element at the specified position in this list.
`boolean`	`addAll(Collection<? extends E> c)` Appends all of the elements in the 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.

LinkedList



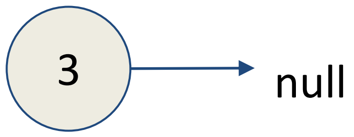
In contrast to an **ArrayList**, a **LinkedList** does not have an array containing all the elements stored. Instead, it chains elements and their values.

Each element contains a **value** and a **link** to the next element.
The **link** is commonly referred to as **next** and 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.

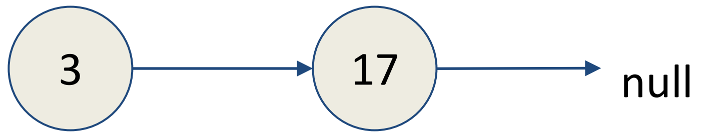
Each element contains a **value** and a **link** to the next element.
The **link** is commonly referred to as **next** and the elements within a **LinkedList** are commonly referred to as a **Node**.



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.

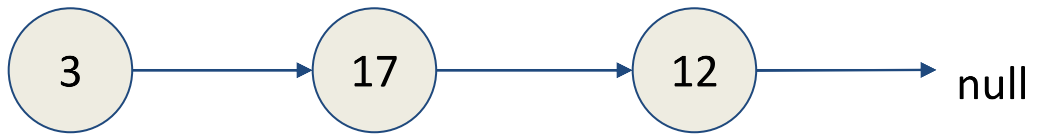
Each element contains a **value** and a **link** to the next element.
The **link** is commonly referred to as **next** and the elements within a **LinkedList** are commonly referred to as a **Node**.



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.

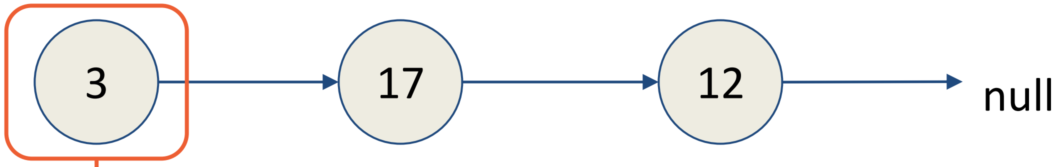
Each element contains a **value** and a **link** to the next element.
The **link** is commonly referred to as **next** and the elements within a **LinkedList** are commonly referred to as a **Node**.



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 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;
}
```

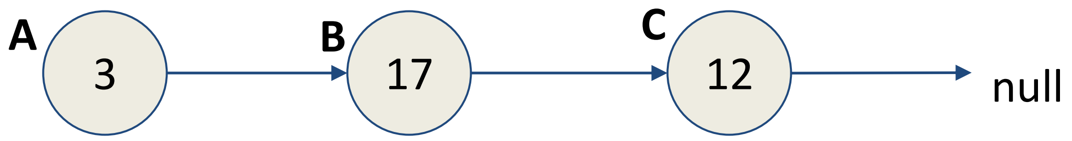
Each **Node** follows a class definition similar to the following.

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 the elements within a **LinkedList** are commonly referred to as a **Node**.



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

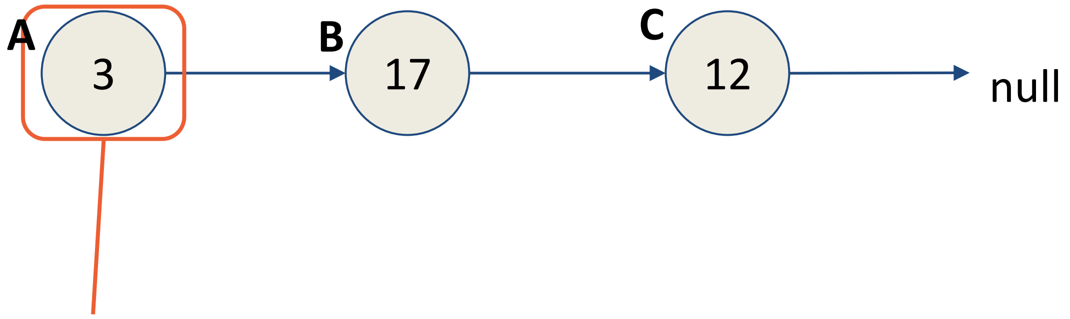
Reference types are an address to an allocation.
So if the matter is "How does the Node know the size of itself?"

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 the elements within a **LinkedList** are commonly referred to as a **Node**.

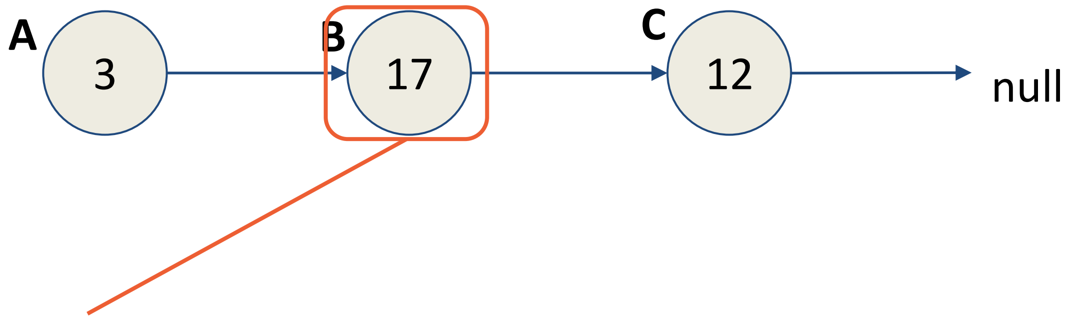


```
public class Node {
        int value = 3;
        Node next = B;
}
```

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 the elements within a **LinkedList** are commonly referred to as a **Node**.

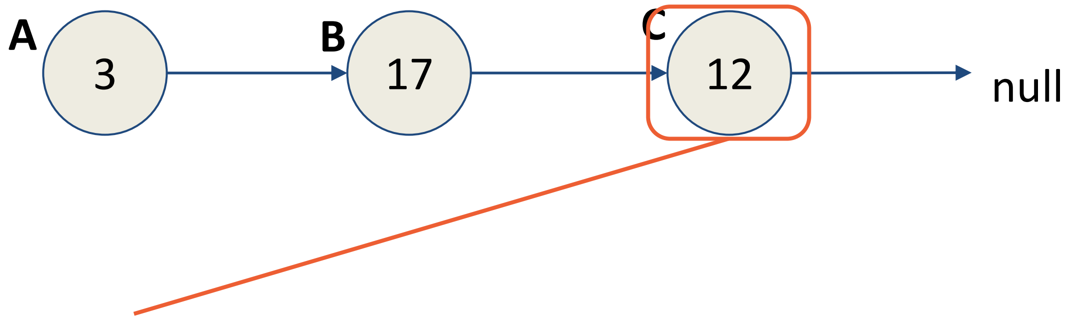


```
public class Node {
        int value = 17;
        Node next = C;
}
```

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 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 is 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 is 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

Syntax:

Map<<u>KeyType,ValueType</u>> <u>name</u> = new MapType<<u>KeyType,ValueType</u>>;

Set<<u>Type</u>> <u>name</u> = new SetType<<u>Type</u>>;

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**.

ArrayList

So what does an unchecked warning look like?

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

Syntax:

List<<u>Type</u>> <u>name</u> = new *ListType*<<u>Type</u>>;

Why is this important for compiler?

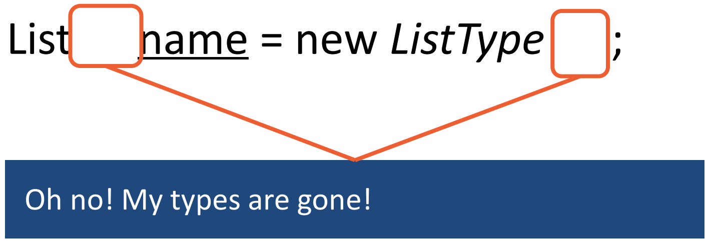
Specifically, we omit the **Type** information from the list.

ArrayList

So what does an unchecked warning look like?

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

Syntax:


List $\boxed{}$ <u>name</u> = new *ListType* $\boxed{}$;

Oh no! My types are gone!

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

ArrayList

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

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 Double(10.6));

    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

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 Double(10.6));

      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.

Demonstration: Unchecked Operation

See you next time!

THE UNIVERSITY OF SYDNEY