

# Learning Objectives: Array Basics

---

- Create an array using both the *initializer list* and *new* methods
- Access and modify array elements
- Iterate through arrays using both a regular `for` loop and an *enhanced for* loop
- Determine array output

# Creating an Array

---

## What Is an Array?

An **array** is a data structure that stores a collection of data such as ints, doubles, Strings, etc. This data is often referred to as the array's **elements**. Being able to store elements into an array helps reduce the amount of time needed to declare and initialize variables. For example, if you wanted to store the names of all family members in your household, you would typically have to declare and initialize String variables and values for each family member. Copy the code below into the text editor on the left and then click the TRY IT button to see the output. You can also click on the [++Code Visualizer++](#) link underneath to see how the program runs behind the scenes.

```
String a = "Alan";
String b = "Bob";
String c = "Carol";
String d = "David";
String e = "Ellen";

System.out.println(a);
```

challenge

### What happens if you:

- Change the a in `System.out.println(a)` to b, c, d, or e?

## Array Creation

To avoid the repetitive task of declaring and initializing multiple variables, you can declare an array and directly assign values or elements into that array like below. This technique is referred to as the **initializer list** method.

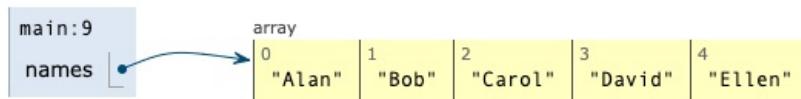
```
String[] names = {"Alan", "Bob", "Carol", "David", "Ellen"};
```

## “Initializer List” Method Syntax:

- Specify the data type that the array will store (i.e. String) followed by empty brackets [].
- Declare the variable name for the array (i.e. names) followed by the assignment symbol =.
- Elements assigned to the array are separated by commas , and enclosed within curly braces {}.

### ▼ Additional information

If you used the Code Visualizer, you’ll notice that the array variable names refers to all of the elements as a collection. An array is considered to be an **object** that bundles all of the data that it holds.



Note that the first array slot, or **index**, is always 0 so Alan is located at index 0 instead of 1.

Alternatively, you can create an array using the **new** method in which you will need to declare and specify the array variable and length before you can assign elements to the array.

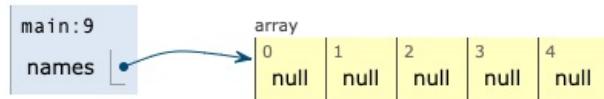
```
String[] names = new String[5];
```

## “New” Method Syntax

- Specify the data type that the array will store (i.e. String) followed by empty brackets [].
- Declare the variable name for the array (i.e. names) followed by the assignment symbol =.
- Declare the keyword **new** followed by the data type (i.e. String) and number of elements in brackets (i.e. [5]).

### ▼ Additional information

If you used the Code Visualizer, you’ll notice that the array variable names refers to all of the elements as a collection. null appears in all of the array slots because no elements have been assigned to them yet.



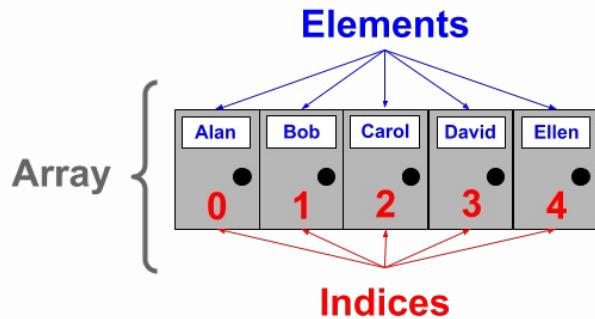
Note that the first

array slot, or **index**, is always 0.

## Array Details

Both the **initializer list** and **new** methods mentioned above will store five elements in the array called `names`. However, the `new` method automatically initializes each element to a default value of `null` while the *initializer list* method does not. Note that array slots are formally called **indices** and each **index** can carry just one type of data. For example, storing an `int` of 5 into a `String` array will result in an error.

P.O. Boxes at the postal office is symbolically similar to arrays. Each row of P.O. Boxes is like an array, except each box can only store *one* item (element) *and* each item within that row must be of the same *type* (i.e. Strings). The position at which each box is located is its index.



.guides/img/ArrayElementsIndices

# Oversize and Perfect Size Arrays

## Perfect size arrays

A **perfect size array** is an array where the number of elements is equal to the memory allocated. (These are almost certainly the variety of array that you're already familiar with.) For example, the maximum daily temperature for a week could be stored in an array with 7 integer values.

Perfect size arrays can be created using an initializer, which sets both the array's length and initializes each element in the array to given values:

```
int[] highTempsF = {93, 80, 62, 75, 74, 89, 97};
```

Perfect size arrays of objects may contain **null** values.

## When to use perfect size arrays

Perfect size arrays are especially useful when the size of the array is fixed by the context of the program. Storing the names of the days of the week in a perfect size array makes sense because the number of days in a week will not change. Compare this to an array that contains the list of cities where a company has locations – companies may add or remove locations over time, so this is not a great candidate for a perfect size array.

## Methods with perfect size array parameters

Perfect size arrays are passed to methods using only the array's reference (its location in *heap* memory). The number of elements and data are accessed from that array reference. Consider this method to store the given value at each element of the array:

```
public static void fill(int[] arrayReference, int value) {
    int index;
    for (int index = 0; index < arrayReference.length; ++index) {
        arrayReference[index] = value;
    }
}
```

When an array **reference** is passed to a method, the method can update the contents of the original array AND those updates will persist after the method has completed, without needing to return a value (note that the

return type of `fill()` is **void**).

## Methods returning perfect size arrays

Methods can also *return* perfect size array references. Since methods can only return one value, a method returning an array MUST return a perfect size array!

## Oversize arrays

An **oversize array** is an array where the number of elements used at any given time in the program is *less than or equal to* the memory allocated for the array (that is, its length). Since the number of elements (the **size**) of an oversize array can be less than the array's length, we use a separate integer variable to keep track of how many elements are currently used.

This is an example of an **empty** oversize array:

```
int[] programScores = new int[10];
int programScoresSize = 0;
```

Since you haven't completed any programming assignments yet this semester, the array only contains its default values. As the semester progresses, you may add more data to this array, and increase the corresponding size variable accordingly.

## Oversize array protocols

There is nothing inherent to Java that enforces the rules of oversize arrays; these are a set of protocols that programmers agree to follow to make oversize array usage predictable:

1. Oversize arrays consist of an **initialized array reference** and an **integer size variable**.
2. Oversize arrays must be **compact**; that is, initialized values in an oversize array are present from index 0 to index size-1 with no gaps.

This means that any data outside of the index range [0 - (size-1)] is considered "junk data".

## Oversize arrays vs perfect size arrays

Let's compare the usage of these two standards of array maintenance. In each of the following questions, select the variable type that *best* matches the stated application.

# Accessing an Array

---

## Array Access

To access and print array elements, you need to know their position. The position at which an element is stored is called its **index**. For example, `numbers[0]` refers to the first element in the array called `numbers`. Array indices always start at `0` and increment by 1 with each element that comes next. Due to this, `numbers[4]` refers to the *fifth* element in the array, *not* the fourth.

```
int[] numbers = {1, 2, 3, 4, 5};  
  
System.out.println(numbers[0]);
```

challenge

### What happens if you:

- Change `numbers[0]` in the code above to `numbers[2]`?
- Change `numbers[0]` in the code above to `numbers[3]`?
- Change `numbers[0]` in the code above to `numbers`?

important

### IMPORTANT

You may have noticed that printing the `numbers` array without specifying an index resulted in an output that starts with `[I@...`. This occurs because printing an array actually prints its memory location, not its elements. You'll learn how to print all elements in an array without having to specify all of their indices on a later page.

## Default “New” Elements

When using the `new` method to create an array, there are default values that populate as elements inside of the array, depending on the array type. For example, `String[] words = new String[5]` will result in *five* `null` elements and `int[] numbers = new int[5]` will populate *five* elements of `0`s within the array. Below is a table showing the default values of different array types when the “new” method is used.

Data Type	Default Value
String	null
int	0
double	0.0
boolean	false

```
double[] decimals = new double[2];
boolean[] bools = new boolean[2];

System.out.println(decimals[0]);
System.out.println(bools[0]);
```

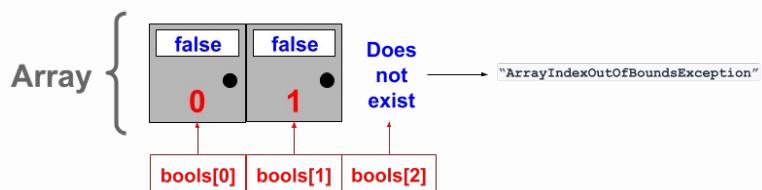
challenge

### What happens if you:

- Change `decimals[0]` in the code above to `decimals[1]`?
- Change `bools[0]` in the code above to `bools[1]`?
- Change `decimals[0]` in the code above to `decimals[2]`?
- Change `bools[0]` in the code above to `bools[2]`?

## IndexOutOfBoundsException Error

A common error that occurs is called the `ArrayIndexOutOfBoundsException` error. This happens when you try to access or print an element at an index that does not exist within the array. In the example above, `decimals[2]` and `bools[2]` both resulted in an `ArrayIndexOutOfBoundsException` because neither array has an index of 2. Both arrays have only indices `[0]` and `[1]` to hold their elements.



.guides/img/ArrayException

# Modifying an Array

---

## Array Modification

To modify an element within an array, simply find the index at which that element is stored and assign a new value to it.

```
int[] grades = {85, 95, 48, 100, 92};  
System.out.println(grades[2]);  
  
grades[2] = 88; //88 will replace 48 at index 2  
System.out.println(grades[2]);
```

challenge

### What happens if you:

- Change `int[] grades = {85, 95, 48, 100, 92};` in the code above to `int[] grades = new int[5];`?
- Change all `System.out.println(grades[2]);` in the code above to `System.out.println(grades[3]);`?
- Change `grades[2] = 88;` in the code above to `grades[3] = 100;`?

## Modifying Multiple Arrays

You can create and modify as many arrays as you'd like. For example, you can create an array to store your family members and another array to store their age.

```
String[] family = {"Dad", "Mom", "Brother", "Sister"};  
int[] age = new int[4];  
  
System.out.println(family[0] + " " + age[0]);  
System.out.println(family[1] + " " + age[1]);  
System.out.println(family[2] + " " + age[2]);  
System.out.println(family[3] + " " + age[3]);
```

challenge

## What happens if you:

- Add `age[0] = 50;` directly below the line `int[] age = new int[4];?`
- Add `age[1] = 45;` below the line `int[] age = new int[4];` but before the print statements?
- Add `age[2] = 25;` below the line `int[] age = new int[4];` but before the print statements?
- Add `age[3] = 20;` below the line `int[] age = new int[4];` but before the print statements?
- Change "Sister" within the String array to "Brother2"?

important

## IMPORTANT

Since the integer array above was created using the *new* method, `0` was populated as elements within the array at first. Then by setting the array indices to specific values, you were able to modify the array to include the appropriate age for each family member.

# Iterating an Array

---

## Array Iteration

Though we can add many elements to our array, printing each of them can get quite tedious. For example, if we have 10 names of friends in our array, we would need to specify each of their array index to print them.

```
String[] friends = {"Alan", "Bob", "Carol", "David", "Ellen",
                     "Fred", "Grace", "Henry", "Ian", "Jen"};  
  
System.out.println(friends[0]);
System.out.println(friends[1]);
System.out.println(friends[2]);
System.out.println(friends[3]);
System.out.println(friends[4]);
System.out.println(friends[5]);
System.out.println(friends[6]);
System.out.println(friends[7]);
System.out.println(friends[8]);
System.out.println(friends[9]);
```

Luckily, we can use loops which we had learned previously to help us with this process. To print out all of our friends' names without repeating the print statement ten times, we can use a `for` loop to iterate 10 times.

```
String[] friends = {"Alan", "Bob", "Carol", "David", "Ellen",
                     "Fred", "Grace", "Henry", "Ian", "Jen"};  
  
for (int i = 0; i < 10; i++) {
    System.out.println(friends[i]);
}
```

challenge

## What happens if you:

- Change `System.out.println(friends[i]);` in the code above to `System.out.println(friends[0]);?`
- Change `System.out.println(friends[i]);` in the code above to `System.out.println(friends[10]);?`

important

## IMPORTANT

Did you notice that the print statement above includes `i` as the index for `friends`? We do this because `i` will take on the values specified by the `for` loop. The loop starts at `0` and increments by `1` until it reaches `9` (not including `10`). Thus, `friends[0]` will print, then `friends[1]`, so on and so forth until `friends[9]` is printed. Then the loop ends.

## Array Length

To make the iteration process easier, we can use an instance variable called `length` to determine how many elements are in our array. To use `length`, just call it by adding a period `.` after our array followed by `length`. For example, `friends.length` will tell us how many elements are in our `friends` array. The advantage of using `length` is that we can initialize additional elements in our array without having to keep track of how many elements are already inside.

```
String[] friends = {"Alan", "Bob", "Carol", "David", "Ellen",
                    "Fred", "Grace", "Henry", "Ian", "Jen"};

for (int i = 0; i < friends.length; i++) {
    System.out.println(friends[i]);
}
```

challenge

## What happens if you:

- add "Kate" as an element to the array right after "Jen"?
- remove "Alan" and "Bob" from the array?

Notice how `friends.length` continues to keep track of how many elements are in our array even though we've made several changes.

# Enhanced For-Loop

## Using an Enhanced For-Loop

There is a special type of `for` loop that can be used with arrays called an **enhanced for loop**. An enhanced for loop, also known as a **for each loop**, can be used to iterate through array elements without having to refer to any array indices. To use an enhanced for loop, you need the following:

- \* The keyword `for` followed by parentheses `()`.
- \* A **typed** iterating variable followed by colon `:` followed by the array name.
- \* **Note** that the iterating variable must be of the *same* type as the array.
- \* Any commands that repeat within curly braces `{}`.
- \* **Note** that when using an enhanced for loop, you can print the iterating variable itself without using brackets `[]`.

```
String[] friends = {"Alan", "Bob", "Carol", "David", "Ellen",
                    "Fred", "Grace", "Henry", "Ian", "Jen"};  
  
for (String i : friends) {
    System.out.println(i);
}
```

challenge

### What happens if you:

- change `System.out.println(i);` in the code above to  
`System.out.println(friends[i]);?`
- change `String i` in the code above to `int i?`

important

## IMPORTANT

One of the main differences between a regular `for` loop and an enhanced `for` loop is that an enhanced `for` loop does not refer to any index or position of the elements in the array. Thus, if you need to access or modify array elements, you **cannot** use an enhanced `for` loop. In addition, you **cannot** use an enhanced `for` loop to iterate through a *part* of the array. Think of an enhanced `for` loop as an *all-or-nothing* loop that just prints all of the array elements or nothing at all. Also note that the iterating variable type **must match** the array type. For example, you cannot use `for (int i : friends)` since `friends` is a String array and `i` is an integer variable. Use `for (String i : friends)` instead.

# Helpful Array Algorithms

---

## Array Algorithms

In addition to being used with loops, arrays can also be used with conditionals to help with tasks such as searching for a particular element, finding a minimum or maximum element, or printing elements in reverse order.

### Searching for a Particular Element

```
String[] cars = {"Corolla", "Camry", "Prius", "RAV4",
                 "Highlander"};
String Camry = "A Camry is not available."; //default String
                                              value

for (String s : cars) { //enhanced for loop
    if (s.equals("Camry")) { //if "Camry" is in array
        Camry = "A Camry is available."; //variable changes if
                                         "Camry" exists
    }
}

System.out.println(Camry); //print whether Camry exists or not
```

challenge

### What happens if you:

- delete "Camry" from the cars array?
- try to modify the code above so that the algorithm will look for Prius in the array and will print A Prius is available. if Prius is an element and A Prius is not available. if it is not an element.

---

#### ▼ Sample Solution

```

String[] cars = {"Corolla", "Camry", "Prius", "RAV4",
                 "Highlander"};
String Prius = "A Prius is not available.';

for (String s : cars) {
    if (s.equals("Prius")) {
        Prius = "A Prius is available.";
    }
}

System.out.println(Prius);

```

## Finding a Minimum or Maximum Value

```

int[] grades = {72, 84, 63, 55, 98};
int min = grades[0]; //set min to the first element in the array

for (int i : grades) { //enhanced for loop
    if (i < min) { //if element is less than min
        min = i; //set min to element that is less
    }
}
//elements are not modified so enhanced for loop can be used

System.out.println("The lowest grade is " + min); //print lowest
                                                 element

```

challenge

### What happens if you:

- replace 72 in the int array with 42?
- try to modify the code so that the algorithm will look for the **maximum** element instead?

---

#### ▼ Sample Solution

```
int[] grades = {72, 84, 63, 55, 98};  
int max = grades[0];  
  
for (int i : grades) {  
    if (i > max) {  
        max = i;  
    }  
}  
  
System.out.println("The highest grade is " + max);
```

---

## Printing Elements in Reverse Order

```
String[] letters = {"A", "B", "C", "D", "E"};  
  
//start at index 4, then decrement by 1 until i < 0, then stop  
for (int i = letters.length - 1; i >= 0; i--) {  
    System.out.println(letters[i]);  
}  
  
//regular for loop needed to access each element index
```

# **Formative Assessment 1**

---

## **Formative Assessment 2**

---

# Learning Objectives: ArrayLists

---

- Create an empty ArrayList
- Add and remove ArrayList elements
- Get and set ArrayList elements
- Iterate through ArrayLists using both a regular `for` loop and an *enhanced for* loop
- Determine ArrayList output
- Determine key differences between ArrayLists and arrays

# Creating an ArrayList

---

## What Is an ArrayList?

Although arrays are very useful for data collection, they are considered **static**, meaning once they are created, you cannot add or remove elements from them without changing the way they are initialized. **ArrayLists**, on the other hand, are **dynamic**, meaning you can make changes to them while the program is running. ArrayLists are particularly helpful when you don't know how large your collection of elements will become. Since ArrayLists are dynamic, you can add and remove elements later on if needed. In order to use ArrayLists, you must import it using `import java.util.ArrayList;` in the header of your program. For convenience, the program file to your left already contains the import statement.

## ArrayList Creation

To create an ArrayList, you need to include the following:

- \* The keyword `ArrayList` followed by the data type in angle brackets `<>`.
- \* Note that unlike arrays, ArrayList types are labeled slightly differently (e.g. `Integer`, `Double`, `Boolean`, and `String`).
- \* A variable name that refers to the ArrayList.
- \* The assignment operator `=` followed by the keyword `new`.
- \* Another `ArrayList` keyword followed by the data type in angle brackets `<>` followed by empty parentheses `()`.

```
ArrayList<Integer> numbers = new ArrayList<Integer>();  
  
System.out.println(numbers);
```

important

## IMPORTANT

Unlike printing arrays, printing an ArrayList will output its elements instead of its memory address. When an ArrayList is initialized, it is empty by default. This is why printing a new ArrayList results in empty brackets `[]`.

## Determining ArrayList Size

ArrayLists use the method `size()` to determine the number of elements that exist instead of `length` which is used for arrays. When an ArrayList is initially created, its size is automatically `0` since all new ArrayLists are empty by default.

```
ArrayList<Integer> numbers = new ArrayList<Integer>();  
  
System.out.println(numbers.size());
```

# Adding and Removing Elements

---

## Adding ArrayList Elements

To add elements to the ArrayList, simply use the `add()` method. The `add()` method will add whatever element that is specified inside parentheses () to the end of the ArrayList by default. If an element is added to an empty ArrayList, that element will be the first and only element in the ArrayList.

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add(50); //add 50 as an element to end of ArrayList

System.out.println(numbers);
```

challenge

### What happens if you:

- add `numbers.add(100);` directly below `numbers.add(50);`?
- add `numbers.add(12.3);` below `numbers.add(50);` but before the print statement?

important

### IMPORTANT

Much like arrays, ArrayLists can only store one type of data. For example, you cannot store 12.3 into an Integer ArrayList.

To add an element to a *specific index* in the ArrayList, you can use the `add()` method with two parameters inside the parentheses (). The first parameter is the index where you want the element to be stored at and the second parameter is the element itself. For example, `numbers.add(0, 12)` will add the number 12 to index 0 causing 12 to become the first element in the ArrayList. This will cause all of the elements to the right of 12 to move up by 1 index number.

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add(50);
numbers.add(100);
System.out.println(numbers);

numbers.add(0, 12); //add 12 as an element to index 0
System.out.println(numbers);
```

challenge

## What happens if you:

- add `numbers.add(2, 75);` directly below `numbers.add(0, 12);`?
- add `numbers.add(4, 250);` below `numbers.add(0, 12);` but before the second print statement?
- add `numbers.add(8, 320);` below `numbers.add(0, 12);` but before the second print statement?

important

## IMPORTANT

Adding `numbers.add(8, 320);` produces the familiar `IndexOutOfBoundsException` error. This occurs because the `ArrayList` does not contain index number 8. However, you can add an element to the `ArrayList` if you specify the last available array index plus 1. For example, if the last available index is 3, then you can use `numbers.add(4, 250);` to add 250 to index 4 which did not exist previously.

## Removing `ArrayList` Elements

To remove an element from an `ArrayList`, use the `remove()` method and specify the `ArrayList` index of the element you want to be removed as a parameter inside the parentheses `()`. Deleting an element will cause all elements to the right of that element to move down by 1 index number.

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add(12);
numbers.add(50);
numbers.add(75);
numbers.add(100);
System.out.println(numbers);

numbers.remove(2); //remove element at index 2
System.out.println(numbers);
```

challenge

### What happens if you:

- add another `numbers.remove(2);` directly below  
`numbers.remove(2);?`
- add a third `numbers.remove(2);` directly below the other two  
`numbers.remove(2);s?`

# Getting and Setting Elements

---

## Getting ArrayList Elements

To get or access ArrayList elements, use the `get()` method and include the index as a parameter inside parentheses `()`.

```
ArrayList<String> contact = new ArrayList<String>();
contact.add("First name");
contact.add("Last name");
contact.add("Phone number");

System.out.println(contact.get(0)); //gets element at index 0
and prints
```

challenge

### What happens if you:

- change `contact.get(0)` in the code above to `contact.get(1)`?
- change `contact.get(0)` in the code above to `contact.get(2)`?
- change `contact.get(0)` in the code above to `contact.get(3)`?

## Setting ArrayList Elements

To set or modify ArrayList elements, use the `set()` method which includes two parameters within parentheses `()`. The first parameter specifies the ArrayList index and the second parameter specifies the element that will replace the current value at the index. For example, `contact.set(2, "Email")` will modify the element at index 2 and change it to Email.

```
ArrayList<String> contact = new ArrayList<String>();
contact.add("First name");
contact.add("Last name");
contact.add("Phone number");
System.out.println(contact);

contact.set(2, "Email"); //change element at index 2 to "Email"
System.out.println(contact);
```

challenge

## What happens if you:

- add `contact.set(0, "Full name");` to the line directly before `contact.set(2, "Email");?`
- change `contact.set(2, "Email");` in the code above to `contact.set(1, "Address");?`
- change `contact.set(2, "Email");` in the code above to `contact.set(3, "Alternative name");?`

important

## IMPORTANT

Both `get()` and `set()` methods require that the `ArrayList` already has an element that exists at the specified index. Otherwise, the `IndexOutOfBoundsException` error will occur.

# Iterating an ArrayList

---

## Iterating ArrayList Elements

Iterating through an ArrayList is very similar to iterating through an array. The main difference is that in an ArrayList, we use `get()` to access the elements instead of brackets `[]`. Both of the code blocks below use a regular `for` to produce the exact same results. The first code block contains an array and the second contains an ArrayList.

```
//iterating through an array
int[] grades = {85, 95, 48, 100, 92};

for (int i = 0; i < grades.length; i++) {
    System.out.println(grades[i]);
}
```

```
//iterating through an ArrayList
ArrayList<Integer> grades = new ArrayList<Integer>();
grades.add(85);
grades.add(95);
grades.add(48);
grades.add(100);
grades.add(92);

for (int i = 0; i < grades.size(); i++) {
    System.out.println(grades.get(i));
}
```

## Enhanced For Loop in ArrayList

We can also use an **enhanced for loop** to iterate through an ArrayList.

```
//iterating an ArrayList with Enhanced For Loop
ArrayList<Integer> grades = new ArrayList<Integer>();
grades.add(85);
grades.add(95);
grades.add(48);
grades.add(100);
grades.add(92);

for (Integer i : grades) { //Integer is required instead of int!
    System.out.println(i);
}
```

important

## IMPORTANT

When using an enhanced for loop for an ArrayList, you must label the iterating variable accordingly. Remember that ArrayLists use Integer, Double, and Boolean instead of int, double, and boolean. Only String is consistently labeled between ArrayLists and arrays. Therefore, `for (Integer i : grades)` is required instead of `for (int i : grades)`.

# ArrayList vs. Array

---

## ArrayList vs. Array

Which one is better: ArrayList or array? The answer is, it really *depends*. If you know how many elements you need in your collection and you don't intend on changing the order of those elements, then it is better to use an **array**. On the other hand, if you don't know how many elements you need and want to modify the order of elements later on, then it is better to use an **ArrayList**.

Although an array is shorter to write and arguably easier to use, it is **static**, meaning it is not possible to add additional elements to the array after it has already been initialized. In contrast, an ArrayList is more **dynamic**, meaning you can add, remove, and reorganize elements as needed later on.

Here is a table showing the differences between ArrayLists and arrays. Note that uppercase Type stands for compatible *ArrayList* types while lowercase type stands for compatible *array* types. Also note that var stands for ArrayList or array name, num stands for an integer number, index stands for index or position number, and element stands for an ArrayList or array element.

Method/Types	ArrayList	Array
Create	<code>ArrayList&lt;Type&gt; var = new ArrayList&lt;Type&gt;()</code>	<code>type[] var = new type[num];</code> <code>or type[] var = {element, element...};</code>
Find number of elements	<code>var.size()</code>	<code>var.length</code>
Access an element	<code>var.get(index)</code>	<code>var[index]</code>
Modify an element	<code>var.set(index, element)</code>	<code>var[index] = element</code>
Add an element	<code>var.add(element) or</code> <code>var.add(index, element)</code>	n/a
Remove an element	<code>var.remove(index)</code>	n/a
for loop	<code>for (int i = 0; i &lt; var.size(); i++)</code> <code>{System.out.println(var.get(i));}</code>	<code>for (int i = 0; i &lt; var.length;</code> <code>i++)</code> <code>{System.out.println(var[</code>

Enhanced for loop	for (Type i : var) {System.out.println(i)}	for (type i : var) {System.out.println(i)}
Common compatible types	Integer, Double, Boolean, Strings	int, double, boolean, Strings

## Using Both an ArrayList and Array

ArrayLists and arrays can be used in tandem with each other. For example, the following code keeps track of the top five students in a class.

```
String[] top = {"First: ", "Second: ", "Third: ", "Fourth: ",
               "Fifth: "};
ArrayList<String> names = new ArrayList<String>();

names.add("Alan");
names.add("Bob");
names.add("Carol");
names.add("David");
names.add("Ellen");

for (int i = 0; i < 5; i++) {
    System.out.println(top[i] + names.get(i));
}
```

challenge

**Without deleting any existing code, try to:**

- switch Alan and Carol's places.
- replace David with Fred.
- make Grace get "Fifth" place and remove Ellen from the list.

### ▼ Sample Solution

```
String[] top = {"First: ", "Second: ", "Third: ", "Fourth: ",
                "Fifth: "};
ArrayList<String> names = new ArrayList<String>();

names.add("Alan");
names.add("Bob");
names.add("Carol");
names.add("David");
names.add("Ellen");

names.set(0, "Carol"); //switch Alan with Carol
names.set(2, "Alan"); //and vice versa

names.set(3, "Fred"); //Fred replaces David

names.add(4, "Grace"); //Grace takes Ellen's place
names.remove(5); //Ellen's "Sixth" place gets removed

for (int i = 0; i < 5; i++) {
    System.out.println(top[i] + names.get(i));
}
```

# Helpful ArrayList Algorithms

---

## ArrayList Algorithms

Like arrays, ArrayLists can be used to search for a particular element and to find a minimum or maximum element. Additionally, ArrayLists can reverse the order of elements rather than just simply printing the elements in reverse order.

### Searching for a Particular Element

```
ArrayList<String> cars = new ArrayList<String>();
String Camry = "A Camry is not available." //default String
value

cars.add("Corolla");
cars.add("Camry");
cars.add("Prius");
cars.add("RAV4");
cars.add("Highlander");

for (String s : cars) { //enhanced for loop
    if (s.equals("Camry")) { //if "Camry" is in ArrayList
        Camry = "A Camry is available." //variable changes if
        "Camry" exists
    }
}

System.out.println(Camry); //print whether Camry exists or not
```

challenge

### What happens if you:

- add `cars.remove(1);` to the line directly below  
`cars.add("Highlander");?`
- try to modify the code above so that the algorithm will look for Prius in the array and will print A Prius is available. if Prius is an element and A Prius is not available. if it is not an element.

## ▼ Sample Solution

```
ArrayList<String> cars = new ArrayList<String>();
String Prius = "A Prius is not available.";

cars.add("Corolla");
cars.add("Camry");
cars.add("Prius");
cars.add("RAV4");
cars.add("Highlander");

for (String s : cars) {
    if (s.equals("Prius")) {
        Prius = "A Prius is available.";
    }
}

System.out.println(Prius);
```

## Finding a Minimum or Maximum Value

```
ArrayList<Integer> grades = new ArrayList<Integer>();
grades.add(72);
grades.add(84);
grades.add(63);
grades.add(55);
grades.add(98);

int min = grades.get(0); //set min to the first element in the
array

for (int i : grades) { //enhanced for loop
    if (i < min) { //if element is less than min
        min = i; //set min to element that is less
    }
}
//elements are not modified so enhanced for loop can be used

System.out.println("The lowest grade is " + min); //print lowest
element
```

challenge

## What happens if you:

- add `grades.set(0, 42);` to the line directly below `grades.add(98);`?
- try to modify the code so that the algorithm will look for the **maximum** element instead?

---

### ▼ Sample Solution

```
ArrayList<Integer> grades = new ArrayList<Integer>();
grades.add(72);
grades.add(84);
grades.add(63);
grades.add(55);
grades.add(98);

int max = grades.get(0);

for (int i : grades) {
    if (i > max) {
        max = i;
    }
}

System.out.println("The highest grade is " + max);
```

---

## Reversing the Order of Elements

```
ArrayList<String> letters = new ArrayList<String>();
letters.add("A");
letters.add("B");
letters.add("C");
letters.add("D");
letters.add("E");

int original = letters.size(); //original size

//regular for loops needed to access element indices

for (int i = letters.size() - 1; i >= 0; i--) {
    letters.add(letters.get(i));
} //add elements in reverse order to the ArrayList

for (int j = 0; j < original; j++) {
    letters.remove(0);
} //remove all the original elements

System.out.println(letters); //print new ArrayList
```

important

## IMPORTANT

Note that we used `letters.remove(0)` rather than `letters.remove(j)` in the code above because `remove()` deletes both the **element** and the **index**. Thus, the next element in the `ArrayList` becomes the *new* 0th index which we want to continue to delete.

# **Formative Assessment 1**

---

## **Formative Assessment 2**

---

# Learning Objectives: 2D Arrays

---

- Create a 2D array using both the *initializer list* and *new* methods
- Access and modify 2D array elements
- Iterate through 2D arrays using both a regular `for` loop and an *enhanced for* loop
- Determine 2D array output

# Creating a 2D Array

## An Array Inside Another Array

An array inside another array is called a **2D array**. A 2D arrays is symbolic of a table where there are rows and columns. The first index number represents the **row** position and the second index number represents the **column** position. Together, the row and column indices enable elements to be stored at specific locations.

		Columns				
		0	1	2	3	4
Rows	0	Alan	Bob	Carol	David	Ellen
	1	00	01	02	03	04
	2	Fred	Grace	Henry	Ian	Jen
		10	11	12	13	14
		Kelly	Liam	Mary	Nancy	Owen
		20	21	22	23	24

.guides/img/2DArray

```
String[][] names = new String[3][5];
```

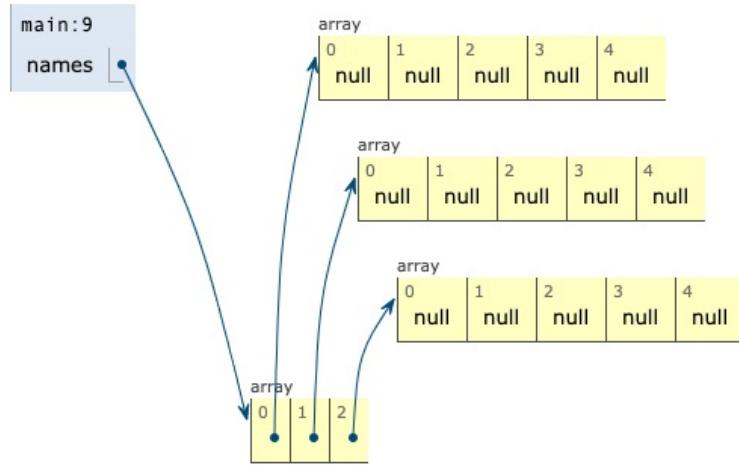
## 2D Array Syntax

- array type followed by **two** empty brackets `[][]` followed by a name for the 2D array.
- The `=` operator followed by the keyword `new` followed by the array type and two brackets `[][]`.
- The number of **rows** goes inside the **first** bracket and the number of **columns** goes inside the **second** bracket.

## Why Array Inside Array?

The way 2D arrays store elements is a little unique. Rather than creating an actual table like shown above, each initial *row* of the 2D array actually refers to another *column* array. This is why a 2D array is considered to be

an array inside of another array.



.guides/img/2DArrayReference

To determine the number of rows and columns in the 2D array, we can use the instance variable `length` like we did for arrays.

```
String[][] names = new String[3][5];  
  
System.out.println(names.length + " rows");  
System.out.println(names[0].length + " columns");
```

important

## IMPORTANT

Note that when determining column length, you must refer to the 2D array's initial row index. For example, `names[0].length` will calculate how many elements are inside row index 0 and these elements determine how many *columns* there are in the row.

# Accessing and Modifying a 2D Array

---

## 2D Array Access

To access and modify elements inside a 2D array, you need to specify the *row* and *column* indices at which the elements are located. For example `names[1][2]` refers to the element that's at row index 1 and column index 2.

		Columns				
		0	1	2	3	4
Rows	0	Alan	Bob	Carol	David	Ellen
	1	00	01	02	03	04
	2	Fred	Grace	Henry	Ian	Jen
	3	10	11	12	13	14
	4	Kelly	Liam	Mary	Nancy	Owen
	5	20	21	22	23	24

.guides/img/2DArray

Below is a code block showcasing a 2D array that contains fifteen P.O. Box names from a postal office. Note that you can use the **initializer list** method to populate elements inside your 2D array. Each *column* array is separated by curly braces {} as well as a comma ,.

```
String[][] names = { {"Alan", "Bob", "Carol", "David", "Ellen"},  
                     {"Fred", "Grace", "Henry", "Ian", "Jen"},  
                     {"Kelly", "Liam", "Mary", "Nancy", "Owen"}  
};  
  
System.out.println(names[1][2]);
```

challenge

## What happens if you:

- change `names[1][2]` within the print statement from above to `names[2][1]`?
- change `names[1][2]` within the print statement from above to `names[3][0]`?

important

## IMPORTANT

Note that you will still get an `ArrayIndexOutOfBoundsException` error if you attempt to access or modify an element at a row or column index that does not exist. Like arrays, you cannot add additional rows or columns of elements to the 2D array after it has been initialized.

## 2D Array Modification

To modify elements within a 2D array, simply access the element and assign another element to it.

```
String[][] names = { {"Alan", "Bob", "Carol", "David", "Ellen"},  
                     {"Fred", "Grace", "Henry", "Ian", "Jen"},  
                     {"Kelly", "Liam", "Mary", "Nancy", "Owen"}  
};  
  
System.out.println(names[1][2]);  
  
names[1][2] = "Harry";  
System.out.println(names[1][2]);
```

challenge

## What happens if you:

- change all `names[1][2]` within the code above to `names[0][0]`?
- change "Harry" in the code above to "Amy"?



# Iterating a 2D Array

---

## 2D Array Iteration

To iterate through a 2D array, we can use two `for` loops, one **nested** inside another. The outer `for` loop is for the rows while the inner `for` is for the columns.

```
int[][] digits = { {1, 2, 3},  
                   {4, 5, 6},  
                   {7, 8, 9} };  
  
for (int i = 0; i < digits.length; i++) {  
    for (int j = 0; j < digits[0].length; j++) {  
        System.out.println(digits[i][j]);  
    }  
}
```

challenge

### What happens if you:

- change `digits[0].length` in the inner `for` loop to `digits[1].length`
- change `println` in the print statement to `print`?

Note that all of the rows' lengths are the same, they each have three elements. Therefore, it doesn't matter if we use `digits[0].length`, `digits[1].length`, or `digits[2].length`. Also note that using `println` prints the elements vertically while `print` prints the elements horizontally. To print the elements so that the columns stay together but the rows separate, we can try something like this:

```

int[][] digits = { {1, 2, 3},
                  {4, 5, 6},
                  {7, 8, 9} };

for (int i = 0; i < digits.length; i++) {
    for (int j = 0; j < digits[0].length; j++) {
        if (j == digits[0].length - 1) {
            System.out.println(digits[i][j]);
        }
        else {
            System.out.print(digits[i][j] + " ");
        }
    }
}

```

The `if` conditional forces the elements to be printed with a newline every time the iterating variable reaches the end of the column index. Otherwise, the elements will be printed with a space instead.

## 2D Array with Enhanced For Loop

Like arrays and ArrayLists, 2D arrays can also make use of the **enhanced for loop**.

```

int[][] digits = { {1, 2, 3},
                  {4, 5, 6},
                  {7, 8, 9} };

for (int[] i : digits) {
    for (int j : i) {
        if ((j == 3) | (j == 6) | (j == 9)) {
            System.out.println(j);
        }
        else {
            System.out.print(j + " ");
        }
    }
}

```

Note that we cannot use an enhanced for loop to manipulate array indices. Our iterating variable goes through the 2D array and takes on each element value rather than each element index. This is why we have the conditional statement `if ((j == 3) | (j == 6) | (j == 9))` rather than `if (j == digits[0].length - 1)`. Additionally, since we have an array inside of another array, our iterating variable `i` is of type `int[]` rather than just `int`.



# **Formative Assessment 1**

---

## **Formative Assessment 2**

---

# The Java Documentation Tool

The Java Development Kit (JDK) contains a **Javadoc** tool that takes the information in the comments of a Java file and generates an HTML page with documentation for that file. For more detailed information, you may visit the [Javadoc website](#).

The **Javadoc** tool recognizes comments it should use by the extra asterisk at the opening of a multi-line comment:

```
/**  
 * This is a Javadoc comment  
 */  
  
/*  
 * This is not a Javadoc style comment  
 */
```

Javadoc comments are typically placed above classes, methods or fields. The first part of the comment is the description followed by the block tags (beginning with @ sign) to describe meta data. The description should go beyond a repeat of the name of the class or method.

Javadoc style conventions:

- The first line contains the opener for a Javadoc comment /\*\*
- The first sentence should be a short summary of the API item (member, class, interface or package description)
- The summary should distinguish between different overloaded methods
- Separate paragraphs with a <p> paragraph tag
- Insert a blank comment line between the description and the tags
- The first line with an @ tag marks the end of the descriptions block
- The last line should be the comment closing \*/

## Best practices for documenting Java code

Every class and method should be preceded by a Javadoc style comment

**Class comments should contain:**

- The
- The
- The

**Method comments should contain:**

- The
- An
- A if there is one or a comment to the effect that nothing is returned.
- A

**Generate the JavaDoc**

- Paste the following command in the terminal window

```
javadoc BookReader.java -d jdoc
```

**Open up the generated document and answer the question below.**

# Javadoc Tags

The Javadoc tool recognizes a set of tags when they are inside a Javadoc type comment (starts with `\*\*`). All tags must start with a @ sign and are case sensitive (must follow the casing in the list below). The tag must be the first thing in a line, leading spaces or a preceding (optional) asterisk are okay. More information about tags may be found [here](#).

There are two types of tags:

- **Block tags** - these start with an @ sign and must be in the tag section after the main description
- **Inline tags** - inline tags are surrounded by curly braces. They can be anywhere in the main description or in the comments for the block tags.

## Tag Ordering

The following is the ordering convention:

1. @author (classes and interfaces only, required)
2. @version (classes and interfaces only, required)
3. @param (methods and constructors only)
4. @return (methods only)
5. @exception (or @throws)
6. @see
7. @since
8. @serial (or @serialField or @serialData)
9. @deprecated

## Order of multiple tags if the same kind

Tags with the same name are grouped together and follow these conventions:

- @author tags should be listed chronologically, the creator of the class should appear at the top.
- @param tags should be listed in the order they are declared in.
- @exception tags (also known as @throws) should be listed alphabetically by their exception name.
- @see tags should be from nearest to farthest access and from least-qualified to fully-qualified.

## Tags and their purpose

---

Tag (value)	Purpose
@author name	Creates an author entry. You can create multiple author entries or one with multiple names
{@code text}	Display the text provided in code font and as it is, not interpreted.
{@docRoot}	Used to specify the relative path of the doc's root directory
@deprecated explanation	You can use this tag in any of the doc comments for: overview, package, class, interface, constructor, method and field
@exception class-name description	The class-name is the exception that may be thrown by the method.
{@inheritDoc}	Copies documentation from the nearest inheritable class
{@link package.class#member label}	Inserts an in-line link with a visible text label that points to the documentation for the specified package
{@linkplain}	Similar to @link but the label is in text format
{@literal text}	Display the text provided as is, not interpreted.
@param name description	Adds a parameter with the specified parameter-name followed by the specified description to the "Parameters" section
@return description	Creates a Returns section with the description text
@see string	The quoted string is displayed as is
@see <a href="spec.html#section">Java Spec</a>	Adds the defined link
@see package.class#member label	Adds a link, with a text label, that points to the documentation for the specified name referenced in the argument
@serial field-description (include or exclude)	Used in the doc comment for a default serializable field.
@serialData data-description	Documents the types and order of data in the serialized form
@serialField field-name field-	Documents an ObjectStreamField

<code>type field-description</code>	<code>component</code>
<code>@since since-text</code>	Documents that the change or feature has existed since the release specified by the since-text
<code>@throws</code>	Same as <code>@exception</code>
<code>{@value} package.class#field</code>	Can display the value of the specified argument
<code>@version version-text</code>	Adds a “Version” subheading with the specified version-text to the generated docs

---

# **Classes vs Objects (MOD-1.B.1, MOD-1.B.2)**

---

## **Vocabulary**

The words “class” and “object” are used in an almost interchangeable manner. There are many similarities between classes and objects, but there is also an important difference.

**Classes** - Classes are a collection of data and the actions that can modify the data. Programming is a very abstract task. Classes were created to give users a mental model of how to think about data in a more concrete way. Classes act as the blueprint.

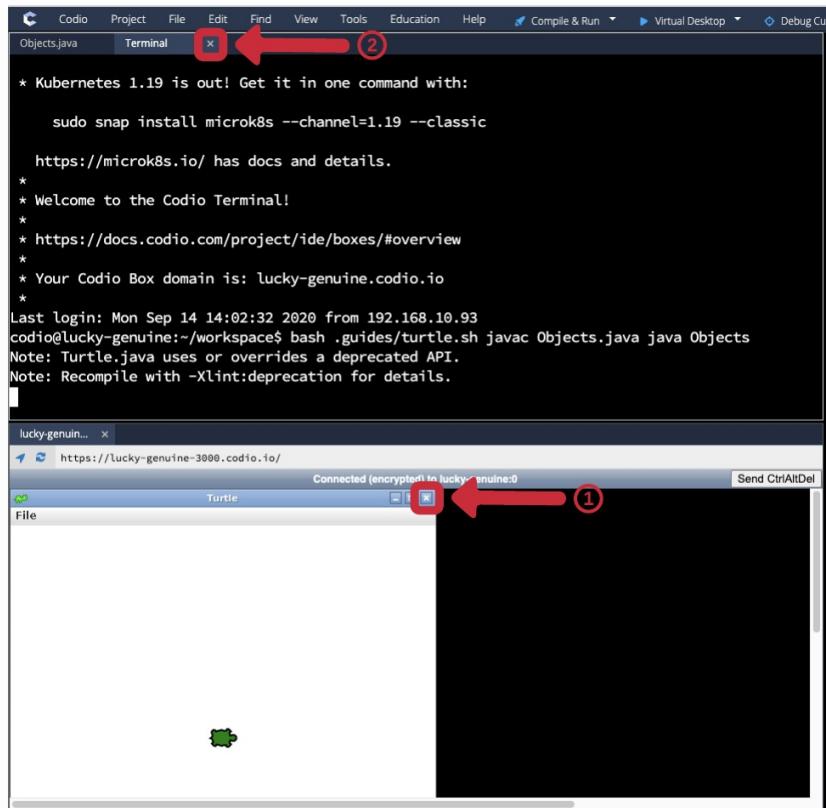
**Objects** - Objects are constructed according to the blueprint that is the class.

**Instance** - Another way that programmers talk about objects is to say that an object is an instance of a particular class.

## **Trying it out with Turtles**

There is a Turtle class that is a blueprint allowing you to easily create Turtle objects. The first Turtle object has been created in the code file in the top-left. Click the button below to see the created Turtle.

**Note:** To “end” Turtle programs, you will need to close the application in the bottom-left pane, then close the terminal in the top-left:

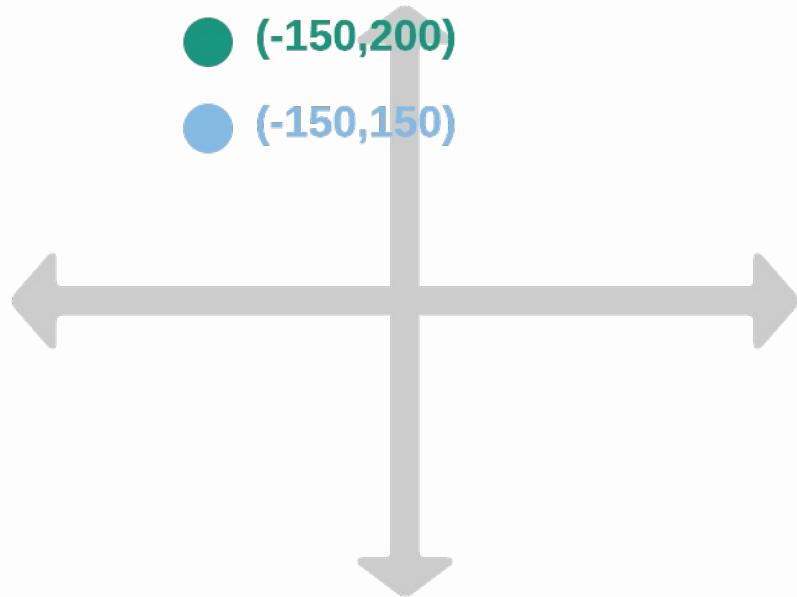


Let's create a second Turtle object:

```
//Creating Turtle 2
Turtle t2 = new Turtle(-150,150);
t2.fillColor(Color.blue);
```

Copy the above code onto Line 12 in the top-left pane.

If you compare the code for the second turtle to the code for the first turtle, you will notice that the will appear **below** the :



You will also notice that the .

#### Try creating more turtles

- Try making a pink third turtle, t3, at position -200,200
- Try making a yellow fourth turtle, t4, at position -200,150

# Class Attributes (MOD-1.B.1)

---

## Class Attributes

In the previous example, you might have noticed that to refer to a color, we used `Color.` before the color name. That's because [Java has a Color class which defines colors for us.](#)

### info

A class **attribute** is a variable associated with that class.

You can access attributes like variables – but you have to also say what class owns the attribute. The syntax is `className.attributeName` – like `Color.yellow`.

The Color class has a list of attributes or fields that represent a lot of the colors – this makes it easier for us to color turtles without having to represent colors in RGB or Red Green and Blue format.

### ▼ What is RGB?

The RGB color model is an additive color model in which red, green, and blue light are added together in various ways to reproduce a broad array of colors. The main purpose of the RGB color model is for the display of images in electronic systems, such as televisions and computers.[This widget let's you see how you can specify a color using the RGB values.](#)

### Try it out

With the Color class, the attributes mapping colors to RGB values stay the same. However, each of the turtles you made on the last page have different `fillColor` attributes. Try printing out the different attributes for each turtle by pasting this on line 24.

```
System.out.println("Turtle 1 is green or " + t1.fillColor);
System.out.println("Turtle 2 is blue or " + t2.fillColor);
System.out.println("Turtle 3 is pink or " + t3.fillColor);
System.out.println("Turtle 4 is yellow or " + t4.fillColor);
```

# Instantiation (MOD-1.D.1, MOD-1.D.2, MOD-1.D.3, VAR-1.D.1)

---

In the previous lesson, we created `Turtle` objects.

info

**Instantiation** is the process of creating a new object from a class

Every time you instantiate or create an object, you will use the keyword `new`. For example:

```
Turtle t1 = new Turtle();
```

If you remember back to variables, this is actually two steps:

```
Turtle t1;  
t1 = new Turtle();
```

If an object has been declared but **not** initialized (i.e. `Turtle t1;`) then is called `null`.

info

The keyword `null` is a special value used to indicate that a reference is not associated with any object.

The second piece (`t1 = new Turtle();`) is actually calling the **constructor**. That's why the first `Turtle` has no paranthesis but the second one does – the first one is simply stating what type of data is in `t1` - the second is calling a special method called a constructor (which is discussed on the next page). You won't have to guess the name of the method to instantiate an object – constructors always have the same name as the class. For example, to invoke the `Color` constructor you would do `Color red = new Color(255,0,0);`

At this point we have talked about the `Color` class and the `Turtle` class – both of which you did not write. When you write code, you don't have to start from nothing every time. Existing classes and class libraries can be

utilized as appropriate to create objects. Some of the most common (String, Integer, Double, Math and ArrayList) are listed on the [AP Java Quick Reference Sheet](#) so you don't have to memorize them but can use them when you write code on the AP exam.

## **Default Constructor (MOD-1.D.1, MOD-1.D.2, VAR-1.D.1, VAR-1.D.2)**

---

The **constructor** is a special method (more on methods in another lesson) for a class. Its job is to define all of attributes associated with the object. In Java, the constructor always has the same name as the class.

Use the visualizer on the left to see how the program executes. (If you get any kind of error, click the Refresh Code button in the top left). As you click the “Forward” button, notice on the following steps...

### **Step 3:**

- The program jumps from line 6 to line 19 because the `new Turtle()` calls the constructor
- There is an empty framework of a `Turtle` object created on the right with the class attributes
- The default value for `int` is 0
- The `fillColor` attribute has not been set, and since it's a `Color` object it is set to `null`

### **Step 11:**

- The `t1` object has the defualt values set in the default constructor
- Since `fillColor` is an `Object`, there is a reference/arrow instead of a value
- Since `x` and `y` are primitives, there is a value instead of a reference/arrow

### **Step 21:**

- `t1.x = 50` only effects the `t1` object

### **Step 22:**

- `t2.y = 200` only effects the `t2` object

The real `Turtle` constructor actually has a lot more attributes:

```
public Turtle()
{
    location=new Point2D.Double(0,0);
    direction=0; //degrees
    shape="turtle"; //Stores a key to the shapes hashmap
    image=null;
    shapeWidth=33;
    shapeHeight=33;
    penWidth=2;
    penColor=Color.BLACK;
    outlineWidth=2;
    outlineColor=Color.BLACK;
    fillColor=new Color(0,255,0,128);
    speed=50; //milliseconds to execute a move
    isPenDown=true;
    isFilling=false;
    isVisible=true;
}
```

# Parameters (MOD-1.C.1 - MOD-1.C.6, MOD-1.D.4)

---

Previously, we made a number of turtles – and we set their location when we instantiated them. This is because the `Turtle` class has **two** constructors. The one on the previous page is called the **default** constructor, meaning it makes a default turtle with no custom information. There is a second constructor that looks like:

```
public Turtle(double x, double y)
{
    location=new Point2D.Double(x,y);
    ...
}
```

You'll notice that unlike the previous constructor, this constructor has **parameters**. Instead of just setting the location to an arbitrary point, the user tells the constructor where to put the turtle.

info

**Parameters** are how you can pass information into methods and constructors.

Use the visualizer on the left to see how the example program executes. (If you get any kind of error, click the Refresh Code button in the top left). As you click the “Forward” button, notice on the following steps...

## Step 3:

- The program jumps from line 5 to line 17 because the `new Turtle()` calls the default constructor
- There is an empty framework of a `Turtle` object created on the right with the class attributes
- The default value for `int` is 0
- The `fillColor` attribute has not been set, and since it's a `Color` object it is set to `null`

## Step 11:

- The t1 object has the default values set in the default constructor
- Since fillColor is an Object, there is a reference/arrow instead of a value
- Since x and y are primitives, there is a value instead of a reference/arrow

## Step 12:

- The program jumps from line 6 to line 28 because the new Turtle(200, 200, Color.BLACK); calls the constructor with the **signature** that matches (public Turtle(int x1, int y1, Color col))

## Step 15:

- fillColor is set using the col parameter

## Steps 16 and 17:

- x and y are set to 200 instead of the default values because of the values passed through the constructors parameters

## Step 20:

- fillColor for t2 is a DIFFERENT color object than t1 since t1 is green and t2 is black.

## Step 21:

- The program jumps from line 7 to line 23 because the new Turtle(300, 300); calls the constructor with the **signature** that matches (public Turtle(int x1, int y1))

## Step 24:

- The fillColor attribute is set to the same color object as t1

## Steps 25 and 26:

- x and y are set to 300 instead of the default values because of the values passed through the constructors parameters

The Java program knows which constructor to used based on it's **signature**. A signature consists of the constructor name and the parameter list. In the example above, you can see how in steps 3, 12 and 21 the values in the instantiation step are being matched to the constructor with matching parameters.

info

To clarify, prgrammers often call the parameter list, in the header of a constructor, which lists the types of the values that are passed and their variable names, **formal parameters**.

In contrast, programmers often call the parameter value that is passed into a constructor **actual parameters or arguments**.

So `int x1, int y1` are the formal paramenters while `(300,300)` are the actual parameters

Parameters are passed using call by value. Call by value initializes the formal parameters with copies of the actual parameters.

## **Methods (MOD-1.E.1 - MOD-1.E.4, MOD-1.E.6, MOD-1.E.7)**

---

Up to this point we have focused on how to create or instantiate objects and an object's attributes. Objects also have behaviors. What the object can do (or what can be done to it) is defined by **methods**.

We've actually already used methods with the Turtle class. When we were setting the fillColor, we were using a method:

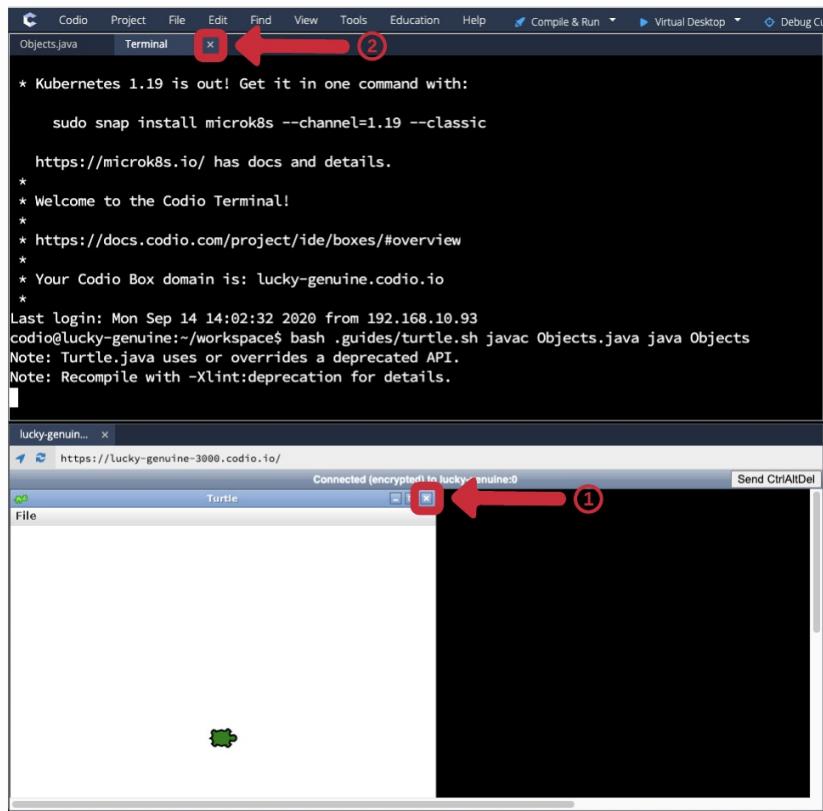
```
Turtle t1 = new Turtle();  
t1.fillColor(Color.blue);
```

The Turtle class has a method with the signature `fillColor(Color)` that allows us to change the Turtle color.

Similar to attributes, methods start with the object name:  
`objectName.methodName()`.

Check out the code on the left – it has a turtle draw a square.

### **▼ Reminder on how to close a Turtle program**



### How do we know the turtle will draw a square instead of random lines?

A method (or constructor) call interrupts the sequential execution of statements, causing the program to first execute the statements in the method (or constructor) before continuing.

Once the last statement in the method (or constructor) has executed, flow of control is returned to the point immediately following where the method (or constructor) was called.

In other words - the program finishes each line in order. It won't go to the next line until it has **finished** the method call on the current line.

### Try out some of these other Turtle methods:

Method	Description
forward()	Move the turtle forward 10 pixels
backward()	Move the turtle backward 10 pixels
right()	Turn the turtle to the right 90 degrees
left()	Turn the turtle to the left 90 degrees

Taking a look at the list of methods above, notice three things:

1. Programmer can use a method by knowing what the method does even if they do not know how the method was written. This is sometimes referred to as **procedural abstraction**
2. A method signature for a method without parameters consists of the method name and an empty parameter list ()
3. All of these methods do things – but they do not **return** values. Methods that do not return a value are called **void methods**.

## More Methods (MOD-1.E.5, MOD-1.E.8, Skill 3.A)

---

All of these Turtle methods are called through objects. These methods are non-static methods. That means these methods are only available when the object is available.

Try running this code:

```
t.forward();
Turtle t = new Turtle();
```

You get an error because `t` has not yet been instantiated when `t.forward()` is run.

You will also get an error if you have a `null` object.

Try running this code:

```
Turtle t = null;
t.forward();
t = new Turtle();
```

You should see something on the terminal that looks like:

```
Exception in thread "main" java.lang.NullPointerException
at MoreMethods.main(MoreMethods.java:7)
```

This **Null Pointer Exception** is thrown when using a `null` reference to call a method or access an instance variable/attribute.

# **Void Methods with Parameters**

## **(MOD-1.F.1, MOD-1.F.2, MOD-1.F.3,**

### **Skill 2.C)**

---

In the last lesson we learned how to make our turtles draw!

Method   Description
:---:   :---:
forward()   Move the turtle forward 10 pixels
backward()   Move the turtle backward 10 pixels
right()   Turn the turtle to the right 90 degrees
left()   Turn the turtle to the left 90 degrees

The methods we learned in the last lesson are acutally **overloaded**. You can use them with **parameters** for more flexibility.

definition

#### **Overloaded**

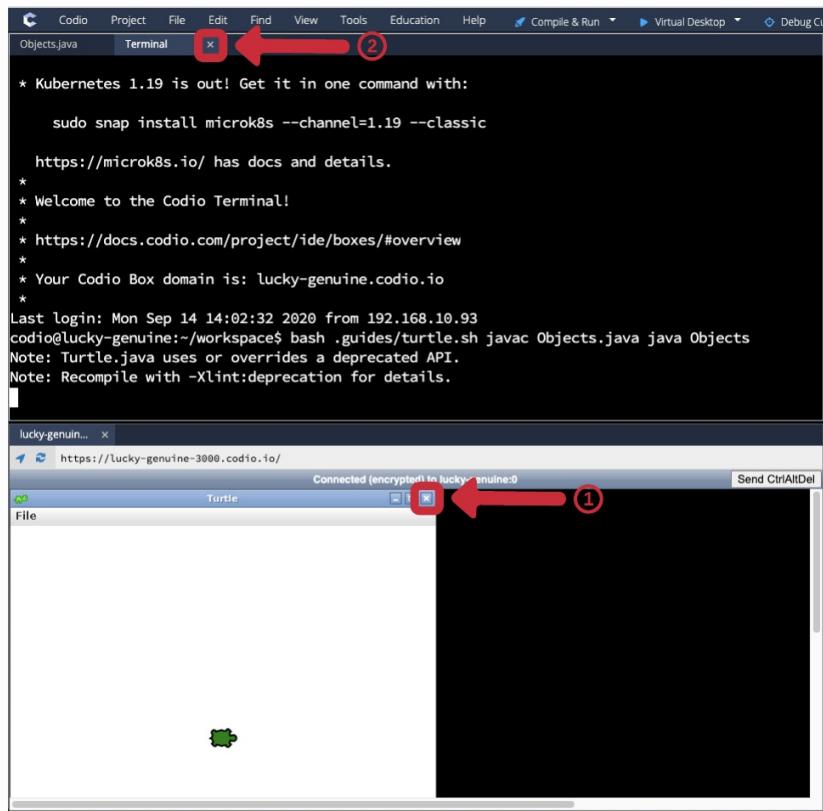
Methods are said to be **overloaded** when there are multiple methods with the same name but a different signature.

In the table below we see a second method signature, but you can have even more methods with the same name (as long as the method signature is unique).

<b>Command</b>	<b>Parameter</b>	<b>Description</b>
forward(n)	Where n represents the number of pixels	Move the turtle forward
backward(n)	Where n represents the number of pixels	Move the turtle backward
right(d)	Where d represents the number of degrees	Turn the turtle to the right
left(d)	Where d represents the number of degrees	Turn the turtle to the left

Try out the code on the left which draws a triangle.

▼ **Reminder on how to close a Turtle program**



Experiment with the turtle methods.

You can also speed up or slow down your turtle using the `speed(double delay)` method. The longer the delay, the **slower** your turtle will draw. The default speed is 50ms.

If you need to pick up() or put down() your turtle pen, there are methods for that too!

## Method Signatures

A method signature for a method with parameters consists of the method name and the ordered list of parameter types.

Just like constructors, when you are calling a method, the actual parameters or arguments must match the type and order of the formal parameters in the method header.

For example, the turtle method:

```
/**  
 * Sets the position and direction of a turtle.  
 * @param x x coordinate  
 * @param y y coordinate  
 * @param direction angle counter-clockwise from east in degrees  
 */  
public void setPosition(double x, double y, double direction)  
{...}
```

Even though there are 3 doubles, we know what order to list them in because of the method header.

# Non-void Methods (MOD-1.G.1, Skill 1.C)

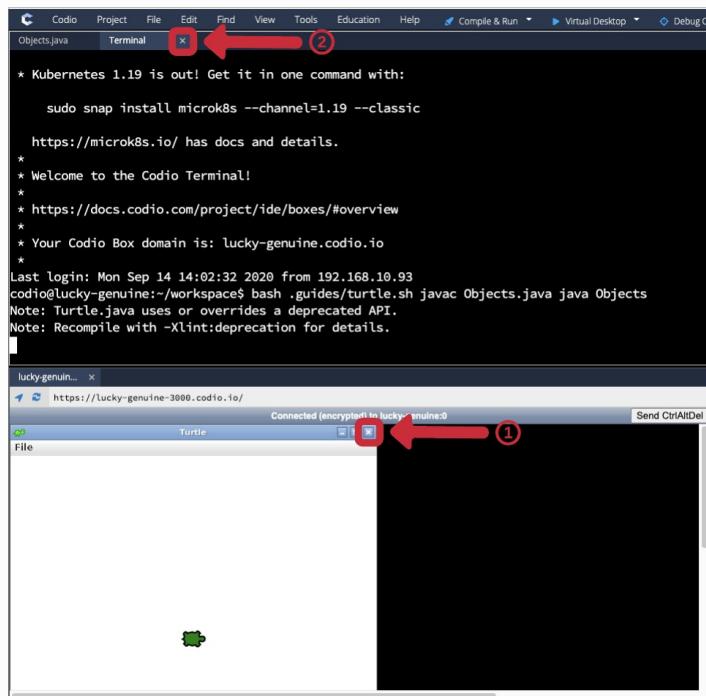
Non-void methods return a value that is the same type as the return type in the signature. To use the return value when calling a non-void method, it must be stored in a variable or used as part of an expression.

A couple of non-void or return functions in the Turtle class:

```
* public double getX()  
* public double getY()  
* public double towards(double x, double y)
```

Check out how they are used in the code to the left to draw a parabola.

## ▼ Reminder on how to close a Turtle program



.guides/img/closingTurtlePrograms

The code consists of the same code over and over again:

```
t.setPosition(t.getX()+1, (t.getX()+1)*(t.getX()+1), //increment  
X and set Y to X^2  
    t.towards(t.getX()+1, (t.getX()+1)*(t.getX()+1)) );  
//use towards to face turtle
```

# **Leap Frog**

---

Enjoy the slightly modified version of leap frog!

Feel free to modify it – changing the multiplier effects how far they jump.

# Creating a String (VAR-1.E.1, VAR-1.E.2)

---

Back in Unit 1, we saw `Strings`. While `int`, `double`, and `boolean` from Unit 1 are primitive types, `String` is an object.

Try instantiating it with its default constructor:

```
String test = new String();  
System.out.println(test);
```

Similar to `Turtle`, the `String` constructor is overloaded. Another constructor has the signature `public String(String original)`. Try out this constructor:

```
String test = new String("test");  
System.out.println(test);
```

Because `Strings` are so common, there is some syntactical sugar to make this process as easy as it is for primitives:

```
String test = "test2";  
System.out.println(test);
```

info

Strings are **immutable** meaning that `String` methods do not change the `String` object.

## Concatenation (VAR-1.E.3, VAR-1.E.4, Skill 2.A)

info

**Concatenation** is the joining of two Strings. For example, “Computer” concatenated with “Science!” would be “Computer Science!”

You can concatenate Strings using the + operator.

Try it out:

```
String first = "Computer ";
String second = "Science!";
System.out.println(first+second);
```

You might have realized, this means that the + operator is overloaded! Java knows if you are doing int + int or String + String):

```
String one = "1";
String two = "2";
System.out.println(one+two);
int oneInt = 1;
int twoInt = 2;
System.out.println(oneInt+twoInt);
```

You can even use the += operator with Strings:

```
String first = "Computer ";
first += "Science!";
System.out.println(first);
```

Finally, you can concatenate Strings with primitives (int, double, and boolean):

```
String first = "Computer ";
first += "Science is #";
first += 1;
System.out.println(first);
```

# Escape Sequences (VAR-1.E.5)

---

Strings can mess up certain characters. Try to print out this quote:  
“Computer Science is no more about computers than astronomy is about telescopes.” — Dijkstra

```
System.out.println(" "Computer Science is no more about  
    computers than astronomy is about telescopes." -  
    Dijkstra ");
```

info

**Escape sequences** start with a `\` and have a special meaning in Java.  
Escape sequences used in this course include `", \,` and `.`

Now try **escaping** the quotation marks:

```
System.out.println("\\"Computer Science is no more about  
    computers than astronomy is about telescopes.\\" -  
    Dijkstra");
```

Escaping removes or adds special meaning. For example, you can use `\n` to create a new line:

```
System.out.println("\\"Computer Science is no more about  
    computers \n than astronomy is about telescopes.\\" \n -  
    Dijkstra");
```

Similarly, because `\` signifies escape characters, you need to escape it as well:

```
System.out.println("The escape character is \\\\" );
```

## APIs and Libraries (VAR-1.E.6 - VAR-1.E.9)

Application program interfaces (APIs) and libraries simplify complex programming tasks. Strings are pretty complex, but because they are so useful, there is a library to handle them.

Documentation for APIs and libraries are essential to understanding the attributes and behaviors of an object of a class. For Java, most of the documentation for the most common libraries can be found on the Oracle website.

Classes in the APIs and libraries are grouped into packages. This hierarchy will be important later – for now just know that there are helpful bundles of libraries called packages.

If you take a look at the documentation for the String class (pane on the left), you'll notice:

- \* the **String** class is part of the `java.lang` package (which is available by default)
- \* the **Field** section lists **String** class attributes
- \* the **Constructor** section lists the different constructor signatures
- \* the **Methods** section lists the different methods you can use with Strings

Let's try out some of these String methods!

# String methods (VAR-1.E.10, VAR-1.E.12, VAR-1.E.13)

## String indeces

Strings are treated as a list of characters. A String object has index values from 0 to length – 1.

<b>String</b>	" H e l l o ! "
<b>Indexes</b>	0 1 2 3 4 5

.guides/img/StringIndices

You can access a subset or part of a String, called a `substring`, using the indices:

```
String test = "Hello World!";
System.out.println(test.substring(0,5)); //characters at indeces
                                         0-4
System.out.println(test.substring(6)); //characters at indeces
                                         6-end
```

You can see from the [AP Java Quick Reference sheet](#) that the `substring` method is overloaded. You can either specify both the start and end, or just the start (and the substring goes until the end of the string).

Attempting to access indices outside this range will result in an `IndexOutOfBoundsException`:

```
String test = "Hello World!";
System.out.println(test.substring(0, 13));
```

If you want to access a character at location `index`, you can use `substring(index, index + 1)`:

```
String test = "Hello World!";
System.out.println(test.substring(4, 5)); //gets character at
                                         index 4
```

## More String Methods

Let's try out the rest of the methods listed on the Quick Reference sheet:

### Java Quick Reference

*Accessible methods from the Java library that may be included in the exam*

Class Constructors and Methods	Explanation
<b>String Class</b>	
<code>String(String str)</code>	Constructs a new <code>String</code> object that represents the same sequence of characters as <code>str</code>
<code>int length()</code>	Returns the number of characters in a <code>String</code> object
<code>String substring(int from, int to)</code>	Returns the substring beginning at index <code>from</code> and ending at index <code>to - 1</code>
<code>String substring(int from)</code>	Returns <code>substring(from, length())</code>
<code>int indexOf(String str)</code>	Returns the index of the first occurrence of <code>str</code> ; returns <code>-1</code> if not found
<code>boolean equals(String other)</code>	Returns <code>true</code> if <code>this</code> is equal to <code>other</code> ; returns <code>false</code> otherwise
<code>int compareTo(String other)</code>	Returns a value <code>&lt;0</code> if <code>this</code> is less than <code>other</code> ; returns zero if <code>this</code> is equal to <code>other</code> ; returns a value <code>&gt;0</code> if <code>this</code> is greater than <code>other</code>

### String(String str)

Constructs a new String object that represents the same sequence of characters as str:

```
String test = new String("this is a constructor");
System.out.println(test);
```

### int length()

Returns the number of characters in a String object:

```
String test = "Hello World!";
System.out.println("The string \""+test+"\" has " +
    test.length() + " characters");
```

### int indexOf()

Returns the index of the first occurrence of str; returns -1 if not found

```
String test = "Hello World! I am a computer :)";
System.out.println(test.indexOf("computer"));
```

### boolean equals(String other)

Returns true if this is equal to other; returns false otherwise

```
String test = "Hello World!";
System.out.println(test.equals("Hello World!"));
```

### **int compareTo(String other)**

Returns a value < 0 if this is less than other; returns zero if this is equal to other; returns a value > 0 if this is greater than other

```
String test = "Hello World!";
System.out.println(test.compareTo("Hello Universe!"));
```

## toString (VAR-1.E.11)

Objects often hold complex data. However, all objects have a `toString()` method which attempts to represent the object as a String:

```
Integer four = new Integer(4);
Integer three = new Integer(3);

System.out.println(four.intValue() + three.intValue());
System.out.println(four.toString() + three.toString());
```

A String object can be concatenated with an object, which implicitly calls the referenced object's `toString` method:

```
Integer four = new Integer(4);
Integer three = new Integer(3);

System.out.println("The value of four is " + four);
System.out.println("The value of three is " + three);
```

# **Integer class (VAR-1.F.1, VAR-1.F.2, Skill 2.C)**

---

After the last lesson, you can probably guess why this code does not print “8”:

```
int a = 5;  
String b = "3";  
System.out.println(a + b);
```

However, you can convert b to an integer to fix the problem.

```
int a = 5;  
String b = "3";  
System.out.println(a + (new Integer(b)).intValue());
```

The `Integer` class, part of the `java.lang` package, has some helpful methods - including `intValue()`.

## **Integer class Methods**

### **Integer(int value) - constructor**

Constructs a new `Integer` object that represents the specified `int` value

```
Integer two = new Integer(2);
```

### **Integer.MIN\_VALUE and Integer.MAX\_VALUE**

The minimum and maximum value represented by an `int` or `Integer`

```
Integer min = new Integer(Integer.MIN_VALUE);  
Integer max = new Integer(Integer.MAX_VALUE);
```

### **int intValue**

Returns the value of this `Integer` as an `int`

```
System.out.println(two.intValue());  
System.out.println(min.intValue());  
System.out.println(max.intValue());
```

# Double class (VAR-1.F.1, VAR-1.F.3, Skill 2.C)

---

The Double class, part of the `java.lang` package, has some helpful methods.

## **Double(double value)**

Constructs a new Double object that represents the specified double value

```
Double two = new Double(2.3);
System.out.println(two);
```

## **double doubleValue**

Returns the value of this Double as a double

```
Double two = new Double(2.3);
System.out.println(two.doubleValue() * 2);
```

# Autoboxing and Unboxing (VAR-1.F.4 - VAR.1.F.7)

## Autoboxing

info

**Autoboxing** is the automatic conversion that the Java compiler makes between primitive types and their corresponding object wrapper classes. This includes converting an `int` to an `Integer` and a `double` to a `Double`.

The Java compiler applies autoboxing when a primitive value is:

- \* Passed as a parameter to a method that expects an object of the corresponding wrapper class.
- \* Assigned to a variable of the corresponding wrapper class.

For example:

```
Integer two = 2;  
Double threeIsh = 3.4;
```

## Unboxing

info

**Unboxing** is the automatic conversion that the Java compiler makes from the wrapper class to the primitive type. This includes converting an `Integer` to an `int` and a `Double` to a `double`.

The Java compiler applies unboxing when a wrapper class object is:

- \* Passed as a parameter to a method that expects a value of the corresponding primitive type.
- \* Assigned to a variable of the corresponding primitive type.

For example:

```
int two = new Integer(2);
```

# Static Math Methods (MOD-1.H.1, CON-1.D.1 - CON-1.D.3)

---

For the last library of this unit, we are going to cover the `Math` class. The `Math` class is also part of the `java.lang` package.

Unlike the previous classes we investigated, the `Math` class contains only **static** methods. This means we don't need to instantiate a `Math` object to use them - they are **ALWAYS** available.

Static methods are called with the syntax `ClassName.MethodName`. For example, `Math.abs(-3)` returns the absolute value of the argument.

## Try out the following Math methods:

### **int abs(int x)**

Returns the absolute value of an `int` value

```
System.out.println(Math.abs(-3));
```

### **double abs(double x)**

Returns the absolute value of an `double` value

```
System.out.println(Math.abs(-3.5));
```

### **double pow(base, exponent)**

Returns the value of the first parameter, `base`, raised to the power of the second parameter, `exponent`

```
System.out.println(Math.pow(2,0));
System.out.println(Math.pow(2,1));
System.out.println(Math.pow(2,2));
System.out.println(Math.pow(2,3));
System.out.println(Math.pow(2,4));
```

### **double sqrt(double x)**

Returns the positive square root of a double value

```
System.out.println(Math.sqrt(16));
System.out.println(Math.sqrt(9));
System.out.println(Math.sqrt(4));
System.out.println(Math.sqrt(1));
```

## Math.random (CON-1.D.3, CON-1.D.4, Skill 1.B)

---

`Math.random()` returns a double value greater than or equal to 0.0 and less than 1.0.

While this seems a little odd, you can manipulate this range based on what you need. For example, to make the range 0 to 99 you simply multiply by 100:

```
System.out.println((int) (Math.random()*100));
```

**Tip:** When playing with random numbers, run the program several times!

If I want to shift the range to say, -25 to 25, then there are two steps:

1. Scale the range to 0 to 50

```
System.out.println((int) (Math.random()*50));
```

2. Shift the range by -25 to get to -25 to 25

```
System.out.println((int) (Math.random()*50 - 25));
```

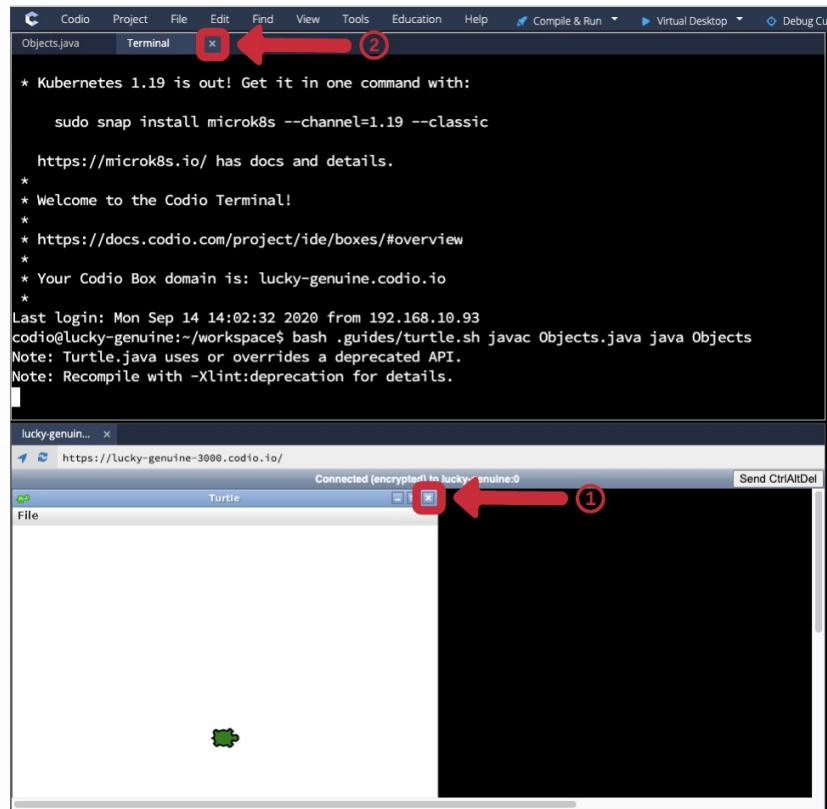
# Random Turtle

Enjoy the random Turtle program from the previous page – and feel free to explore!

Print statements have been added so you can see how far the turtle is turning and walking.

The turtle is also being set to a random color by generating random RGB values.

## ▼ Reminder on how to close a Turtle program



# Learning Objectives

---

- Recognize and understand the fundamental concepts of exceptions in Java, including the distinction between checked and unchecked exceptions
- Understand and implement the basic constructs of exception handling in Java, including the use of `try`, `catch`, and `finally` blocks.
- Handle exceptions using Java keywords

||| Info  
## Make Sure You Know

- Understand Java Syntax

# Understanding Exceptions in Java

## What are Exceptions?

In Java, exceptions are events that disrupt the normal flow of a program's instructions. They are objects that encapsulate information about an error state or unexpected behavior encountered by the program. Understanding exceptions is crucial for writing robust and error-resistant Java applications.

Java uses a class hierarchy to differentiate between different types of errors and exceptions. At the top of this hierarchy is the `Throwable` class, which has two main subclasses: `Error` and `Exception`.

`Error`: Represents serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.

`Exception`: Conditions that a program might want to catch.

Let's look at an example:

```
classDiagram
    Throwable <|-- Error
    Throwable <|-- Exception
    Exception <|-- RuntimeException
    Exception <|-- IOException
```

An exception is a condition that interrupts the usual execution of our programs. It can be thrown by the Java Virtual Machine (JVM) or manually by us under specific circumstances. To manage these exceptions, we write exception handlers, which are blocks of code designed to catch and address thrown exceptions. Additionally, we have the ability to declare exceptions in our method signatures, particularly when we anticipate that a method might encounter situations requiring exception handling, such as attempting to read a file that might not exist.

## Checked vs. Unchecked Exceptions

**Checked Exceptions:** These are exceptions that must be either caught or declared in the method signature. They extend from the `Exception` class but not from `RuntimeException`.

**Unchecked Exceptions:** These are exceptions that do not need to be explicitly caught or declared thrown. They extend from `RuntimeException`.

### Example of a Checked Exception

```
import java.io.*;

public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            FileInputStream file = new
FileInputStream("nonexistent.txt");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

### Example of an Unchecked Exception

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        System.out.println(numbers[3]); // Throws
ArrayIndexOutOfBoundsException
    }
}
```

### Common Java Exception Types

**NullPointerException:** Thrown when an application attempts to use null in a case where an object is required.

**ArrayIndexOutOfBoundsException:** Thrown to indicate that an array has been accessed with an illegal index.

**IOException:** Represents an input/output operation failure or interruption.

# **Basic Exception Handling: try...catch...finally**

## **Basic Exception Handling**

Exception handling is a critical component of Java programming, allowing developers to manage and recover from errors during execution. The try-catch-finally construct provides a structured way to detect errors, handle them gracefully, and perform cleanup activities, ensuring the program can continue to run or terminate cleanly.

## The try block

The try block encloses code that might throw an exception. If an exception occurs within the try block, execution of that block is stopped, and control is passed to the corresponding catch block.

Below is an example:

```
try {  
    // Code that might throw an exception  
}
```

## **The catch block**

The catch block is used to handle the exception. It must be associated with a try block and can specify which type of exception it can handle. You can have multiple catch blocks for different types of exceptions.

Below is an example:

```
```java-hide-clipboardtry {  
    try {  
        // Code that might throw an exception  
    } catch (ExceptionType name) {  
        // Code to handle the exception  
    }  
}
```

```
## The `finally` block
```

The `finally` block contains code that is always executed, regardless of whether an exception was thrown or caught. This block is typically used for cleanup activities, such as releasing resources.

Below is an example:

```
```java-hide-clipboard
try {
    // Code that might throw an exception
} catch (ExceptionType name) {
    // Code to handle the exception
} finally {
    // Code that is always executed
}
```

Click the button to run the code in the top-left panel, and enter a non-integer when prompted in the bottom-left panel:

When you enter a non-number, you will see something that looks like:

```
Enter integer => dd
Exception in thread "main" java.lang.NumberFormatException: For
input string: "dd"
at
java.base/java.lang.NumberFormatException.forInputString(NumberF
ormatException.java:65)
at java.base/java.lang.Integer.parseInt(Integer.java:652)
at java.base/java.lang.Integer.parseInt(Integer.java:770)
```

So to handle the exception, we add a try...catch block to the getInteger method:

```
public static int getInteger() {
    try {
        final Scanner scanner = new Scanner(System.in);
        System.out.print("Enter integer => ");
        return Integer.parseInt(scanner.nextLine());
    } catch(NumberFormatException wrongInput) {
        System.out.println("Invalid input! Please enter an
                           integer.");
        return getInteger();
    }
}
```

By wrapping the previous code in a `try...catch` block, we are able to handle the `NumberFormatException` by printing out a message to the user and re-running the method.

Update the code file in the top-left and try re-running the code file in the terminal in the bottom-left:

## try...catch...finally

A `finally` segment can be added to a `try...catch` block for code that needs to execute regardless of whether an exception occurs:

```
try {
    // code that might cause an exception
} catch(Exception e) {
    //code that runs if exception occurs
} finally {
    // code that always runs
}
```

Expanding our example from above, we need to call `.close()` method of `Scanner` to ensure it released input stream:

```
public static int getInteger() {
    final Scanner scanner = new Scanner(System.in);
    try {
        System.out.print("Enter integer => ");
        return Integer.parseInt(scanner.nextLine());
    } catch(NumberFormatException wrongInput) {
        System.out.println("Invalid input! Please enter an
                           integer.");
        return getInteger();
    } finally {
        scanner.close(); // we always call .close()
    }
}
```

Update the code file in the top-left and try re-running the code file in the terminal in the bottom-left:

## try-finally

It is possible to have a `try` block followed by a `finally` block (without the `catch` in between). If an exception is thrown in the `try` block, the program immediately executes the `finally` block before propagating the exception up the call stack.

There are a couple of important nuances to the `finally` block:

1. Avoid a `return` statement in the `finally` block because it ignores any `return` statements in the `try` or `catch` blocks.
1. Avoid throwing exceptions in the `finally` block because it disregards the exception thrown or `return` statements in the `try` and `catch` blocks.

# Creating and Using Custom Exceptions

There may be cases where the existing Exceptions provided by Java are not sufficient and you need to create an Exception class.

## How to Create an Exception Class

To create an exception, you write a new class which inherits from `Throwable` or one of its descendants.

The naming convention for custom exceptions is to end with `Exception`, for example:

```
public class MyException extends Exception {  
    public MyException() {  
    }  
  
    public MyException(final String message) {  
        super(message);  
    }  
  
    public MyException(final String message, final Throwable  
cause) {  
        super(message, cause);  
    }  
  
    public MyException(final Throwable cause) {  
        super(cause);  
    }  
}
```

The example above overrides all the public constructors from `Throwable`, but you could override only some of them.

Note that the above example creates a **checked** exception because we are extending `Exception`. To create an **unchecked** exception, you would need to extend `RuntimeException`:

```
public class MyException extends RuntimeException {  
    public MyException() {  
    }  
  
    public MyException(final String message, final Throwable  
cause) {  
        super(message, cause);  
    }  
  
}
```

## ## Customizing Exception messages

If you are just looking for custom messages, as shown in the constructors above, you could use a predefined exception from the Java library and leverage the String `message` parameter.

For example:

```
if(filepath.contains(" ")) {  
    throw new RuntimeException("Provided filepath contains  
space(s)");  
}
```

In cases where you just want to customize the message, you do **not** need to make an entire Exception class and doing so will hurt your code readability.

## Example: Adding ErrorCodeS

One use case that might necessitate the creation of custom exceptions is to provide more localized error codes for runtime exceptions that enable users or support staff to quickly search documentation.

For example:

```
public class MyUncheckedException extends RuntimeException {  
  
    private final ErrorCode code;  
  
    public MyUncheckedException(ErrorCode code) {  
        super();  
        this.code = code;  
    }  
  
    public MyUncheckedException(String message, Throwable cause,  
        ErrorCode code) {  
        super(message, cause);  
        this.code = code;  
    }  
  
    public ErrorCode getCode() {  
        return this.code;  
    }  
}
```

# **Learning Objectives**

- Explain the concept of exception chaining
- Understand the significance of exception chaining in enhancing error traceability and debugging effectiveness
- Understand the implications of using exceptions for flow control on code readability and program performance

# Exception Chaining and Wrapping

## What is Exception Chaining?

Exception chaining in Java is a technique that allows developers to link exceptions together. This process helps in preserving the original exception details when a new exception is thrown, making it easier to trace back to the initial error cause. It's particularly useful in scenarios where exceptions are caught and rethrown as different types.

## Why Use Exception Chaining?

- **Preserves Root Cause Information:** Maintains the stack trace and message of the original exception.
- **Enhances Debuggability:** Provides a complete error trace that aids in debugging.
- **Improves Error Reporting:** Allows developers to throw more specific exceptions while retaining the context of the original error.

Java supports exception chaining through constructors in the `Throwable` class that accept a cause:

```
public class MyCustomException extends Exception {  
    public MyCustomException(String message, Throwable cause) {  
        super(message, cause);  
    }  
}
```

## Scenario

Imagine we have a method responsible for loading configuration settings from a file. There's a potential for a `FileNotFoundException` to be thrown if the file doesn't exist. Instead of allowing this exception to propagate up the call stack, we can catch it and throw a more specific `ConfigurationLoadException` that better describes the context of the error.

### *Implementation*

First, define the custom exception class, `ConfigurationLoadException`, which includes a constructor for exception chaining:

```
public class ConfigurationLoadException extends Exception {  
    public ConfigurationLoadException(String message, Throwable  
cause) {  
        super(message, cause);  
    }  
}
```

### ***Implementation that uses exception chaining:***

```
public void loadConfiguration(String filePath) throws  
ConfigurationLoadException {  
    try {  
        // Simulate attempting to load a configuration file  
        throw new FileNotFoundException("Configuration file not  
found at " + filePath);  
    } catch (FileNotFoundException e) {  
        // Chain the FileNotFoundException within a  
ConfigurationLoadException  
        throw new ConfigurationLoadException("Failed to load  
configuration settings", e);  
    }  
}
```

In this example, when `FileNotFoundException` is caught, it is wrapped within a `ConfigurationLoadException` and rethrown. This way, the original exception `FileNotFoundException` is preserved as the cause of the `ConfigurationLoadException`, maintaining the full stack trace and error message for debugging purposes.

#### **Why This Matters:**

By chaining exceptions, developers ensure that errors are not only more descriptive but also retain all the debugging information from the original exception. This approach makes diagnosing problems significantly easier, as the complete context of the error is maintained.

# Controlling Program Flow with Exceptions

## Steering Your Code with Exceptions

Throwing exceptions in Java isn't just about signaling that something's gone wrong. It can also be a savvy way to steer your program's execution down the right path, especially when you hit a snag that your usual control flow tools can't handle. This section will unpack how to smartly use exceptions for directing your program, when it makes sense to do so, and when it might just complicate things.

### When do you use exceptions for flow control?

Exceptions are ideal for managing situations where an error is severe enough that the normal flow of the program cannot continue. For example, if a critical resource like a database connection fails during an application startup, throwing and handling an exception can allow the application to exit gracefully or switch to a backup system.

Let's look at an example on the use of exceptions for controlling program flow, such as validating user input or handling external resource availability.

```
try {
    int userInput = getUserInput();
    if (userInput < 0) {
        throw new IllegalArgumentException("Input must be non-negative.");
    }
    processInput(userInput);
} catch (IllegalArgumentException e) {
    System.out.println("Error: " + e.getMessage());
    // Redirect user to input prompt again
}
```

### Steps and Logic

**Try Block:** Encapsulates the primary logic that may throw exceptions.

### **Get User Input:**

```
int userInput = getUserInput();
```

Retrieves user input assumed to be an integer. This method should handle user interactions and safely convert input to an integer.

### **Validate Input:**

```
if (userInput < 0) { throw new IllegalArgumentException("Input must be non-negative."); }
```

Checks if the input is less than zero. If true, throws an `IllegalArgumentException` with a message explaining the reason. This stops further processing by exiting the try block.

### **Process Input:**

```
processInput(userInput);
```

Called if the input is validated (non-negative). Contains logic for processing the valid input.

### **Catch Block:**

```
catch IllegalArgumentException e { System.out.println("Error: " + e.getMessage()); }
```

Catches the `IllegalArgumentException` from invalid input. Prints the error message derived from the exception.

### **Error Handling and User Feedback:**

The catch block can redirect the user back to the input prompt for another attempt at valid input. This step is not shown in the code but typically involves looping back in the user interface.

## **Conclusion**

Using exceptions for flow control is a powerful technique in Java, but it should be employed judiciously. As demonstrated, exceptions can neatly handle unexpected conditions, especially when regular control structures might not suffice. However, overuse or misuse of this strategy can lead to code that is hard to understand and maintain. It is essential to balance clarity and the exceptional nature of exceptions to maintain the overall health of your application codebase.

Remember, exceptions are costly in terms of system resources and should not replace simple conditional checks or loops. They are best reserved for truly exceptional conditions that fall outside the normal functioning of the application.

## **Best Practices Recap**

- Use exceptions for flow control only when absolutely necessary and when other control structures are inadequate.

- Always ensure that exceptions used in flow control are documented clearly, both in the code and in any technical documentation.
- Regularly review your use of exceptions to ensure they are used appropriately and do not obscure logic.

## Next Steps

Consider exploring more advanced topics in exception handling, such as creating custom exceptions, managing exception hierarchies, and understanding the performance implications of exceptions in large-scale applications. Each of these areas offers deeper insights into effective error handling and software reliability.

Feel encouraged to implement the principles discussed here in your coding practices and continuously refine your approach as you gain more experience and feedback.

# Passing Exceptions: throws and throw

For checked exceptions, you can either handle them or pass them to the calling method.

There are two ways to pass an exception: `throws` and `throw`.

## throws

`throws` is used at the end of a method header to indicate that the method could throw an exception:

```
public String readFile(final String location) throws IOException
{
    // code
}
```

In the example above, we would say `readFile` throws `IOException`.

Your method can `throws` several exceptions by separating them with a comma:

```
public String readFile(final String location) throws
    InvalidPathException, IOException {
    // code
}
```

The exceptions that are thrown are then passed back to the calling method to be either thrown or handled.

challenge

## Try it out

See how the compiler reports **checked**, un-thrown exceptions by clicking the button below:

Let's take care of the IOException. In the code file to the left:

1. Have `readFile` throw `IOException`
1. The exception above is now passed to `main`, so it needs to be thrown there as well

The compiler is happy with the code now. Try throwing an **unchecked** `InvalidPathException`.

### ▼ Note

Unlike `IOException`, `InvalidPathException` does not need to be declared or explicitly handled, demonstrating the difference between checked and unchecked exceptions

## throw

In some situations, you want to explicitly check for an exception – possibly an unchecked exception. The `throw` keyword allows you to generate custom exceptions:

```
if (location == null || location.isEmpty()) {  
    throw new IllegalArgumentException("Location is empty or  
        null");  
}
```

The example above checks the `location` variable and throws an exception if it is invalid.

Notice that because we are creating the Exception with the `throw` keyword we can pass an optional `message` string to provide custom error messages - in this case explaining which argument is illegal and why.

challenge

## Try it out

Try running the code on the left and notice you get a very long `NoSuchFileNotFoundException` which prints the entire call stack:

throw a new `NoSuchFileNotFoundException` inside the provided `if` statement and run the code again to see how it shortens your output:

Notice that the program is passing the `NoSuchFileNotFoundException` to `main` but it is not handled or passed there. Update `main` to throw all subclasses of `Exception`.

Also, fix the file name to be “hello.txt” to see the program work:

### ▼ Hint

- **Verify File Presence:** Always check if the file exists using `Files.isRegularFile(filepath)` before attempting to read it. If it doesn't exist, throw a `NoSuchFileNotFoundException`.
- **Catch Exceptions:** Use a try-catch block in the `main` method to handle all potential exceptions, ensuring your program can gracefully handle errors without crashing.
- **Test Different Scenarios:** Test your code with both existing and non-existing file paths to understand how your exception handling responds to different situations.

# **Learning Objectives**

**Learners will be able to...**

- **Understand and apply the principles of exception-safe design to enhance system reliability and integrity**
- **Correctly sequence Java code snippets that implement file reading using try-with-resources and handle exceptions**
- **Understand and apply the principles of exception-safe system design**
- **Implement and manage nested exceptions**

# Designing Robust Systems with Exceptions

We will be diving into the practical applications of exception handling in Java. Let's focus on crafting robust systems. This lesson underscores the strategic use of exceptions to navigate errors and uphold system stability.

## The Integral Role of Exceptions in System Design

Exceptions play a crucial role in system design, acting as critical indicators of error conditions. Proper management of exceptions is key to ensuring system resilience and maintaining normal operation even under adverse conditions.

### Exception-Conscious Design Strategies

- Proactive Failure Anticipation:** Construct system components with the foresight of potential failures, employing exceptions to flag these scenarios.
- Judicious Exception Utilization:** Employ exceptions sparingly, reserving them for genuinely exceptional situations beyond local resolution scopes

## Core Principles for Exception-Safe Design

1. **Fail-Safe State:** Systems should be designed to enter a safe state when an exception occurs, allowing for recovery or a controlled shutdown.
2. **Transactional Integrity:** Implement a transactional approach where operations are *atomic* or *all-or-nothing*. This means they either complete fully or roll back entirely, with exceptions used to signal any interruption.
3. **Diligent Resource Management:** Use techniques like `try-with-resources` to manage resources, ensuring that all resources are released properly when an exception is thrown.

The `try-with-resources` statement is a Java feature that simplifies the management of resources such as files, sockets, or database connections, which need to be closed after operations are completed. It ensures that each resource is closed at the end of the statement, which helps in avoiding resource leaks and makes your exception handling code cleaner and more reliable.

## Key Features of try-with-resources

**Automatic Resource Management:** Resources declared within the `try` block are automatically closed after the block is executed, regardless of whether an exception is thrown or not.

**Reduces Boilerplate Code:** Eliminates the need for a `finally` block just to close resources.

**Exception Suppression:** If both the `try` block and `try-with-resources` statements throw exceptions, the exception from the `try` block is propagated, and the exception from the `try-with-resources` block is suppressed.

Let's look at an example of `try-with-resources`.

Suppose you need to read a file and process its content. Using `try-with-resources`, you can ensure that the file is closed after you're done with it, whether the operations succeed or fail due to an exception.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadFileExample {
    public static void main(String[] args) {
        String path = "path/to/your/file.txt";

        // Using try-with-resources to ensure the file reader is
        // closed after use
        try (BufferedReader reader = new BufferedReader(new
FileReader(path))) {
            String line = null;
            while ((line = reader.readLine()) != null) {
                // Process each line read from the file
                System.out.println(line);
            }
        } catch (IOException e) {
            // Handle possible I/O errors
            System.err.println("Error reading from file: " +
e.getMessage());
        }
    }
}
```

The example above demonstrates efficient file reading, resource management, and robust error handling in Java. When you run this program with a valid file path, you will see each line of the file printed to

the console. For example, if the file `file.txt` contains “Hello, World!”, “Welcome to Java programming.”, and “This is an example file.”, these lines will appear in the console output.

If an error occurs, such as the file not being found, an error message will be displayed. You should pay attention to the import statements for `BufferedReader`, `FileReader`, and `IOException`, and ensure the file path is correct. The `try-with-resources` statement is crucial for automatically closing the `BufferedReader`, preventing resource leaks. The `while` loop reads each line until the end of the file, printing each line to the console. Error handling with `IOException` provides useful feedback for debugging.

## Exception Handling at System Interfaces

### Example: Transactional Banking Operations

Consider a banking system that employs transactional integrity in its operations, ensuring all transactions are atomic. You need to implement a `BankAccount` class that handles deposits and withdrawals, ensuring exceptions are thrown for invalid operations.

#### Expected Behavior:

If you attempt to withdraw more money than is available in the account, the `withdraw` method should throw an `IllegalArgumentException` with the message “Insufficient funds”.

If you attempt to deposit a non-positive amount, the `deposit` method should throw an `IllegalArgumentException` with the message “Invalid deposit amount”.

#### Implementation:

##### 1. Implement the Constructor:

You need to implement the constructor

`public BankAccount(double initialBalance)` to initialize the balance of the bank account.

```
java public BankAccount(double initialBalance) {      this.balance =  
initialBalance; }
```

##### 2. Implement Withdraw Method:

Implement the `withdraw` method to decrease the balance by the specified amount. Throw an `IllegalArgumentException` if the amount is greater than the current balance.

```
public void withdraw(double amount) throws  
    IllegalArgumentException {  
    if (balance < amount) throw new  
        IllegalArgumentException("Insufficient funds");  
    balance -= amount; // Adjust balance only if no exception  
                      // is thrown  
}
```

### 3. Implement Deposit Method:

Implement the deposit method to increase the balance by the specified amount. Throw an `IllegalArgumentException` if the amount is less than or equal to zero.

```
public void deposit(double amount) throws  
    IllegalArgumentException {  
    if (amount <= 0) throw new IllegalArgumentException("Invalid  
        deposit amount");  
    balance += amount; // Proceed with deposit, no exceptions  
                      // expected here  
}
```

### 4. Get Balance Method:

Implement the `getBalance` method to return the current balance of the bank account.

```
public double getBalance() {  
    return balance;  
}
```

The code in `BankAccount.java` should look familiar to the code below:

```

public BankAccount(double initialBalance) {
    this.balance = initialBalance;
}

public void withdraw(double amount) throws
    IllegalArgumentException {
    if (balance < amount) throw new
        IllegalArgumentException("Insufficient funds");
    balance -= amount; // Adjust balance only if no
    exception is thrown
}

public void deposit(double amount) throws
    IllegalArgumentException {
    if (amount <= 0) throw new
        IllegalArgumentException("Invalid deposit amount");
    balance += amount; // Proceed with deposit, no
    exception expected here
}

public double getBalance() {
    return balance;
}

```

Add the following to Main.java , then try the code:

```

BankAccount userAccount = new BankAccount(500); // 
Start with an initial balance

try {
    userAccount.deposit(500);
    userAccount.withdraw(1200); // This should trigger
    an exception
} catch (IllegalArgumentException e) {
    System.err.println("Transaction failed: " +
    e.getMessage());
}

System.out.println("Current balance: $" +
userAccount.getBalance());
}
}

```

You should expect to see the following output:

```

Transaction failed: Insufficient funds
Current balance: $1000

```

The initial account balance was 500, then a deposit of 500 was made which totals 1000. However, a withdrawal of 1200 was attempted which resulted in the exception.

This program demonstrates how the banking operations utilize exceptions to maintain the integrity of account transactions, signaling errors like insufficient funds effectively.

#### #### Conclusion

The art of building robust systems with Java exceptions lies in meticulous error signaling, management, and recovery planning. Adhering to the principles of exception-safe design fosters the creation of systems that not only endure but also gracefully navigate the complexities of real-world operations.

# Real-World Applications of Exception Handling

This lesson delves into the practical deployment of exception handling in Java, showcasing its pivotal role in crafting resilient and user-centric software. Understanding these real-world applications illuminates the versatility and indispensability of exception handling in modern software development.

## Essential Setup

Your Java development environment should be equipped with the standard Java libraries. You will be working with the following imports:

```
import java.nio.file.Files;  
import java.nio.file.Paths;  
import java.io.IOException;  
import java.util.stream.Stream;
```

## File Operations and Exception Handling

File Operations in Java frequently leverage exception handling to address issues like missing files, permission errors, or read/write failures, ensuring robust data management.

### Illustration: Secure File Reading

Add the code below into the file:

```

public void secureFileRead(String filePath) {
    // Use try-with-resources to ensure the BufferedReader is
    // closed automatically
    try (BufferedReader reader = new BufferedReader(new
        FileReader(filePath))) {
        String content;
        // Read the file line by line
        while ((content = reader.readLine()) != null) {
            // Print each line to the console
            System.out.println(content);
        }
    } catch (FileNotFoundException e) {
        // Handle the case where the file is not found
        System.err.println("File not accessible: " + filePath);
        // Wrap and rethrow the exception as a RuntimeException
        throw new RuntimeException("File not accessible", e);
    } catch (IOException e) {
        // Handle other I/O errors that may occur during reading
        System.err.println("Error reading file: " + filePath);
        // Wrap and rethrow the exception as a RuntimeException
        throw new RuntimeException("Error reading file", e);
    }
}

```

## Exception Handling in Network Operations

Network programming in Java extensively employs exception handling to navigate connectivity challenges, timeouts, or protocol discrepancies, enhancing communication reliability.

### Scenario: Network Connection Establishment

```

public void establishNetworkConnection(String serverUrl) {
    boolean isConnected = false; // Simulated connection status
    if (!isConnected) {
        throw new RuntimeException("Connection to " + serverUrl
+ " failed");
    }
}

```

This method, `establishNetworkConnection`, demonstrates the typical use of exception handling in network programming in Java. The method simulates an attempt to establish a network connection to a specified server URL. It uses a boolean variable, `isConnected`, to represent the success or failure of the connection attempt. If `isConnected` is false, which it is statically set to in this example, the method throws a `RuntimeException` with a message indicating the failure of the connection.

## User Input Validation via Exceptions

Utilizing exceptions for user input validation in Java streamlines error handling, maintaining code clarity and conciseness.

### Example: Age Verification

```
public int verifyUserAge() {  
    Scanner scanner = new Scanner(System.in);  
    System.out.print("Enter your age: ");  
    int age = scanner.nextInt();  
    if (age < 0) {  
        scanner.close();  
        throw new IllegalArgumentException("Invalid age: Age  
cannot be negative.");  
    }  
    scanner.close();  
    return age;  
}
```

**Expected Result:** Negative age inputs trigger an exception, facilitating immediate and centralized error handling.

## GUI Applications and Exception Management

In Java GUI applications, exception handling proves invaluable in managing errors during event-driven operations like button clicks, ensuring uninterrupted application functionality.

### Handling GUI Events

Consider a Java Swing application where actions tied to button clicks are prone to errors. Implementing exception handling within event handlers allows for the seamless presentation of error dialogs, preserving user experience.

```
java-hide-clipboardpublic void actionPerformed(ActionEvent e) {  
    try {          // Code that might throw an exception      } catch  
(Exception ex) {          JOptionPane.showMessageDialog(null, "Error  
occurred: " + ex.getMessage());      } }
```

This adapted content leverages Java's capabilities and features to demonstrate how exception handling is implemented in various scenarios such as file operations, network connectivity, user input validation, and GUI event management. This practical approach helps Java developers understand the critical role of exceptions in building robust, user-friendly software systems.

# Limitations and Best Practices

## Exception Handling Best Practices in Java

This lesson explores the nuanced landscape of Java exception handling, highlighting its limitations while offering best practices to harness its full potential. A thorough understanding of these elements is essential for crafting clean, efficient, and maintainable Java code.

### Recognizing the Limitations

Exception handling in Java is not without its challenges:

1. **Performance Considerations:** Exception handling mechanisms can incur runtime overhead, particularly when exceptions are frequently used or in performance-sensitive contexts.
2. **Control Flow Complexity:** Over-reliance on exceptions can obscure the program's control flow, complicating debugging and maintenance efforts.
3. **Silent Error Swallowing:** Improperly managed exceptions may be caught and suppressed without adequate handling, leading to elusive bugs.

### Best Practices for Mastery

To effectively leverage exception handling in Java, consider the following best practices:

#### Reserve Exceptions for the Truly Exceptional

Utilize exceptions for abnormal, non-routine conditions. Employing exceptions as a substitute for regular control flow mechanisms can degrade performance and clarity.

##### #### Localize Exception Handling

Aim to manage exceptions close to their origin. This strategy enhances code readability and minimizes the complexity associated with exception propagation.

##### #### Embrace Try-With-Resources for Resource Management

Adopt the `try-with-resources` statement to ensure deterministic resource management, particularly in the face of exceptions, thereby preventing resource leaks.

##### #### Specify Catch Blocks

Opt for catching specific exceptions over generic catch-all blocks `catch (Exception e)`. This targeted approach facilitates precise error handling

and enriches error reporting.

#### #### Avoid Throwing Exceptions from Destructors

Avoid throwing exceptions from `finalize` methods, as this can lead to unpredictable behavior and resource leaks. Instead, use `try-finally` blocks for cleanup.

#### #### Document Exception Safety Levels

Clearly articulate the exception safety guarantees (no-throw, strong, basic, or none) provided by your methods. This transparency aids consumers of your code in crafting robust exception handling strategies.

#### #### Align with Standard Exceptions

Whenever feasible, use the standard exception classes from the Java Standard Library or derive custom exceptions from them. This alignment ensures compatibility with the broader Java exception handling ecosystem.

#### #### Rigorously Test Exception Paths

Thoroughly test your application's behavior in response to exceptions, ensuring that all resources are correctly released and that the application provides coherent error feedback.

#### ## Wrapping Up

While exception handling is an indispensable facet of Java programming, it demands careful consideration to avoid its pitfalls. By adhering to established best practices, you can effectively navigate its limitations, employing exceptions to enhance the reliability, maintainability, and efficiency of your Java applications. The objective is to leverage exception handling to simplify your codebase, not to complicate it further.

# **Advanced Topics in Exception Handling**

In this lesson, we explore advanced topics that enhance the scope and efficacy of exception handling in Java. These concepts are crucial for navigating complex scenarios, thereby increasing the robustness and maintainability of Java applications.

## **Nested Exceptions in Java**

Java supports nested exceptions, which allows an exception to encapsulate another. This feature preserves the context and details of the initial error, providing a deeper understanding of exceptions when they occur.

### **Example: Demonstrating Nested Exceptions**

This code is already in the file.

```

public class ExceptionDemo {

    public static void triggerNestedException() throws Exception
    {
        try {
            // Attempt to throw a primary exception
            throw new RuntimeException("Primary exception");
        } catch (RuntimeException e) {
            // Nest it within a secondary exception
            throw new Exception("Secondary exception", e);
        }
    }

    public static void manageNestedException() {
        try {
            // Call function that may throw a nested exception
            triggerNestedException();
        } catch (Exception e) {
            // Catch the outer exception and display its message
            System.err.println("Outer Exception: " +
e.getMessage());
            if (e.getCause() != null) {
                // Display the nested exception's message
                System.err.println("Inner Exception: " +
e.getCause().getMessage());
            }
        }
    }

    public static void main(String[] args) {
        manageNestedException();
    }
}

```

When the program is run, it should produce an output that looks similar to this:

```

Outer Exception: Secondary exception
Inner Exception: Primary exception

```

By understanding and running this code, you will learn how to effectively manage and debug nested exceptions in Java, an essential skill for developing robust and maintainable applications.

*Conclusion*

Overall, this deep dive into advanced exception handling equips you with a robust set of strategies for managing complex error scenarios in Java. By leveraging these techniques, you can significantly improve the resilience, clarity, and maintainability of your Java applications. These enhancements prepare your applications to meet the demands of complex and critical operating environments, ensuring they perform reliably under various conditions.

# Lab 1 : Simple File Reader

## Instructions:

In this lab, you will develop a Java application to read a series of text lines from a file. Your program will need to handle exceptions gracefully, such as file not found errors and I/O errors, ensuring the system remains robust and user-friendly.

### 1. File Reading:

- Implement the `readFileContent` method to open and read a specified text file. Use exception handling to catch and report file access errors.
- Use the `try-with-resources` statement to ensure the file stream is closed properly.

### 2. Error Handling:

- In your `readFileContent` method, catch the `FileNotFoundException` to handle the scenario where the file does not exist.
- Catch `IOException` for other types of I/O errors.

### 3. User Feedback:

- Provide clear console output for any errors encountered, ensuring the program does not terminate unexpectedly. Display a friendly error message to inform the user of what went wrong.

### 4. Logging:

- Extend your application to log any errors to the console. This includes both file not found and general I/O errors.

## Submission Requirements:

- Submit your modified `FileReaderApp.java` file.
- Ensure your program compiles and runs without errors.
- Your program should handle all specified error conditions and provide appropriate user feedback.

# Lab 2: Database Connection Simulator

In this lab you will develop a Java application that simulates connecting to and querying a database. The focus will be on using exceptions to manage potential errors that could occur during database operations, such as connection failures or query issues.

## Instructions:

1. **Simulate Database Connections:**
  - Use exception handling to manage a custom exception, `DatabaseConnectionException`, when the connection fails.
2. **Query Simulation:**
  - Use exception handling to manage a custom exception, `QueryExecutionException`, for handling query-related errors.
3. **Transaction Management:**
  - Create a method `processTransaction` that ensures all database operations either complete successfully or the transaction is rolled back using custom exception handling.
  - Simulate a transaction scenario where multiple queries are executed as part of a single transaction.
4. **User Feedback and Logging:**
  - Provide clear console output for any errors encountered, ensuring the program does not terminate unexpectedly.
  - Extend your application to log both successful operations and errors.

## Submission Requirements:

- Submit your modified `DatabaseApp.java` file.
- Ensure your program compiles and runs without errors.
- Your program should handle all specified error conditions, log operations as described, and provide appropriate user feedback.

# Lab 3: Network Operations Simulator

In this lab, students will develop a Java application that simulates sending and receiving data over a network. The focus will be on using exceptions to manage potential errors that could occur during network communication, such as connection timeouts or data format issues.

## Instructions:

1. **Setup Network Communication:**
  - Use exception handling to manage a custom exception, `NetworkConnectionException`, when the connection attempt fails.
2. **Data Transmission:**
  - Use exception handling to manage a custom exception, `DataTransmissionException`, for handling data transmission errors.
3. **Error Recovery:**
  - Create methods `retryConnection` and `retryDataTransmission` that attempt to re-establish a connection or resend data if the first attempt fails.
  - Implement retry logic that attempts to reconnect or resend data up to a maximum number of retries before failing permanently.
4. **Logging and User Feedback:**
  - Provide clear console output for any errors encountered as well as successful connections and transmissions, ensuring the program does not terminate unexpectedly.
  - Extend your application to log both successful operations and errors to a log file or console.

## Submission Requirements:

- Submit your modified `NetworkSimulator.java` file.
- Ensure your program compiles and runs without errors.
- Your program should handle all specified error conditions, log operations as described, and provide appropriate user feedback.

# Lab 4: GUI Error Management

In this lab, students will develop a Java application that performs asynchronous operations, such as handling background tasks or processing events. The focus will be on managing exceptions that occur in asynchronous programming contexts, such as threads or runnable tasks.

## Instructions:

1. **Create a Simple GUI:**
  - Implement a basic GUI using Java Swing that includes user input fields (e.g., text fields for entering numbers) and buttons to perform operations (e.g., calculate something based on input).
  - Design the layout to be straightforward and user-friendly.
2. **Implement Exception Handling:**
  - Add error handling to manage exceptions that might occur during operations (e.g., `NumberFormatException` when parsing user input).
  - Use dialog boxes to display error messages effectively when exceptions occur.
3. **Enhance User Feedback:**
  - Ensure that the GUI provides immediate, understandable feedback for errors through dialog boxes or updating the GUI elements (e.g., error messages displayed near input fields).
  - Validate user inputs and provide feedback before processing any operations to prevent exceptions where possible.
4. **Logging:**
  - Implement logging to record user actions and any exceptions that occur. This can help in debugging and understanding user behavior.

## Submission Requirements:

- Submit your modified `GuiApp.java` file.
- Ensure your program compiles and runs without errors.
- Your program should handle all specified error conditions, provide appropriate user feedback, and log relevant actions and errors.

# Lab 5: Exception Handling in File Operations

## Lab 5: Exception Handling in File Operations

This lab focuses on mastering file I/O operations in Java with an emphasis on robust exception handling to manage potential errors during file access and manipulation tasks.

### Instructions

#### *API Setup:*

##### **1. Setup Basic File I/O Operations:**

- You will use Java’s `BufferedReader` and `FileWriter` classes to implement file reading and writing functionalities.

#### *Simulating Errors:*

##### **2. Implement File Reading with Exception Handling:**

- The provided code already includes methods to read from a file.
- Ensure that all possible I/O exceptions are handled gracefully.

#### *Handling Client Errors:*

##### **3. Write to a File with Custom Error Handling:**

- Implement the `writeFile` method that writes a specified string to a file.
- Handle any exceptions that might occur during the file writing process, providing clear error messages to the console.

#### *Global Exception Handling:*

##### **4. Implement Robust Cleanup:**

- Use the `finally` block or `try-with-resources` statement to ensure that all resources are closed properly to avoid resource leaks.

#### *Testing and Documentation:*

**5. Document the Functionality:**

- Document how each part of your file I/O operations responds to different types of file access issues, specifying how exceptions are handled and resources are managed.

***Submission Requirements:***

- Ensure your program compiles and runs without errors.
- Your implementation should handle all specified error conditions and provide appropriate feedback and cleanup.

# **Learning Objectives - Classes and Objects**

---

- Define the terms class, objects, instance, and instantiate
- Identify the difference between classes and objects
- Create a user-defined object
- Add an attribute to an object with dot notation
- Define class attribute
- Explain the difference between shallow and deep copies

# Built-In Objects

---

## The String Object

You have already been using built-in Java objects. Strings are an example of a Java object.

```
//add code below this line

String s = new String("I am a string");
System.out.println(s.getClass());

//add code above this line
```

challenge

### Try these variations:

Explore some of the methods associated with the string class.

- \* Add the line of code `System.out.println(s.isEmpty());`
- \* Add the line of code `System.out.println(s.getBytes());`
- \* Add the line of code `System.out.println(s.endsWith("g"));`

Java says that the class of `s` is `java.lang.String` (which is a string). Add the following code and run the program again.

```
//add code below this line

String s = new String("I am a string");
System.out.println(s.getClass());

Method[] methods = s.getClass().getDeclaredMethods();
for (Method m : methods) {
    System.out.println("Method name: " + m + "\n");
}

//add code above this line
```

The variable `methods` is an array of all methods associated with the `String` class. If you look carefully at the output, you may be confused by the information on the screen. However, a few things, like `toUpperCase` and `toLowerCase`, may seem familiar. Methods will be covered in a later lesson, but it is important to understand that a string is not a simple collection of characters. Because a string is a class, it is a powerful way of collecting and modifying data.

## Vocabulary

In the text above, the words “class” and “object” are used in an almost interchangeable manner. There are many similarities between classes and objects, but there is also an important difference. Working with objects has a lot of specialized vocabulary.

**Classes** - Classes are a collection of data and the actions that can modify the data. Programming is a very abstract task. Classes were created to give users a mental model of how to think about data in a more concrete way. Classes act as the blueprint. They tell Java what data is collected and how it can be modified.

**Objects** - Objects are constructed according to the blueprint that is the class. In the code above, the variable `s` is a string object. It is not the class. The string class tells Java that `s` has methods like `length`, `concat`, and `replace`. When a programmer wants to use a class, they create an object.

**Instance** - Another way that programmers talk about objects is to say that an object is an instance of a particular class. For example, `s` is an instance of the `String` class.

**Instantiation** - Instantiation is the process where an object is created according to blueprint of the class. The phrase “define a variable” means to create a variable. The variable is given a name and a value. Once it has been defined, you can use the variable. With objects, you use the phrase “instantiate an object”. That means to create an object, give it a name, store any data, and define the actions the object can perform.

The diagram shows a snippet of Java code: `String s = new String("I am a string"); s.toUpperCase();`. A red bracket labeled "Instantiate s as an instance (object) of the string class" points to the line `s = new String("I am a string");`. A blue bracket labeled "The string class is the blueprint that gives objects the ability to store a series of characters and have an action to turn the characters to uppercase" points to the line `s.toUpperCase();`. A blue icon with a question mark and the letter "A" is positioned next to the blue bracket.

```
String s = new String("I am a string");
s.toUpperCase();
```

Class vs Object

# User-Defined Objects

---

## Defining an Object

Assume you want to collect information about actors. Creating a class is a good way to keep this data organized. The `class` keyword are used to define a class. For now, do not add anything as the body of the class.

```
//add class definitions below this line

class Actor {

}

//add class definitions above this line
```

### ▼ Naming classes

The convention for naming classes in Java is to use a capital letter. A lowercase letter will not cause an error message, but it is not considered to be “correct”. If a class has a name with multiple words, all of the words are pushed together, and a capital letter is used for the first letter of each word. This is called camel case.

Classes are just the blueprint. To you use a class, you need to instantiate an object. Here is an object to represent Helen Mirren. Be sure to put this code in the `main` method.

```
//add code below this line

Actor helen = new Actor();
System.out.println(helen.getClass());

//add code above this line
```

So you now have `helen`, which is an instance of the `Actor` class.

## Adding Attributes

The point of having a class is to collect information and define actions that can modify the data. The Actor class should contain things like the name of the actor, notable films, awards they have won, etc. These pieces of information related to a class are called attributes. Attributes are declared in the class itself. The example below adds the `firstName` and `lastName` attributes which are both strings.

```
//add class definitions below this line
```

```
class Actor {  
    String firstName;  
    String lastName;  
}
```

```
//add class definitions above this line
```

You can change the value of an attribute with the assignment operator, `objectName.attribute = attributeName`. Notice that you always use `objectName.attribute` to reference an attribute. This is called dot notation. Once an attribute has a value, you can treat it like any other variable. Add the following code to the `main` method. You are assigning values to the attributes `firstName` and `lastName`, and then printing these values.

```
//add code below this line
```

```
Actor helen = new Actor();  
helen.firstName = "Helen";  
helen.lastName = "Mirren";  
System.out.println(helen.firstName + " " + helen.lastName);
```

```
//add code above this line
```

challenge

## Try these variations:

- Change the print statement to:

```
System.out.println(helen.firstName.toUpperCase() + " " +  
helen.lastName.toLowerCase());
```

- Add the attribute `totalFilms` and assign it the value `80`
- Add the print statement `System.out.println(helen);`

## ▼ Code

You may have noticed that printing the object `helen` returns `Actor@` followed a series of numbers and letters. Java is telling you that `helen` is an object of class `Actor` and the numbers and letters represent the object's location in memory.

```
import java.lang.Class;

//add class definitions below this line

class Actor {
    String firstName;
    String lastName;
    int totalFilms;
}

//add class definitions above this line

public class UserDefined {
    public static void main(String[] args) {

        //add code below this line

        Actor helen = new Actor();
        helen.firstName = "Helen";
        helen.lastName = "Mirren";
        helen.totalFilms = 80;
        System.out.println(helen.firstName.toUpperCase() + " " +
                           helen.lastName.toLowerCase());
        System.out.println(helen.totalFilms);
        System.out.println(helen);

        //add code above this line
    }
}
```

# The Constructor

---

## Too Much Code

Imagine that the Actor class has more attributes than on the previous page.

```
//add class definitions below this line

class Actor {
    String firstName;
    String lastName;
    String birthday;
    int totalFilms;
    int oscarNominations;
    int oscarWins;
}

//add class definitions above this line
```

Now create a object for Helen Mirren with values for each attribute. Adding each attribute individually requires several lines of code. This is especially true if you more than one instance of the Actor class.

```
//add code below this line

Actor helen = new Actor();
helen.firstName = "Helen";
helen.lastName = "Mirren";
helen.birthday = "July 26";
helen.totalFilms = 80;
helen.oscarNominations = 4;
helen.oscarWins = 1;
System.out.println(helen.firstName + " " + helen.lastName);

//add code above this line
```

The class Actor creates a class and its attributes. It does not assign value to any attributes; the user has to do this. A class is suppose to be a blueprint. It should lay out all of the attributes and their values for the user. Classes can do this when you use the constructor.

## The Constructor

The constructor is a special method for a class. Its job is to assign value for attributes associated with the object. These attributes can also be called instance variables. In Java, the constructor is the class name, parentheses, and curly brackets. Inside of the constructor, give attributes their values.

```
//add class definitions below this line

class Actor {
    String firstName;
    String lastName;
    String birthday;
    int totalFilms;
    int oscarNominations;
    int oscarWins;

    Actor() {
        firstName = "Helen";
        lastName = "Mirren";
        birthday = "July 26";
        totalFilms = 80;
        oscarNominations = 4;
        oscarWins = 1;
    }
}

//add class definitions above this line
```

Instantiating `helen` as an instance of the `Actor` class automatically calls the constructor. Since the instance variables (attributes) have values, you can remove those lines of code from the `main` method.

```
//add code below this line

Actor helen = new Actor();
System.out.println(helen.firstName + " " + helen.lastName);

//add code above this line
```

challenge

## Try these variations:

- Add this print statement to the `main` method:

```
System.out.println(helen.firstName + " " + helen.lastName +  
    "'s birthday is " + helen.birthday);
```

- Add this print statement to the `main` method:

```
System.out.println(helen.firstName + " " + helen.lastName +  
    " won " + helen.oscarWins + " Oscar out of " +  
    helen.oscarNominations + " nominations");
```

# The Constructor and Parameters

---

## The Constructor and Parameters

Now imagine that you want to use the Actor class to instantiate an object for Helen Mirren and Tom Hanks. Create the Actor class just as before.

```
//add class definitions below this line

class Actor {
    String firstName;
    String lastName;
    String birthday;
    int totalFilms;
    int oscarNominations;
    int oscarWins;

    Actor() {
        firstName = "Helen";
        lastName = "Mirren";
        birthday = "July 26";
        totalFilms = 80;
        oscarNominations = 4;
        oscarWins = 1;
    }
}

//add class definitions above this line
```

Now instantiate two Actor objects, one for Helen Mirren and the other for Tom Hanks. Print the `firstName` and `lastName` attributes for each object.

```
//add code below this line

Actor helen = new Actor();
Actor tom = new Actor();
System.out.println(helen.firstName + " " + helen.lastName);
System.out.println(tom.firstName + " " + tom.lastName);

//add code above this line
```

The constructor Actor class only creates an object with information about Helen Mirren. You can make the Actor class more flexible by passing it an argument for each of attributes in the constructor. Parameters for the constructor method work just as they do for user-defined methods, be sure to indicate the data type for each parameter.

```
//add class definitions below this line

class Actor {
    String firstName;
    String lastName;
    String birthday;
    int totalFilms;
    int oscarNominations;
    int oscarWins;

    Actor(String fn, String ln, String bd, int tf, int on, int ow)
    {
        firstName = fn;
        lastName = ln;
        birthday = bd;
        totalFilms = tf;
        oscarNominations = on;
        oscarWins = ow;
    }
}

//add class definitions above this line
```

When you instantiate the two Actor objects, you can pass the constructor the relevant information for both Helen Mirren and Tom Hanks. The code should now print the correct first and last names.

```
//add code below this line

Actor helen = new Actor("Helen", "Mirren", "July 26", 80, 4,
1);
Actor tom = new Actor("Tom", "Hanks", "July 9", 76, 5, 2);
System.out.println(helen.firstName + " " + helen.lastName);
System.out.println(tom.firstName + " " + tom.lastName);

//add code above this line
```

challenge

## Try these variations:

- Create an instance of the Actor class for Denzel Washington (December 28, 47 films, 8 nominations, 2 wins)
- Print the `birthday` and `totalFilms` attributes for the newly created object

### ▼ Code

Your code for the object representing Denzel Washington should look something like this:

```
//add code below this line

Actor denzel = new Actor("Denzel", "Washington",
    "December 28", 47, 8, 2);
System.out.println(denzel.birthday);
System.out.println(denzel.totalFilms);

//add code above this line
```

## Default Values

We can assume that the average actor probably has not been nominated or won an Oscar. So instead of making these attributes parameters for the constructor, we can give them the default value of 0. These attributes can always be updated later on.

```
//add class definitions below this line

class Actor {
    String firstName;
    String lastName;
    String birthday;
    int totalFilms;
    int oscarNominations;
    int oscarWins;

    Actor(String fn, String ln, String bd, int tf) {
        firstName = fn;
        lastName = ln;
        birthday = bd;
        totalFilms = tf;
        oscarNominations = 0;
        oscarWins = 0;
    }
}

//add class definitions above this line
```

You can update the attributes once the object has been instantiated if need be.

```
//add code below this line

Actor helen = new Actor("Helen", "Mirren", "July 26", 80);
System.out.println(helen.oscarNominations);
System.out.println(helen.oscarWins);

helen.oscarNominations = 4;
helen.oscarWins = 1;

System.out.println(helen.oscarNominations);
System.out.println(helen.oscarWins);

//add code above this line
```

# Class Attributes

## Class Attributes

Up until now, the attributes created in the class are independent from one another when new objects are created. These are called object attributes. You can create an instance of the Actor class for Helen Mirren and another for Dwayne Johnson. Each object has different values for their attributes. A class attribute, however, is an attribute whose value is shared by each instance of a class.

To illustrate this, add the attribute `static String union` to the Actor class. The `static` keyword tells Java that this attribute is a class attribute. Set the value of `union` to the string "Screen Actors Guild". Notice that class attributes are declared and initialized with a value outside of the constructor.

```
//add class definitions below this line

class Actor {
    String firstName;
    String lastName;
    String birthday;
    int totalFilms;
    int oscarNominations;
    int oscarWins;
    static String union = "Screen Actors Guild";

    Actor(String fn, String ln, String bd, int tf) {
        firstName = fn;
        lastName = ln;
        birthday = bd;
        totalFilms = tf;
        oscarNominations = 0;
        oscarWins = 0;
    }
}

//add class definitions above this line
```

Create two instances of the Actor class and print the `union` attribute for both instances. Then change the attribute for one instances and print the `union` attribute for both instances.

```
//add code below this line

Actor helen = new Actor("Helen", "Mirren", "July 26", 80);
Actor dwayne = new Actor("Dwayne", "Johnson", "May 2", 34);
System.out.println(helen.union);
System.out.println(dwayne.union);
helen.union = "Teamsters";
System.out.println(helen.union);
System.out.println(dwayne.union);

//add code above this line
```

Because this is a class attribute, the value is shared among all the instances. A change for instance, is a change for all instances.

challenge

### Try this variation:

- Remove the keyword `static` and run the code again.

## Class Attributes as Constants

Changing a class attribute can cause problems for other instances of the same class. That is why class attributes are often created as a constant. Remember, the Java convention for constants is to write the variable name in all caps.

```
//add class definitions below this line

class Actor {
    String firstName;
    String lastName;
    String birthday;
    int totalFilms;
    int oscarNominations;
    int oscarWins;
    static final String UNION = "Screen Actors Guild";

    Actor(String fn, String ln, String bd, int tf) {
        firstName = fn;
        lastName = ln;
        birthday = bd;
        totalFilms = tf;
        oscarNominations = 0;
        oscarWins = 0;
    }
}

//add class definitions above this line
```

Java will now prevent users from changing the value of UNION. You should see an error message that Java cannot assign a value to a final variable.

```
//add code below this line

Actor helen = new Actor("Helen", "Mirren", "July 26", 80);
Actor dwayne = new Actor("Dwayne", "Johnson", "May 2", 34);
helen.UNION = "Teamsters";

//add code above this line
```

# Shallow and Deep Copies

---

## Shallow Copies

The code below will instantiate a as an instance of the `ComicBookCharacter` class. Object a will be given a name, an age, and a type. Object b will be a copy of a. Finally, the `name` attribute of object a is changed. What do you expect to see when the `name` attributes of objects a and b are printed?

```
class ComicBookCharacter {
    String name;
    int age;
    String type;
}

//add class definitions above this line

public class Copies {
    public static void main(String[] args) {

        //add code below this line

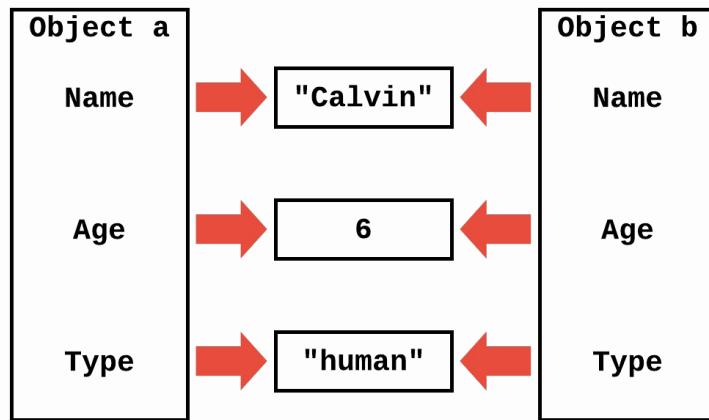
        ComicBookCharacter a = new ComicBookCharacter();
        a.name = "Calvin";
        a.age = 6;
        a.type = "human";

        ComicBookCharacter b = a;
        a.name = "Hobbes";

        System.out.println("Object a name: " + a.name);
        System.out.println("Object b name: " + b.name);

        //add code above this line
    }
}
```

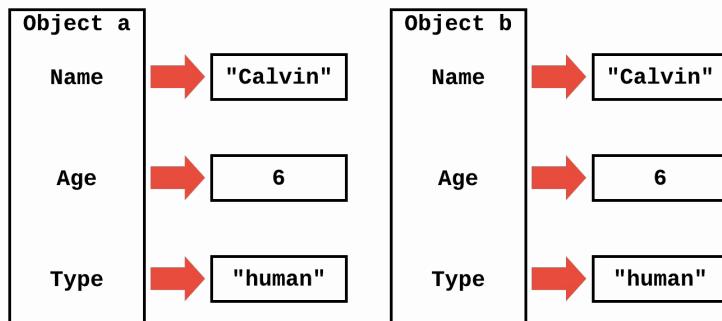
Both of the `name` attributes changed, even though the code only changed the `name` attribute of object a. This is because object b is a shallow copy of object a. Java makes a copy of object a, but object b shares the attributes with object a. That is why changing `name` for object a also affects the `name` attribute for object b.



Shallow Copy

## Deep Copy

A deep copy is when Java makes a copy of object a and makes copies of each attribute. A deep copy keeps the attributes of one object independent of the other object. To create a deep copy, you are going to use something called a copy constructor.



Deep Copy

The copy constructor is a constructor that creates an object by initializing it with a previously created object of the same class. Start by creating another constructor that takes a `ComicBookCharacter` as a parameter. Assign each object attribute with the corresponding attribute from the parameter.

```

//add class definitions below this line

class ComicBookCharacter {
    String name;
    int age;
    String type;

    ComicBookCharacter(String n, int a, String t) {
        name = n;
        age = a;
        type = t;
    }

    ComicBookCharacter(ComicBookCharacter c) {
        name = c.name;
        age = c.age;
        type = c.type;
    }
}

//add class definitions above this line

```

Now instantiate b using the copy constructor. Object b will now be a deep copy of object a, and their attributes are independent from one another.

```

//add code below this line

ComicBookCharacter a = new ComicBookCharacter("Calvin", 6,
    "human");

ComicBookCharacter b = new ComicBookCharacter(a);
a.name = "Hobbes";

System.out.println("Object a name: " + a.name);
System.out.println("Object b name: " + b.name);

//add code above this line

```

# **Introduction to Objects Formative Assessment 1**

---

# **Introduction to Objects Formative Assessment 2**

---

## **Learning Objectives - Mutability**

---

- Define the term **mutable**
- Construct a function to modify instance variables  
**(attributes)**

# Mutability and External Methods

---

## Mutability

Objects are mutable, which means that objects (specifically their attributes) can change value. Think of a video game; the main character in the game is constantly changing. It could be their position on the screen, the score, their health, the items in their inventory, etc. Imagine a simple class called Player. A newly instantiated Player object has a health of 100, a score of 0, and starts on level 1. This object can lose health, increase their score, and advance levels.

```
//add class definitions below this line

class Player {
    int health;
    int score;
    int level;

    Player() {
        health = 100;
        score = 0;
        level = 1;
    }
}

//add class definitions above this line
```

Print out the attributes of player1. Then change each attribute and print out the attributes again.

```
//add code below this line

Player player1 = new Player();
System.out.println("This player has " + player1.health + "
    health, a score of " + player1.score + " and is on level
    " + player1.level + ".");
player1.health -= 10;
player1.score += 25;
player1.level += 1;
System.out.println("This player has " + player1.health + "
    health, a score of " + player1.score + " and is on level
    " + player1.level + ".");

//add code above this line
```

challenge

### Try these variations:

- Change the health of player1 to 0?
- Print the status of player1 once their health is 0?  
 ▼ One Possible Solution

```
System.out.println("This player is dead. They died on
    level " + player1.level + " with a score of " +
    player1.score + ".");
```

## External Methods and Objects

One of the benefits of methods is code reusability. The example above has a repetition of the `System.out.println` statement. This is a good opportunity to use a method.

```
//add method definitions below this line

public static void printPlayer(Player p) {
    System.out.println("This player has " + p.health + " health,
        a score of " + p.score + " and is on level " + p.level +
        ".");
}

//add method definitions above this line
```

In the `main` method, replace the strings inside the print statements with a call to the `printPlayer` method. Don't forget to pass the `player1` object to the `printPlayer` method.

```
//add code below this line

Player player1 = new Player();
printPlayer(player1);
player1.health -= 10;
player1.score += 25;
player1.level += 1;
printPlayer(player1);

//add code above this line
```

Using a method to print the status of `player1` may not seem like it was worth the effort to change the code. But when these methods become more complex, The efficiency becomes clear.

```
//add method definitions below this line

public static void printPlayer(Player p) {
    if (p.health <= 0) {
        System.out.println("This player is dead. They died on
                           level " + p.level + " with a score of " + p.score +
                           ".");
    } else {
        System.out.println("This player has " + p.health + "
                           health, a score of " + p.score + " and is on level " +
                           p.level + ".");
    }
}

//add method definitions above this line
```

Now that the `printPlayer` method will return two different strings, call the method when the `player1` object has full health, and call it again when the object has no health.

```
//add code below this line

Player player1 = new Player();
printPlayer(player1);
player1.health = 0;
player1.score += 25;
player1.level += 1;
printPlayer(player1);

//add code above this line
```

challenge

## Can you:

- Create a function to change a player's health?  
▼ One possible solution

```
public static void changeHealth(Player p, int amount) {
    p.health += amount;
}
```

- Create a function to change a player's level?  
▼ One possible solution

```
public static void changeLevel(Player p) {
    p.level += 1;
}
```

# **Changing Objects with External Methods Formative Assessment 1**

---

# **Changing Objects with External Methods Formative Assessment 2**

---

## **Learning Objectives - Instance Methods**

---

- Define the term **Instance method**
- Convert an **external method** that modifies an object into an **instance method**
- Demonstrate the **syntax for calling a method**

# External Methods vs Class Methods

## Class Methods

Back in the introduction to classes and objects lesson, a class was defined as “a collection of data and the actions that can modify the data.” The constructor built the “collection of data”, but nothing in the class modified the data. Instead, external methods were used to modify the object.

Instead of external methods, instance methods should be used to modify an object. Think of a instance method as a method that is attached to an object. The instance method is the most common type of method when creating classes. Notice how instance methods are declared inside of the class. These methods are called instance methods because they have access to the instance variables (the attributes declared in the constructor). Methods are invoked using dot-notation.

<p><b>External Method</b></p> <pre>class Player {     int health;     int score;     int level;      Player() {         health = 100;         score = 0;         level = 1;     }      public static void changeLevel(Player p) {         p.level += 1;     }      Player mario = new Player();     changeLevel(mario);</pre>	<p><b>Instance Method</b></p> <pre>class Player {     int health;     int score;     int level;      Player() {         health = 100;         score = 0;         level = 1;     }      void changeLevel() {         level += 1;     }      Player mario = new Player();     mario.change_level();</pre>
	

External Methods vs Instance Methods

When mutability was first introduced, you made a `Player` class with a few external methods. You are now going to transform these external methods into instance methods. The `Player` class will be defined just as before. This time, however, `printPlayer` will be a part of the class.

```
//add class definitions below this line

class Player {
    int health;
    int score;
    int level;

    Player() {
        health = 100;
        score = 0;
        level = 1;
    }

    void printPlayer() {
        if (health <= 0) {
            System.out.println("This player is dead. They died on
                level " + level + " with a score of " + score + ".");
        } else {
            System.out.println("This player has " + health + " health,
                a score of " + score + " and is on level " + level +
                ".");
        }
    }
}

//add class definitions above this line
```

Instantiate a Player object. Call the class method printPlayer using dot-notation.

```
//add code below this line

Player mario = new Player();
mario.printPlayer();

//add code above this line
```

challenge

## Try this variation:

Call printPlayer like this:

```
Player mario = new Player();
printPlayer(mario);
```

### ▼ Why did this generate an error?

Java says it cannot find the symbol printPlayer, even though the definition is on line 14. Because nothing comes before printPlayer, Java assumes that this is an external method. However, printPlayer is a part of the Player class, which means it is an instance method.

Instance methods must be called with dot-notation like

```
mario.printPlayer();
```

## More Player Methods

The next instance methods to add to the Player class are those to print the health and level attributes of the Player instance. Start with the instance method changeHealth. This method takes amount as a parameter. changeHealth will add amount to the health attribute. If a player's health increases, amount is positive. If their health decreases, amount is negative.

```
void printPlayer() {
    if (health <= 0) {
        System.out.println("This player is dead. They died on
                           level " + level + " with a score of " + score + ".");
    } else {
        System.out.println("This player has " + health + " health,
                           a score of " + score + " and is on level " + level +
                           ".");
    }
}

void changeHealth(int amount) {
    health += amount;
}
```

The instance method changeLevel is going to be similar to changeHealth except for one difference. changeLevel has no parameters. In video games, players go up in levels; rarely do they decrease. So the level attribute will increase by one when the instance method is called.

```
void changeHealth(int amount) {  
    health += amount;  
}  
  
void changeLevel() {  
    level += 1;  
}
```

challenge

## Try these variations:

- Call `changeHealth` and `chagneLevel` for `mario`, and then print the player to make sure the instance methods work.

▼ One possible solution

```
//add code below this line  
  
Player mario = new Player();  
mario.printPlayer();  
mario.changeHealth(-10);  
mario.changeLevel();  
mario.printPlayer();  
  
//add code above this line
```

- Create an instance method to change a player's score?

▼ One possible solution

```
void changeScore(int amount) {  
    score += amount;  
}
```

▼ Why learn about external methods that modify objects when Java has instance methods?

It might seem like a waste of time to learn how to write external methods that modify objects. But this approach builds upon concepts you have already seen — external methods and objects. This allows you to understand mutability without having to worry about instance methods. Once you understand how these ideas work, transforming an external method into an instance method is much simpler. External methods that

modify objects serve as an intermediary step on the way to learning about instance methods.

# More Class Methods

---

## More on Class Methods and Objects

Changes to objects should happen exclusively through instance methods. This makes your code easier to organize and easier for others to understand. Imagine you are going to create a class that keeps track of a meal. In this case, a meal can be thought of as all of the drinks, appetizers, courses, and desserts served. Each one of these categories will become an instance variable (attribute). Assign each attribute an ArrayList of strings.

```
//add class definitions below this line

class Meal {
    ArrayList<String> drinks = new ArrayList<String>();
    ArrayList<String> appetizers = new ArrayList<String>();
    ArrayList<String> mainCourse = new ArrayList<String>();
    ArrayList<String> dessert = new ArrayList<String>();
}

//add class definitions above this line
```

Next, add an instance method to add a drink to the `Meal` object. Use the `.add` method to add an element to the list. So `drinks.add(d)` adds the drink `d` to the ArrayList `drinks`.

```
//add class definitions below this line

class Meal {
    ArrayList<String> drinks = new ArrayList<String>();
    ArrayList<String> appetizers = new ArrayList<String>();
    ArrayList<String> mainCourse = new ArrayList<String>();
    ArrayList<String> dessert = new ArrayList<String>();

    void addDrink(String d) {
        drinks.add(d);
    }
}

//add class definitions above this line
```

Create a `Meal` object and test your code to make sure it is working as expected.

```
//add code below this line

Meal dinner = new Meal();
dinner.addDrink("water");
System.out.println(dinner.drinks);

//add code above this line
```

Now create the `addAppetizer` instance method for the class. Like the method above, `addAppetizer` accepts a string as a parameter and adds it to the `appetizers` attribute.

```
void addDrink(String d) {
    drinks.add(d);
}

void addAppetizer(String a) {
    appetizers.add(a);
}
```

Add "bruschetta" to the `dinner` object and print it.

```
//add code below this line

Meal dinner = new Meal();
dinner.addDrink("water");
System.out.println(dinner.drinks);
dinner.addAppetizer("bruschetta");
System.out.println(dinner.appetizers);

//add code above this line
```

challenge

## Create the following instance methods:

- `addCourse` - accepts a string which represents a course and adds it to the meal.
- `addDessert` - accepts a string which represents a dessert and adds it to the meal.

Test your code using "roast chicken" as a main course and "chocolate cake" as a dessert. Then print out each course of the meal.

▼ Meal code

```
import java.util.ArrayList;

//add class definitions below this line

class Meal {
    ArrayList<String> drinks = new ArrayList<String>();
    ArrayList<String> appetizers = new ArrayList<String>();
    ArrayList<String> mainCourse = new ArrayList<String>();
    ArrayList<String> dessert = new ArrayList<String>();

    void addDrink(String d) {
        drinks.add(d);
    }

    void addAppetizer(String a) {
        appetizers.add(a);
    }

    void addCourse(String c) {
        mainCourse.add(c);
    }

    void addDessert(String d) {
        dessert.add(d);
    }
}

//add class definitions above this line

public class MoreMethods {
    public static void main(String[] args) {

        //add code below this line

        Meal dinner = new Meal();
        dinner.addDrink("water");
        dinner.addAppetizer("bruschetta");
        dinner.addCourse("roast chicken");
        dinner.addDessert("chocolate cake");

        System.out.println(dinner.drinks);
        System.out.println(dinner.appetizers);
        System.out.println(dinner.mainCourse);
```

```
System.out.println(dinner.dessert);
```

*//add code above this line*

```
}
```

```
}
```

# Printing the Meal 1

---

## Planning the Method

Before writing the method to print the meal, think about what you want the output should like. Imagine that a meal consists of the following courses:

- Drinks - water and coffee
- Appetizers - nothing served as an appetizer
- Main course - roast chicken, mashed potatoes, and salad.
- Dessert - chocolate cake

Change your code to reflect this meal. Also, add the `printMeal` instance method even though it has not yet been declared.

```
//add code below this line

Meal dinner = new Meal();
dinner.addDrink("water");
dinner.addDrink("coffee");
dinner.addCourse("roast chicken");
dinner.addCourse("mashed potatoes");
dinner.addCourse("salad");
dinner.addDessert("chocolate cake");
dinner.printMeal();

//add code above this line
```

The `printMeal` instance method is going to invoke the helper method `printCourse`. Call `printCourse` four times, passing it the `ArrayList` that represents the course as well as the name of the course.

```

void addDessert(String d) {
    dessert.add(d);
}

void printMeal() {
    printCourse(drinks, "drinks");
    printCourse(appetizers, "appetizers");
    printCourse(mainCourse, "main course");
    printCourse(dessert, "dessert");
}

```

The `printCourse` method should be able to handle an empty `ArrayList` (nothing served), an `ArrayList` of length 1, an `ArrayList` of length 2, and an `ArrayList` of 3 or more elements. Each of these scenarios has specific requirements: Is the verb singular or plural? Do you need a comma-separated list or just the word "and"? Should a word be capitalized?

## Nothing was Served

Printing a message for an empty `ArrayList` becomes tricky because the sentence changes based on the course.

- No **drinks were** served with the meal.
- No **appetizers were** served with the meal.
- No **main course was** served with the meal.
- No **dessert was** served with the meal.

Start by checking to see if `course` is an empty `ArrayList` using the `size` method.

```

void printMeal() {
    printCourse(drinks, "drinks");
    printCourse(appetizers, "appetizers");
    printCourse(mainCourse, "main course");
    printCourse(dessert, "dessert");
}

void printCourse(ArrayList<String> course, String name) {
    if (course.size() == 0) { // check for empty ArrayList
}
}

```

Next, create a string variable `verb` that will represent the text in bold above. Then ask if the `name` parameter matches each of the four courses: "drinks", "appetizers", "main course", or "dessert". When you have a

match, set verb to the appropriate string (the bold text above). Print a sentence that tells the user that no items were served for that course. Be sure to incorporate the variable verb to provide the proper context.

```
void printCourse(ArrayList<String> course, String name) {  
    if (course.size() == 0) { // check for empty ArrayList  
        String verb = "";  
        if (name.equals("drinks")) {  
            verb = "drinks were";  
        } else if (name.equals("appetizers")) {  
            verb = "appetizers were";  
        } else if (name.equals("main course")) {  
            verb = "main course was";  
        } else if (name.equals("dessert")) {  
            verb = "dessert was";  
        }  
        System.out.println("No " + verb + " served with the  
        meal.");  
    }  
}
```

Running the code now should produce "No appetizers were served with the meal." since it is the only course that is an empty ArrayList.

challenge

## Try this variation:

Use the comment symbol // to comment out all of the lines with a method that adds a course to the dinner object. Run the program. The output should be:

```
No drinks were served with the meal.  
No appetizers were served with the meal.  
No main course was served with the meal.  
No dessert was served with the meal.
```

### ▼ Code

```
//add code below this line  
  
Meal dinner = new Meal();  
// dinner.addDrink("water");  
// dinner.addDrink("coffee");  
// dinner.addCourse("roast chicken");  
// dinner.addCourse("mashed potatoes");  
// dinner.addCourse("salad");  
// dinner.addDessert("chocolate cake");  
dinner.printMeal();  
  
//add code above this line
```

# Printing the Meal 2

---

## One Item Was Served

This is a relatively simple case. The trickiest part will be capitalizing the word at the beginning of the sentence. Start by asking if the size of course is 1. If only one item is served, that item should be capitalized followed by "was served with the meal.".

```
void printCourse(ArrayList<String> course, String name) {  
    if (course.size() == 0) { // check for empty ArrayList  
        String verb = "";  
        if (name.equals("drinks")) {  
            verb = "drinks were";  
        } else if (name.equals("appetizers")) {  
            verb = "appetizers were";  
        } else if (name.equals("main course")) {  
            verb = "main course was";  
        } else if (name.equals("dessert")) {  
            verb = "dessert was";  
        }  
        System.out.println("No " + verb + " served with the  
                           meal.");  
    } else if (course.size() == 1) { // check for one item  
  
    }  
}
```

Create the string variable `item` and set it to the first element in `course`. This string needs to be capitalized. Use `substring(0,1)` to represent the first character in the string. Then call the `toUpperCase()` method to capitalize this character. Finally, concatenate this character with the rest of the string, which is represented by `substring(1)`. Print out a sentence using the `item` string.

```
} else if (course.size() == 1) { // check for one item  
    String item = course.get(0);  
    item = item.substring(0, 1).toUpperCase() +  
          item.substring(1);  
    System.out.println(item + " was served with the meal.");  
}
```

**Note**, remove the comment symbol // for dinner.addDessert("chocolate cake");.

## Two Items Were Served

If there are two items being served, the first item should be capitalized followed by and and the second item. The sentence will end with " were served with the meal.". Start by asking if the size of course is 2.

```
void printCourse(ArrayList<String> course, String name) {  
    if (course.size() == 0) {  
        String verb = "";  
        if (name.equals("drinks")) {  
            verb = "drinks were";  
        } else if (name.equals("appetizers")) {  
            verb = "appetizers were";  
        } else if (name.equals("main course")) {  
            verb = "main course was";  
        } else if (name.equals("dessert")) {  
            verb = "dessert was";  
        }  
        System.out.println("No " + verb + " served with the  
                           meal.");  
    } else if (course.size() == 1) { // check for one item  
        String item = course.get(0);  
        item = item.substring(0, 1).toUpperCase() +  
              item.substring(1);  
        System.out.println(item + " was served with the meal.");  
    } else if (course.size() == 2) { // check for two items  
        }  
    }
```

Create the string variables item1 and item2. Set them to the two elements in course. Capitalize item1 just as before. Print out a sentence that incorporates item1 and item2.

```
} else if (course.size() == 2) { // check for two items  
    String item1 = course.get(0);  
    String item2 = course.get(1);  
    item1 = item1.substring(0, 1).toUpperCase() +  
           item1.substring(1);  
    System.out.println(item1 + " and " + item2 + " were served  
                       with the meal.");  
}
```

**Note**, remove the comment symbol // for dinner.addDrink("water"); and dinner.addDrink("coffee");.

## More than Two Items Were Served

If more than two items are served, then you need a comma-separated list. The first item should be capitalized followed by a comma and a space. The next items are followed by commas and spaces. The final item in the list is prefaced with and. No comma is used after the last item. The sentence ends with " were served with the meal.". Start by using an else statement to capture all instances where the size of course is greater than 2.

```
void printCourse(ArrayList<String> course, String name) {  
    if (course.size() == 0) {  
        String verb = "";  
        if (name.equals("drinks")) {  
            verb = "drinks were";  
        } else if (name.equals("appetizers")) {  
            verb = "appetizers were";  
        } else if (name.equals("main course")) {  
            verb = "main course was";  
        } else if (name.equals("dessert")) {  
            verb = "dessert was";  
        }  
        System.out.println("No " + verb + " served with the  
                           meal.");  
    } else if (course.size() == 1) { // check for one item  
        String item = course.get(0);  
        item = item.substring(0, 1).toUpperCase() +  
              item.substring(1);  
        System.out.println(item + " was served with the meal.");  
    } else if (course.size() == 2) { // check for two items  
        String item1 = course.get(0);  
        String item2 = course.get(1);  
        item1 = item1.substring(0, 1).toUpperCase() +  
               item1.substring(1);  
        System.out.println(item1 + " and " + item2 + " were served  
                           with the meal.");  
    } else { // more than two items  
  
    }  
}
```

Create the string variable item and set it to the first element in course. Capitalize this string just as before. Use print instead of println to print this capitalized string followed by a comma and a space.

```

} else { // more than two items
    String item = course.get(0);
    item = item.substring(0, 1).toUpperCase() +
        item.substring(1);
    System.out.print(item + ", ");
}

```

Create a for loop to iterate over the course ArrayList. We have already printed the first element from the ArrayList. So initialize the loop variable with 1 instead of 0. The last element in course needs to have the word and appear before element. The last element occurs when i is equal to the size of course minus 1. Check for this condition, and use a print statement when printing and and the element. If it is not the last element, use a print statement to print the element followed by a comma. After the loop, use a println statement to print the rest of the sentence.

```

} else { // more than two items
    String item1 = course.get(0);
    item1 = item1.substring(0, 1).toUpperCase() +
        item1.substring(1);
    System.out.print(item1 + ", ");
    for (int i = 1; i < course.size(); i++) {
        if (i == course.size() - 1) { // check for last element
            System.out.print("and " + course.get(i) + " ");
        } else {
            System.out.print(course.get(i) + ", ");
        }
    }
    System.out.println("were served with the meal.");
}

```

**Note**, remove the comment symbol // for remaining lines of code.

challenge

## Check your work:

Create different meals and make sure your program works as expected. For example:

```
//add code below this line

Meal dinner = new Meal();
dinner.addDrink("white wine");
dinner.addAppetizer("tapenade");
dinner.addAppetizer("antipasto");
dinner.addCourse("cauliflower bolognese");
dinner.addCourse("butternut squash soup");
dinner.addCourse("kale salad");
dinner.printMeal();

//add code above this line
```

## ▼ Code

```
import java.util.ArrayList;

//add class definitions below this line

class Meal {
    ArrayList<String> drinks = new ArrayList<String>();
    ArrayList<String> appetizers = new ArrayList<String>();
    ArrayList<String> mainCourse = new ArrayList<String>();
    ArrayList<String> dessert = new ArrayList<String>();

    void addDrink(String d) {
        drinks.add(d);
    }

    void addAppetizer(String a) {
        appetizers.add(a);
    }

    void addCourse(String c) {
        mainCourse.add(c);
    }

    void addDessert(String d) {
        dessert.add(d);
    }

    void printMeal() {
        printCourse(drinks, "drinks");
        printCourse(appetizers, "appetizers");
        printCourse(mainCourse, "main course");
        printCourse(dessert, "dessert");
    }
}
```

```

}

void printCourse(ArrayList<String> course, String name)
{
    if (course.size() == 0) {
        String verb = "";
        if (name.equals("drinks")) {
            verb = "drinks were";
        } else if (name.equals("appetizers")) {
            verb = "appetizers were";
        } else if (name.equals("main course")) {
            verb = "main course was";
        } else if (name.equals("dessert")) {
            verb = "dessert was";
        }
        System.out.println("No " + verb + " served with the
meal.");
    } else if (course.size() == 1) { // check for one item
        String item = course.get(0);
        item = item.substring(0, 1).toUpperCase() +
        item.substring(1);
        System.out.println(item + " was served with the
meal.");
    } else if (course.size() == 2) { // check for two
items
        String item1 = course.get(0);
        String item2 = course.get(1);
        item1 = item1.substring(0, 1).toUpperCase() +
        item1.substring(1);
        System.out.println(item1 + " and " + item2 + " were
served with the meal.");
    } else { // more than two items
        String item1 = course.get(0);
        item1 = item1.substring(0, 1).toUpperCase() +
        item1.substring(1);
        System.out.print(item1 + ", ");
        for (int i = 1; i < course.size(); i++) {
            if (i == course.size() - 1) {
                System.out.print("and " + course.get(i) + " ");
            } else {
                System.out.print(course.get(i) + ", ");
            }
        }
        System.out.println("were served with the meal.");
    }
}

//add class definitions above this line

public class MoreMethods {

```

```
public static void main(String[] args) {  
  
    //add code below this line  
  
    Meal dinner = new Meal();  
    dinner.addDrink("white wine");  
    dinner.addAppetizer("tapenade");  
    dinner.addAppetizer("antipasto");  
    dinner.addCourse("cauliflower bolognese");  
    dinner.addCourse("butternut squash soup");  
    dinner.addCourse("kale salad");  
    dinner.printMeal();  
  
    //add code above this line  
}  
}
```

# **Changing Objects with Instance Methods Formative Assessment 1**

---

# **Changing Objects with Instance Methods Formative Assessment 2**

---

# **Learning Objectives - Static Methods**

---

- Define the term static method
- Identify when using a static method makes sense
- Demonstrate the syntax for calling an instance method
- Explain the relationship between static methods and objects
- Identify the limitations of static methods

# Static Methods

---

## Static Methods

Imagine a Rectangle class in which objects have a length, width, and a method to calculate the area.

```
//add class definitions below this line

class Rectangle {
    int width;
    int length;

    Rectangle(int w, int l) {
        width = w;
        length = l;
    }

    int area() {
        return width * length;
    }
}

//add class definitions above this line
```

Create two instances of the Rectangle class, and then calculate the combined area of the two rectangles.

```
//add code below this line

Rectangle rect1 = new Rectangle(12, 27);
Rectangle rect2 = new Rectangle(9, 3);
int combinedArea = rect1.area() + rect2.area();
System.out.println(combinedArea);

//add code above this line
```

This works, but the combined area has to be calculated by the user. Since the combined area is related to the Rectangle class, a better solution would be to add this functionality to the class. Another type of method in Java is a

static method. Static methods use the `static` keyword in the method definition.

Static methods are most often used to add functionality to the whole class and not just an instance of the class. A good rule of thumb is to use a static method when none of the attributes of an instance are changed. In the example below, the attributes for the `r1` and `r2` object do not change. So a static method is a better choice than an instance method.

```
//add class definitions below this line

class Rectangle {
    int width;
    int length;

    Rectangle(int w, int l) {
        width = w;
        length = l;
    }

    int area() {
        return width * length;
    }

    static int combinedArea(Rectangle r1, Rectangle r2) {
        return r1.area() + r2.area();
    }
}

//add class definitions above this line
```

Static methods are called in a unique way. They still use dot notation, but instead of using the instance name use the class name followed by the dot and method name.

```
//add code below this line

Rectangle rect1 = new Rectangle(12, 27);
Rectangle rect2 = new Rectangle(9, 3);
System.out.println(Rectangle.combinedArea(rect1, rect2));

//add code above this line
```

challenge

## Try this variation:

Create the static method `describe` for the `Rectangle` class that prints a description of the rectangle.

### ▼ Code

```
static void describe(Rectangle r) {  
    System.out.println("The rectangle has width of " +  
        r.width + " and a length of " + r.length + ".");  
}
```

# Independence From Objects

---

## Independence From Objects

Another unique characteristic of static methods is that they do not require the instantiation of an object before you can use them. Take a look at the `Car` class. It has some instance attributes, a constructor, and the static method `honk`.

```
//add class definitions below this line

class Car {
    String make;
    String model;
    String color;

    Car(String ma, String mo, String co) {
        make = ma;
        model = mo;
        color = co;
    }

    static void honk() {
        System.out.println("Beep! Beep!");
    }
}

//add class definitions above this line
```

In the `main` method, **do not** instantiate a `Car` object. Instead, call the `honk` method from the class.

```
//add code below this line

Car.honk();

//add code above this line
```

This works because `honk` is a static method. If you remove the `static` keyword from the method definition, your program will not compile. You have used several static methods several times up to now. Type casting

between different data types is often done with static methods.

- `String.valueOf(20)` - This static method returns the string representation of integer 20.
- `Integer.parseInt("20")` - This static method returns the integer representation of the string "20".
- `Double.valueOf(20)` - This static method returns the double representation of the integer 20.

challenge

## Try this variation:

The `Math` class is full of static methods. Using the [Java documentation](#) for the `Math` class, see if you can write code using static methods that do the following things:

- Prints the largest number between 17 and 142
- Prints the absolute value of -2.34
- Prints 3 to the power of 5
- Prints the cosine of 34.1
- Prints a random number between 0 and 1

### ▼ Code

Here are the static methods:

```
System.out.println(Math.max(17, 142));
System.out.println(Math.abs(-2.34));
System.out.println(Math.pow(3, 5));
System.out.println(Math.cos(34.1));
System.out.println(Math.random());
```

## Limits of Static Methods

Because static methods are independent from objects, that means that static methods cannot directly access instance attributes of an object. Add the following static method to the `Car` class.

```
static void describe() {
    System.out.println(color + " " + make + " " + model);
}
```

Now call this method from the `Car` class as before. **Important**, running this code will cause an error. Java says that a non-static variable (instance attribute) cannot be referenced from a static context. That is, the static method `describe` cannot directly access the `color`, `make`, or `model` instance attributes of the `Car` class.

```
//add code below this line  
  
Car.describe();  
  
//add code above this line
```

The way to avoid these types of errors is to pass a `Car` object to the `describe` method. Static methods can indirectly access instance attributes through an instance of the class. Change the `describe` method so that it has a `Car` object as a parameter. `describe` can now access the instance attributes by referencing the object name and the attribute using dot notation.

```
static void describe(Car c) {  
    System.out.println(c.color + " " + c.make + " " + c.model);  
}
```

Instantiate an instance of the `Car` class, and then pass this instance to the `describe` method. Java should run the code without an error.

```
//add code below this line  
  
Car myCar = new Car("Honda", "Accord", "red");  
Car.describe(myCar);  
  
//add code above this line
```

# **Static Methods Formative Assessment 1**

---

# **Static Methods Formative Assessment 2**

---

# **Learning Objectives - Superclass & Subclass**

---

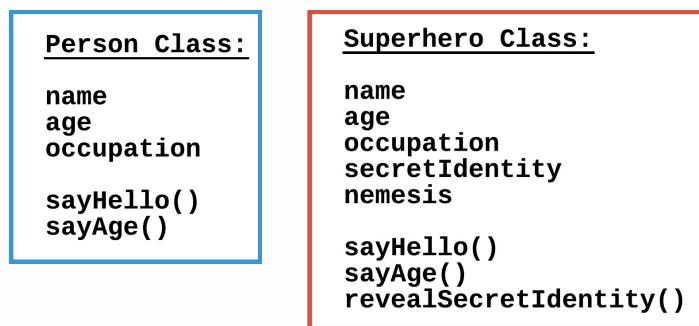
- Define the terms inheritance, superclass, and subclass
- Explain the relationship between the superclass and subclass
- Explain the role of the `super` keyword
- Create a subclass class from a given superclass
- Use `instanceof` to determine an object's parent class
- Define the substitution principle

# What is Inheritance?

---

## Defining Inheritance

Imagine you want to create two Java classes, Person and Superhero. These respective classes might look something like this:



### No\_Inheritance

There are some similarities between the Person class and the Superhero class. If the Person class already exists, it would be helpful to “borrow” from the Person class so you only have to create the new attributes and methods for the Superhero class. This situation describes inheritance — one class copies the attributes and methods from another class.

## Inheritance Syntax

In the IDE on the left, the Person class is already defined. To create the Superhero class that inherits from the Person class, add the following code at the end of the program. Notice how Superhero definition adds `extends Person`. This is how you indicate to Java that the Superhero class inherits from the Person class. You can also say that Person is the superclass and Superhero is the subclass.

```
//add class definitions below this line

class Superhero extends Person {

}

//add class definitions above this line
```

Now declare an instance of the Superhero class and print the value of the name and age attributes using their getter methods.

```
//add code below this line  
  
Superhero s = new Superhero();  
System.out.println(s.getName());  
System.out.println(s.getAge());  
  
//add code above this line
```

#### ▼ Why does the program print null and 0?

The Person class does not have a constructor. So when Java creates the attributes, it gives null as the initial value for strings and 0 for the initial value of integers.

challenge

### Try these variations:

- Print the occupation attribute.
- Call the sayHello method from the Superhero object.

#### ▼ Solution

```
//add code below this line  
  
Superhero s = new Superhero();  
System.out.println(s.getName());  
System.out.println(s.getAge());  
System.out.println(s.getOccupation());  
s.sayHello();  
  
//add code above this line
```

## Limitations of Inheritance

Java place some rules about how inheritance works. Most importantly, only public methods and attributes in superclass are directly accessible to the subclass. The following code creates an error. While the subclass inherits the name attribute, it can only access it through the getter method. For inheritance to work properly, the superclass needs to have getters, setters, and other public methods

```
//add code below this line  
  
Superhero s = new Superhero();  
System.out.println(s.name);  
  
//add code above this line
```

Constructors can also be a bit tricky with inheritance. Java will automatically call the constructor of the superclass when the subclass object is instantiated. The superclass should have a constructor without any parameters. Add the following constructor to the Person class. This will avoid the situation above where attributes have values like null and 0.

```
public Person() {  
    name = "Sarah";  
    age = 37;  
    occupation = "VP Sales";  
}
```

Now call the sayHello and sayAge methods from the Superhero object.

```
//add code below this line  
  
Superhero s = new Superhero();  
s.sayHello();  
s.sayAge();  
  
//add code above this line
```

challenge

## Try these variations:

- Change the constructor of the Person class so that the default values are Rodrigo, 19, student.

### ▼ Solution

```
public Person() {  
    name = "Rodrigo";  
    age = 19;  
    occupation = "student";  
}
```

- Change the constructor of the Person class to look like the following code:

```
public Person(String n, int a, String o) {  
    name = n;  
    age = a;  
    occupation = o;  
}
```

### ▼ Why is there an error?

Java tries to call the constructor of the superclass when instantiating a subclass object. This happens automatically and without any parameters, but the Person class expects three parameters. This is why the code generates an error. The next page describes how to call a superclass constructor with parameters.

# Super

---

## The super Keyword

The `super` keyword is used in the subclass to invoke methods in the superclass. This is how constructors with parameters are called in inheritance. The `Person` class to the left has a constructor without any parameters. Add a second constructor to the `Person` class that has three parameters.

```
public Person() {  
    name = "Sarah";  
    age = 37;  
    occupation = "VP Sales";  
}  
  
public Person(String n, int a, String o) {  
    name = n;  
    age = a;  
    occupation = o;  
}
```

By default, Java will call the constructor with no parameters when the `Superhero` object is instantiated. Create a constructor for the `Superhero` class with three parameters. Use the `super` keyword followed by the parameters. Java will pass the parameters to the constructor for `Person` because that is the superclass. You are now able to specify the name, age, and occupation attributes for the subclass.

```
//add class definitions below this line  
  
class Superhero extends Person {  
    public Superhero(String n, int a, String o) {  
        super(n, a, o);  
    }  
}  
  
//add class definitions above this line
```

Instantiate a Superhero object with a name, age, and occupation. Call the sayName and sayAge methods to verify that the Superhero object has the values Wonder Woman and 27 for the name and age attributes.

```
//add code below this line  
  
Superhero hero = new Superhero("Wonder Woman", 27,  
    "intelligence officer");  
hero.sayHello();  
hero.sayAge();  
  
//add code above this line
```

challenge

## Try this variation:

Add the following constructor to the Person class.

```
public Person(String n) {  
    name = n;  
    age = 0;  
    occupation = "";  
}
```

Create a constructor for the Superhero class that will call this new constructor in the superclass.

### ▼ Solution

```
public Superhero(String n) {  
    super(n);  
}
```

You can call this new constructor with the following object instantiation:

```
//add code below this line  
  
Superhero hero = new Superhero("Wonder Woman");  
hero.sayHello();  
  
//add code above this line
```

## Every Class has a Superclass

The Superhero class has a superclass because it extends the Person class. But what about the Person class, does it have a superclass? In Java, every class has a superclass. The code below creates Person object. The variable `c` represents the class of the Person object, and the variable `sc` represents the superclass of the Person object.

```
//add code below this line

Person p = new Person();
Class c = p.getClass();
Class sc = c.getSuperclass();
System.out.println("Class: " + c);
System.out.println("Superclass: " + sc);

//add code above this line
```

The second line of output should be `class java.lang.Object`. That means the `Object` class is the superclass for every object in Java. However, you do not need to declare that a user-defined class extends the `Object` class. Java assumes this automatically.

# Inheritance Hierarchy

---

## Inheritance Hierarchy

You have seen how the `Superhero` class becomes the subclass of the `Person` class through inheritance. The relationship between these two classes is called inheritance (or class) hierarchy. Java has a built-in operator to help you determine the hierarchy of classes.

Start by creating four classes, where `ClassA` is the superclass to `ClassB` and `ClassC` is the superclass to `ClassD`. These classes do not need to do anything, so do not create attributes, a constructor, or methods.

```
//add class definitions below this line

class ClassA {}
class ClassB extends ClassA {}
class ClassC {}
class ClassD extends ClassC {}

//add class definitions above this line
```

Now, create an instance for each class.

```
//add code below this line

ClassA a = new ClassA();
ClassB b = new ClassB();
ClassC c = new ClassC();
ClassD d = new ClassD();

//add code above this line
```

The `instanceof` operator returns a boolean when comparing an object and a class. It returns `true` if the object is an instance of a class, and it returns `false` if the object is not an instance of a class. The code below prints the class for object `b` which is `ClassB`. It then prints `true` because object `b` is an instance of `ClassB`. Surprisingly, it also prints `true` when asking if `b` is an instance of `ClassA`. This is because of inheritance. Because `ClassB` inherits from `ClassA`, `b` is considered to be an instance of `ClassA`.

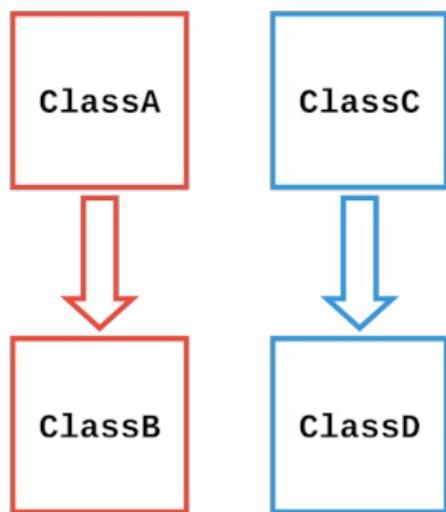
```
//add code below this line

ClassA a = new ClassA();
ClassB b = new ClassB();
ClassC c = new ClassC();
ClassD d = new ClassD();

System.out.println(b.getClass());
System.out.println(b instanceof ClassB);
System.out.println(b instanceof ClassA);

//add code above this line
```

Think of inheritance hierarchy as a downward flow chart. The superclass is on top while the subclass is below. The hierarchy is a one-way relationship. Inheritance always flows from the superclass to the subclass.



.guides/img/inheritance/inheritance\_herarchy

challenge

### Try these variations:

Add the following code to your program.

```
* System.out.println(a instanceof ClassB);
* System.out.println(d instanceof ClassC);
```

# Substitution Principle

---

## Substitution Principle

When one class inherits from another, Java considers them to be related. They may have different data types, Java allows a subclass to be used in place of the superclass. This is called the substitution principle. In the code below, ClassB inherits from ClassA. Both classes have the greeting method which prints a greeting specific to the class.

```
//add class definitions below this line

class ClassA {
    public void greeting() {
        System.out.println("Hello from Class A");
    }
}

class ClassB extends ClassA {
    public void greeting() {
        System.out.println("Hello from Class B");
    }
}

//add class definitions above this line
```

According to the substitution principle, an object of ClassB can be used in a situation that expects an object of ClassA. The substitution method explicitly calls for an argument of ClassA.

```
//add method definitions below this line

public static void substitution(ClassA a) {
    a.greeting();
}

//add method definitions above this line
```

Instantiate an object of ClassB and pass it to the substitution method. Even though the object b has the wrong data type, the code should still work due to the substitution principle. Because ClassB extends ClassA,

object b can be used in place of an object of type ClassA. Run the code to verify the output.

```
//add code below this line  
  
ClassB b = new ClassB();  
substitution(b);  
  
//add code above this line
```

challenge

### Try this variation:

- Remove `extends ClassA` from the `ClassB` declaration and run the code again.
  - ▼ **Why did this produce an error?**  
Deleting `extends ClassA` means that `ClassB` no longer inherits from `ClassA`. Therefore, the substitution principle no longer applies. Java now says there is a type mismatch error.

## The Substitution Principle is a One-Way Relationship

Like inheritance hierarchy, the substitution principle is a one-way relationship. Add a third class `ClassC` that inherits from `ClassB`. We now have an inheritance chain that flows from `ClassA` to `ClassB` to `ClassC`.

```
//add class definitions below this line

class ClassA {
    public void greeting() {
        System.out.println("Hello from Class A");
    }
}

class ClassB extends ClassA {
    public void greeting() {
        System.out.println("Hello from Class B");
    }
}

class ClassC extends ClassB {
    public void greeting() {
        System.out.println("Hello from Class C");
    }
}

//add class definitions above this line
```

The substitution principle states that the substitution will work with any subclass. Since ClassB is a subclass of ClassA and ClassC is a subclass of ClassA, then ClassC can be substituted for ClassA. Create an object of type ClassC and pass it to the substitution method.

```
//add code below this line

ClassC c = new ClassC();
substitution(c);

//add code above this line
```

Change the method header for substitution so that the parameter is of type ClassB.

```
//add method definitions below this line

public static void substitution(ClassB b) {
    b.greeting();
}

//add method definitions above this line
```

Now create an object of type ClassA and pass it to the substitution method. This should now cause an error because the substitution principle no longer applies because the substitution expects ClassB or a subclass. ClassA is a superclass, so it cannot be substituted for ClassB.

```
//add code below this line  
  
ClassA a = new ClassA();  
substitution(a);  
  
//add code above this line
```

# **Superclass & Subclass Formative Assessment 1**

---

# **Superclass & Subclass Formative Assessment 2**

---

# **Learning Objectives - Extending & Overriding**

---

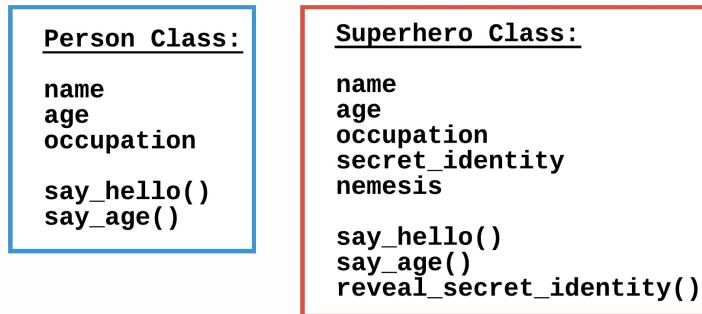
- Define the terms extending and overriding
- Extend the superclass with new method
- Override methods from the superclass with new functionality
- Annotate a method when overriding it

# Extending a Class

---

## Extending the Constructor

The idea of inheritance is to borrow from a superclass and then add on functionality. Up until now, we have talked about borrowing from a superclass. The process of adding functionality to a subclass is known as either extending or overriding. Extending a class means that new attributes and methods are given to the subclass.



Superhero and Person Classes

The code below will first call upon the superclass constructor (using `super`) to create the attributes `name`, `age`, and `occupation`. The constructor is extended when the attribute `secretIdentity` is added to the `Superhero` class.

```
//add class definitions below this line

class Superhero extends Person {
    private String secretIdentity;

    public Superhero(String na, int a, String o, String s) {
        super(na, a, o);
        secretIdentity = s;
    }

    public String getSecretIdentity() {
        return secretIdentity;
    }

    public void setSecretIdentity(String newIdentity) {
        secretIdentity = newIdentity;
    }
}

//add class definitions above this line
```

Instantiate a `Superhero` object and print out each of the attributes. You should see the three attributes from the `Person` class as well as the new attribute `secretIdentity`.

```
//add code below this line

Superhero hero = new Superhero("Spider-Man", 17, "student",
    "Peter Parker");
System.out.println(hero.getName());
System.out.println(hero.getAge());
System.out.println(hero.getOccupation());
System.out.println(hero.getSecretIdentity());

//add code above this line
```

### ▼ Inheritance is a One-Way Street

Inheritance shares attributes and methods from the superclass to the subclass. When a subclass is extended, it cannot share the new additions with their superclass. In the code above, `Superhero` has access to `name`, but `Person` does not have access to `secretIdentity`.

challenge

## Try this variation:

- Rewrite the Superhero class so that it extends the Person class by adding the attribute nemesis, Doc Octopus.

### ▼ Solution

```
//add class definitions below this line

class Superhero extends Person {
    private String secretIdentity;
    private String nemesis;

    public Superhero(String na, int a, String o, String s,
                     String ne) {
        super(na, a, o);
        secretIdentity = s;
        nemesis = ne;
    }

    public String getSecretIdentity() {
        return secretIdentity;
    }

    public void setSecretIdentity(String newIdentity) {
        secretIdentity = newIdentity;
    }

    public String getNemesis() {
        return nemesis;
    }

    public void setNemesis(String newNemesis) {
        nemesis = newNemesis;
    }
}

//add class definitions above this line
```

Now instantiate a Superhero object and make sure that the new attribute works as expected.

```
//add code below this line

Superhero hero = new Superhero("Spider-Man", 17,
    "student", "Peter Parker", "Doc Octopus");
System.out.println(hero.getName());
System.out.println(hero.getAge());
System.out.println(hero.getOccupation());
System.out.println(hero.getSecretIdentity());
System.out.println(hero.getNemesis());

//add code above this line
```

## Extending a Class by Adding New Methods

Another way to extend a class is to create new methods (besides getters and setters) that are unique to the subclass. For example, the `sayHello` method will give the superhero's name, but it will not divulge their secret identity. Create the method `revealSecretIdentity` to print the attribute `secretIdentity`.

```

//add class definitions below this line

class Superhero extends Person {
    private String secretIdentity;
    private String nemesis;

    public Superhero(String na, int a, String o, String s, String ne) {
        super(na, a, o);
        secretIdentity = s;
        nemesis = ne;
    }

    public String getSecretIdentity() {
        return secretIdentity;
    }

    public void setSecretIdentity(String newIdentity) {
        secretIdentity = newIdentity;
    }

    public String getNemesis() {
        return nemesis;
    }

    public void setNemesis(String newNemesis) {
        nemesis = newNemesis;
    }

    public void revealSecretIdentity() {
        System.out.println("My real name is " + secretIdentity +
            ".");
    }
}

//add class definitions above this line

```

Now test out the newly added method.

```

//add code below this line

Superhero hero = new Superhero("Spider-Man", 17, "student",
    "Peter Parker", "Doc Octopus");
hero.revealSecretIdentity();

//add code above this line

```

challenge

## Try this variation:

- Create the method `sayNemesis` that prints the string:

`My nemesis is Doc Octopus..`

▼ **Solution**

```
//add class definitions below this line

class Superhero extends Person {
    private String secretIdentity;
    private String nemesis;

    public Superhero(String na, int a, String o, String s,
                     String ne) {
        super(na, a, o);
        secretIdentity = s;
        nemesis = ne;
    }

    public String getSecretIdentity() {
        return secretIdentity;
    }

    public void setSecretIdentity(String newIdentity) {
        secretIdentity = newIdentity;
    }

    public String getNemesis() {
        return nemesis;
    }

    public void setNemesis(String newNemesis) {
        nemesis = newNemesis;
    }

    public void revealSecretIdentity() {
        System.out.println("My real name is " + secretIdentity +
                           ".");
    }

    public void sayNemesis() {
        System.out.println("My nemesis is " + nemesis + ".");
    }
}

//add class definitions above this line
```

Now invoke the sayNemesis method.

```
//add code below this line  
  
Superhero hero = new Superhero("Spider-Man", 17,  
    "student", "Peter Parker", "Doc Octopus");  
hero.sayNemesis();  
  
//add code above this line
```

# Method Overriding

---

## Overriding a Method

Extending a class means adding new attributes or methods to the subclass. Another way to add new functionality to a subclass is through method overriding. Overriding a method means to inherit a method from the superclass, keep its name, but change the contents of the method.

Extend the `Superhero` class by overriding the `sayHello`. Add this method to the `Superhero` class. Remember, the `name` attribute is part of the superclass, so you need to use the `getName` method to access this attribute.

```
public void sayHello() {  
    System.out.println("My name is " + getName() + ", and  
                      criminals fear me.");  
}
```

Instantiate a `Superhero` object and call the `sayHello` method.

```
//add code below this line  
  
Superhero hero = new Superhero("Storm", 30, "Queen of  
                               Wakanda", "Ororo Munroe", "Shadow King");  
hero.sayHello();  
  
//add code above this line
```

### ▼ Differentiating Overriding and Extending

The difference between extending and overriding can be slight. Both approaches are used to make a subclass unique from the superclass. Overriding deals with changing a pre-existing method from the superclass, while extending deals with adding new methods and attributes.

challenge

## Try this variation:

- Override the `sayAge` method so that it prints the string:  
Young or old, I will triumph over evil.

### ▼ Solution

Add the following method to the `Superhero` class:

```
public void sayAge() {  
    System.out.println("Young or old, I will triumph over  
    evil.");  
}
```

Call the method to verify it works as expected.

```
//add code below this line  
  
Superhero hero = new Superhero("Storm", 30, "Queen of  
    Wakanda", "Ororo Munroe", "Shadow King");  
hero.sayHello();  
hero.sayAge();  
  
//add code above this line
```

## What Happens When You Override a Method?

If you can override a method from the superclass, what happens to the original method? Java defaults to the instance. So `hero.sayHello()` will always use the method from the `Superhero` class. But that does not mean you cannot call `sayHello` from the `Person` class. Just as we used the `super` keyword to access the constructor from the superclass, we can use `super` to invoke `sayHello` from the `Person` class. Add the method `oldHello` to the `Superhero` class. Use `super` followed by the method name to tell Java to look in the `Person` class.

```
public void sayHello() {
    System.out.println("My name is " + getName() + ", and
                      criminals fear me.");
}

public void sayAge() {
    System.out.println("Young or old, I will triumph over
                      evil.");
}

public void oldHello() {
    super.sayHello();
}
```

```
//add code below this line

Superhero hero = new Superhero("Storm", 30, "Queen of
                               Wakanda", "Ororo Munroe", "Shadow King");
hero.sayHello();
hero.oldHello();

//add code above this line
```

challenge

## Try this variation:

- Add the method `oldAge` to the `Superhero` class and then call it.

### ▼ Solution

Add the following method to the `Superhero` class.

```
public void oldAge() {  
    super.sayAge();  
}
```

Call the method to verify the output works as expected.

```
//add code below this line  
  
Superhero hero = new Superhero("Storm", 30, "Queen of  
Wakanda", "Ororo Munroe", "Shadow King");  
hero.sayAge();  
hero.oldAge();  
  
//add code above this line
```

# Overriding Annotation

---

## Overriding Annotation

Java allows you to provide an optional annotation when overriding a method. The `@Override` annotation lets the Java compiler know that the following method overrides a method from the superclass. Adding annotations when overriding methods is also helpful when other developers look at your code. Java can flag an issue when method overriding does not work as expected. For example, the class below is supposed to override the `greeting` method, but there is a typo.

```
//add class definitions below this line

class ClassB extends ClassA {
    public void greting() {
        System.out.println("Hello from Class B");
    }
}

//add class definitions above this line
```

Now create instances of `ClassA` and `ClassB` and call the `greeting` method. You would expect the output to be a greet from each class. However, due to a typo, `ClassB` does not override the `greeting` method. Instead it defines the method `greting`. Java runs the program without a problem, but the output is not correct.

```
//add code below this line

ClassA a = new ClassA();
ClassB b = new ClassB();

a.greeting();
b.greeting();

//add code above this line
```

If we add the `@Override` annotation to `ClassB`, you will see how Java notices the problem and brings it to your attention. Java expects `ClassB` to override the `greeting` method. Because it does not, the compiler generates an error

message. Annotating method overriding helps programmers catch small problems that are otherwise hard to track down.

```
//add class definitions below this line

class ClassB extends ClassA {
    @Override
    public void greeting() {
        System.out.println("Hello from Class B");
    }
}

//add class definitions above this line
```

challenge

### Try this variation:

- Fix the typo for the greeting method in ClassB.

#### ▼ Solution

```
class ClassB extends ClassA {
    @Override
    public void greeting() {
        System.out.println("Hello from Class B");
    }
}
```

# Prohibit Overriding

---

## Prohibit Overriding

In Java, you can override any public method from the superclass. The code below overrides the greeting method. But what if you do not want another user to override a method from your class? The `final` keyword will keep users from overriding a method. Notice how the greeting method in `ClassA` uses `final`. Copy and paste this code into the IDE and run it.

```
//add class definitions below this line

class ClassA {
    public final void greeting() {
        System.out.println("Hello from Class A");
    }
}

class ClassB extends ClassA {
    public void greeting() {
        System.out.println("Hello from Class B");
    }
}

//add class definitions above this line
```

Instantiate an object of type `ClassB` and call the `greeting` method. This code will not work. Java says that you cannot override a method that uses `final`.

```
//add code below this line

ClassB b = new ClassB();
b.greeting();

//add code above this line
```

Normally, you would override `greeting` and then use the `super` keyword to access this method in the superclass. Using `final` makes this impossible. Instead of trying to override `greeting`, create the method `greeting2` in `ClassB`.

```
//add class definitions below this line

class ClassA {
    public final void greeting() {
        System.out.println("Hello from Class A");
    }
}

class ClassB extends ClassA {
    public void greeting2() {
        System.out.println("Hello from Class B");
    }
}

//add class definitions above this line
```

Call both `greeting` and `greeting2` with the `b` object. The program should run and print two different greetings.

```
//add code below this line

ClassB b = new ClassB();
b.greeting();
b.greeting2();

//add code above this line
```

# **Extending & Overriding Formative Assessment 1**

---

## **Extending & Overriding Formative Assessment 2**

---

# **Learning Objectives - Multiple Inheritance**

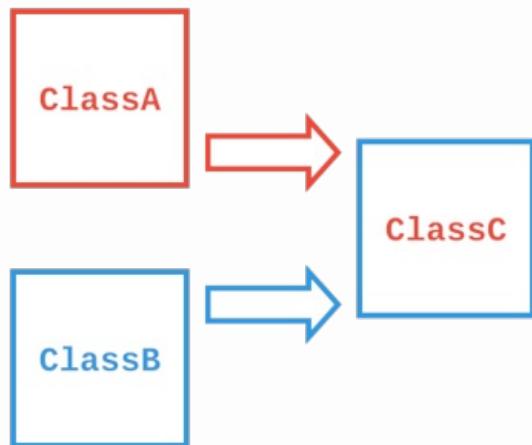
- Define multiple inheritance
- Explain why multiple inheritance from two different classes is not allowed in Java
- Create multiple inheritance from a class with a superclass
- Define the inheritance hierarchy of an object with more than one superclass

# Multiple Inheritance

---

## Multiple Inheritance

Multiple inheritance is a condition where a class inherits from more than one superclass. Java, however, does not allow multiple inheritance if the superclasses are of different types. The image below shows `ClassA` and `ClassB` being of different types. This causes an error message.



Multiple Superclasses

Java prohibits this kind of multiple inheritance because there is an ambiguity problem. Assume that both `ClassA` and `ClassB` have the method `greeting`. If an object of `ClassC` invokes the `greeting` method, which one does Java use? The one from `ClassA` or the one from `ClassB`? This is why a class cannot have more than one superclass of a different type.

## Multilevel Inheritance

Another form of multiple inheritance is called multilevel inheritance. That is a condition where a class inherits from more than one superclass, but each superclass is the same type. The image below shows `ClassC` inheriting from `ClassB`, which in turn inherits from `ClassA`. This is allowed in Java.



### Multilevel Inheritance

The classes Carnivore and Dinosaur are already defined. Carnivore is the superclass for Dinosaur. Create the Tyrannosaurus class which is a subclass for Dinosaur. The constructor for Tyrannosaurus takes a string and two doubles, and it calls the constructor from the Dinosaur class.

```

//add class definitions below this line

class Tyrannosaurus extends Dinosaur {
    public Tyrannosaurus(String d, double s, double w) {
        super(d, s, w);
    }
}

//add class definitions above this line

```

Instantiate a Tyrannosaurus object with the appropriate arguments. This t-rex tiny is 12 meters tall, weighs 14 metric tons, and eats whatever it wants. Print one of the attributes to make sure inheritance is working as expected.

```

//add code below this line

Tyrannosaurus tiny = new Tyrannosaurus("whatever it wants",
    12, 14);
System.out.println(tiny.getSize());

//add code above this line

```

challenge

### Try these variations:

- Print the weight attribute
- Print the diet attribute

# Extending and Overriding Methods

---

## Extending a Class with Multiple Inheritance

Multilevel inheritance has no effect on extending a subclass. The `bonjour` method is not present in either superclass. There is no need for special syntax to extend `ClassC`. Extending a class works just like it does for single inheritance.

```
//add class definitions below this line

class ClassC extends ClassB {
    public void bonjour() {
        System.out.println("Bonjour");
    }
}

//add class definitions above this line
```

Instantiate a `ClassC` object and call the `bonjour` method.

```
//add code below this line

ClassC c = new ClassC();
c.bonjour();

//add code above this line
```

challenge

## Try this variation:

- Extend ClassC with the method goodbye that prints Goodbye. Then call this method.

### ▼ Solution

```
//add class definitions below this line

class ClassC extends ClassB {
    public void bonjour() {
        System.out.println("Bonjour");
    }

    public void goodbye() {
        System.out.println("Goodbye");
    }
}

//add class definitions above this line
```

## Overriding a Method with Multiple Inheritance

Like extending a class, overriding a method works the same in multilevel inheritance as it does in single inheritance. Override the hello method so that it prints a message.

```
//add class definitions below this line

class ClassC extends ClassB {
    public void hello() {
        System.out.println("Hello from Class C");
    }
}

//add class definitions above this line
```

Now call the hello method.

*//add code below this line*

```
ClassC c = new ClassC();
c.hello();
```

*//add code above this line*

# Multiple Inheritance Hierarchy

---

## Determining Inheritance Hierarchy

You can use the `instanceof` operator to determine inheritance hierarchy with multilevel inheritance just like you can with single inheritance. Create the following classes.

```
//add class definitions below this line

class ClassA {}
class ClassB extends ClassA {}
class ClassC {}
class ClassD extends ClassB {}

//add class definitions above this line
```

Now instantiate an object of type `ClassD`. Use the `instanceof` operator to determine the objects inheritance hierarchy. The program should print `true` twice because `ClassD` is a subclass of both `ClassA` and `ClassB` due to multilevel inheritance.

```
//add code below this line

ClassD d = new ClassD();
System.out.println(d instanceof ClassA);
System.out.println(d instanceof ClassB);

//add code above this line
```

challenge

## Try these variations:

- Change the print statement to:

```
System.out.println(d instanceof ClassC);
```

### ▼ Why is this an error?

Object d is not an instance of ClassC, but Java does not return false, why? When there is no inheritance chain between the object and class being compared, Java returns an error.

- Change the program to be:

```
//add code below this line  
  
ClassA a = new ClassA();  
System.out.println(a instanceof ClassB);  
  
//add code above this line
```

### ▼ Why is this false?

Object a is above ClassB in the inheritance chain. That is, ClassB is not a superclass to object a.

## Overridden Methods in the Superclasses

We talked about the ambiguity problem when superclasses of different types have a method with the same name. Java does not know which one to call. With multilevel inheritance, however, this problem does not exist. The code below shows an inheritance chain where Child inherits from Grandparent and Parent. Both Grandparent and Parent have a method called hello.

```

//add class definitions below this line

class Grandparent {
    public void hello() {
        System.out.println("Hello from the Grandparent class");
    }
}

class Parent extends Grandparent {
    public void hello() {
        System.out.println("Hello from the Parent class");
    }
}

class Child extends Parent {}

//add class definitions above this line

```

Instantiate an object of type `Child` and call the `hello` method. Java prints `Hello from the Parent class`. Why? How come the ambiguity problem does not exist? When `Parent` declares the `hello` method, it overrides the `hello` method from `Grandparent`. Therefore Java uses the overriden method from `Parent` and not the method from `Grandparent`.

```

//add code below this line

Child c = new Child();
c.hello();

//add code above this line

```

Now imagine all three class have a `hello` method and that we want to access all of them from `Child`. This is possible as long as you use the `super` keyword. However, when an object of type `Child` uses `super` it is referencing `Parent`. In order for `Child` to invoke an overriden method in `Grandparent`, then `Parent` must also use the `super` keyword.

```

//add class definitions below this line

class Grandparent {
    public void hello() {
        System.out.println("Hello from the Grandparent class");
    }
}

class Parent extends Grandparent {
    public void hello() {
        System.out.println("Hello from the Parent class");
    }

    public void parentHello() {
        super.hello();
    }
}

class Child extends Parent {
    public void hello() {
        System.out.println("Hello from the Child class");
    }

    public void parentHello() {
        super.hello();
    }

    public void grandparentHello() {
        super.parentHello();
    }
}

//add class definitions above this line

```

Instantiate a `Child` object and call each of its three methods. You should see a message from the `Child`, `Parent`, and `Grandparent` classes.

```

//add code below this line

Child c = new Child();
c.hello();
c.parentHello();
c.grandparentHello();

//add code above this line

```



# **Multiple Inheritance Formative Assessment 1**

---

# **Multiple Inheritance Formative Assessment 2**

---

# **Learning Objectives - Advanced Topics**

---

- Set up and import an object defined in a separate file
- Create and manipulate a list of objects
- Define object composition
- Compare and contrast component and composite classes
- Identify when to use composition and when to use inheritance
- Represent an object as a string
- Create and implement an interface
- Compare the values of two objects

# Importing User-Defined Classes

---

## Importing a User-Defined Class

You may have noticed that writing your own classes adds many lines of code before your the logic of your program even begins. To better organize your code, define classes in a separate file. Then import the module into your program so you can use the class.

Make sure you are in the file for defining the class (look at the comments at the top of the files). Start by creating a simple `Employee` class.

### ▼ Switching Between Files

Notice that the IDE on the left now has more than one file. Click on the tabs on the top to switch between the files.



Switching Files

```
// This file is the class definition

class Employee {
    private String name;
    private String title;

    public Employee(String n, String t) {
        name = n;
        title = t;
    }

    public void display() {
        System.out.println("Employee: " + name);
        System.out.println("Title: " + title);
    }
}
```

Now go to the file for your program (look at the comments at the top of the files). Instantiate an `Employee` object with two strings as arguments. Then call the `display` method.

```
//add code below this line

Employee e = new Employee("Calvin", "CEO");
e.display();

//add code above this line
```

## How Does This Work?

You may have noticed that the file that implements the `Employee` object did not use the `import` keyword. How does Java know to look in the `ClassDefinition.java` file to find the class definition for `Employee`? Java has a two-step process to run a program. First you compile it. This turns the source code (the code you wrote) and compiles (transforms) it to byte code. The next step is to run the byte code on the Java Virtual Machine. All of this is hidden behind the TRY IT buttons.

If you compile two (or more) files that are in the same folder, Java now knows about all of the classes in all of the files. You do not need an `import` statement because the compilation process has already informed Java of all of the classes.

The terminal command `javac` means to compile Java source code. Give this command the path to each of the Java files `ImportingClasses.java` and `ClassDefinition.java`. Notice how both Java files are in the same folder. These files **must** be in the same folder. Copy and paste the command below into the terminal and press `Enter` on the keyboard. This is the point where Java would return any error messages. If Java does not do anything, then the compilation process was successful.

```
javac code/advanced/ImportingClasses.java  
code/advanced/ClassDefinition.java
```

Once the byte code has been made, go ahead and run the program. The command to run a Java program is `java`. The parameter `-cp` stands for class path. This is the folder in which the Java program lives. Finally `ImportingClasses` is the actual program (the file with the `main` method). This command will produce the output from your program.

```
java -cp code/advanced/ ImportingClasses
```

The rest of this module will continue to use the `TRY IT` button, it is important to understand how Java can use multiple files for a single program.

# ArrayList of Objects

---

## ArrayList of Objects

You may find yourself needing several instances of a class. Keeping these objects in an ArrayList is a good way to organize your code. It also simplifies interacting with the objects because you can iterate through the ArrayList as opposed to manipulating each object separately.

The first thing you need to do is to create a class. We are going to make a class to represent the apps on your smartphone.

```
// Define the App class

class App {
    private String name;
    private String description;
    private String category;

    public App(String n, String d, String c) {
        name = n;
        description = d;
        category = c;
    }

    public void display() {
        System.out.println(String.format("%s is a(n) %s app that is
                                         %s.", name, category, description));
    }
}
```

Next, we need to create a list with objects of the App class as each element. To speed this up, we are going to read from a CSV file that has the information for the name, description, and category attributes.

### ▼ Why use a CSV file?

This page is about manipulating a list of objects. Instead of manually creating several objects, we are going to read information from the apps.csv file and use it to create several objects in a simple loop.

```

name,description,category
Gmail,the official app for Google's email
    service,communication
FeedWrangler,used to read websites with an RSS feed,internet
Apollo,used to read Reddit,social media
Instagram,the official app for Facebook's Instagram
    service,social media
Overcast,used to manage and listen to podcasts,audio
Slack,the official app for Slack's email
    replacement,communication
YouTube,the official app for Google's video service,video
FireFox,used to browse the web,internet
OverDrive,used to checkout ebooks from the library,ebooks
Authenticator,used for two-factor authentication,internet

```

Make sure that you are altering the `ListofObjects.java` file. Start by creating an `ArrayList` of type `App` and a variable that has the path to the CSV file. Then read the file and skip the header row. Iterate through each row of the CSV file. Add a new `App` object to the list. The first element is the name of the app, the second element is the description, and the third element is the category. Finally, print the `ArrayList`.

```

//add code below this line

ArrayList<App> apps = new ArrayList<App>();
String path = "code/advanced/app.csv";

try {
    CSVReader reader = new CSVReader(new FileReader(path));
    reader.skip(1);
    for (String[] row : reader) {
        apps.add(new App(row[0], row[1], row[2]));
    }
    reader.close();
    System.out.println(apps);
} catch (Exception e) {
    System.out.println(e);
} finally {
    System.out.println("Finished reading a CSV");
}

//add code above this line

```

## ▼ Explaining the Output

The output from the above print statement is an ArrayList of elements that look something like this:

```
App@378bf509
```

This is how Java represents an object. Each element is an App object. The @ symbol and numbers is the location in memory where the object is stored (your memory locations will be different). If you see 10 of these, then your code is working properly.

## Interacting with the Objects

Now that there is a list of objects, we can manipulate each object by iterating through the list. We no longer need the print statement in our program. Replace it with a for loop. On each iteration, call the display method.

```
//add code below this line

ArrayList<App> apps = new ArrayList<App>();
String path = "code/advanced/app.csv";

try {
    CSVReader reader = new CSVReader(new FileReader(path));
    reader.skip(1);
    for (String[] row : reader) {
        apps.add(new App(row[0], row[1], row[2]));
    }
    reader.close();
} catch (Exception e) {
    System.out.println(e);
} finally {
    System.out.println("Finished reading a CSV");
}

for (App app : apps) {
    app.display();
}

//add code above this line
```

challenge

**Try these variations:**

- Call the `display` method on only the third app.

▼

### Solution

Normally, you would use a variable when instantiating an object. In this case, however, objects need to be referenced by the index in a list. Indexes start counting at 0, so the third element would be `.get(2)`:

```
//add code below this line

ArrayList<App> apps = new ArrayList<App>();
String path = "code/advanced/app.csv";

try {
    CSVReader reader = new CSVReader(new
        FileReader(path));
    reader.skip(1);
    for (String[] row : reader) {
        apps.add(new App(row[0], row[1], row[2]));
    }
    reader.close();
} catch (Exception e) {
    System.out.println(e);
} finally {
    System.out.println("Finished reading a CSV");
}

apps.get(2).display();

//add code above this line
```

- Call the `display` method for all objects that have “social media” as the `category` attribute.

▼

### Solution

Add the `getCategory` accessor method to the `App` class.

```
public String getCategory() {
    return category;
}
```

Iterate over the list and use a conditional to determine if the category attribute is “social media”. If true, call the display method.

```
//add code below this line

ArrayList<App> apps = new ArrayList<App>();
String path = "code/advanced/app.csv";

try {
    CSVReader reader = new CSVReader(new
        FileReader(path));
    reader.skip(1);
    for (String[] row : reader) {
        apps.add(new App(row[0], row[1], row[2]));
    }
    reader.close();
} catch (Exception e) {
    System.out.println(e);
} finally {
    System.out.println("Finished reading a CSV");
}

for (App app : apps) {
    if (app.getCategory().equals("social media")) {
        app.display();
    }
}

//add code above this line
```

# Composition

---

## Composition

Composition is a way to make a functional whole out of smaller parts. If you were to create a `Car` class, this would start out as a simple exercise. Every car has a make, a model, and a year it was produced. Representing this data is simple: two strings and an integer.

```
//add class definitions below this line

class Car {
    private String make;
    private String model;
    private int year;

    public Car(String ma, String mo, int y) {
        make = ma;
        model = mo;
        year = y;
    }

    public void describe() {
        System.out.println(String.format("%s %s %s", make, model,
            year));
    }
}

//add class definitions above this line
```

Create an instance of the `Car` class and call the `describe` method.

```
//add code below this line

Car car = new Car("De Tomaso", "Pantera", 1979);
car.describe();

//add code above this line
```

The `Car` class, however, is missing an important component: the engine. What data type would you use to represent an engine? Creating another class is the best way to do this. Modify the `Car` class so that it has an `engine` attribute and a getter for this attribute. Then create the `Engine` class with attributes for configuration (V8, V6, etc.), displacement, horsepower, and torque. Finally, add the `ignite` method to the `Engine` class.

```
//add class definitions below this line

class Car {
    private String make;
    private String model;
    private int year;
    private Engine engine;

    public Car(String ma, String mo, int y, Engine e) {
        make = ma;
        model = mo;
        year = y;
        engine = e;
    }

    public void describe() {
        System.out.println(String.format("%s %s %s", make, model,
            year));
    }

    public Engine getEngine() {
        return engine;
    }
}

class Engine {
    private String configuration;
    private double displacement;
    private int horsepower;
    private int torque;

    public Engine(String c, double d, int h, int t) {
        configuration = c;
        displacement = d;
        horsepower = h;
        torque = t;
    }

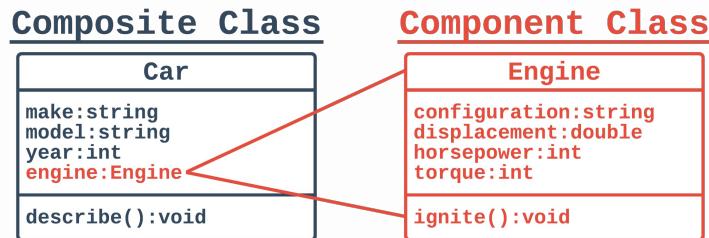
    public void ignite() {
        System.out.println("Vroom vroom!");
    }
}

//add class definitions above this line
```

Since the Car class takes an Engine object, instantiate an Engine object first. Then pass that object to the constructor for the Car class. To call the ignite method, you need to call the getter method for the engine attribute and then call the method.

```
//add code below this line  
  
Engine engine = new Engine("V8", 5.8, 326, 344);  
Car car = new Car("De Tomaso", "Pantera", 1979, engine);  
car.getEngine().ignite();  
  
//add code above this line
```

The combination of the Car class and the Engine class lead to a better representation of an actual car. This is the benefit of object composition. Because the Engine class is a part of the Car class, we can say that the Engine class is the component class and the Car class is the composite class.



.guides/img/advanced/composite\_component

challenge

## Try these variations:

- In the Car class, add the start method then have the last line of the script call start instead of ignite:

```
// Car class
public void start() {
    engine.ignite();
}

// main method
Engine engine = new Engine("V8", 5.8, 326, 344);
Car car = new Car("De Tomaso", "Pantera", 1979, engine);
car.start();
```

- In the Engine class, create the info method then call info instead of describe:

```
// Engine class
public void info() {
    describe();
}

// main method
Engine engine = new Engine("V8", 5.8, 326, 344);
Car car = new Car("De Tomaso", "Pantera", 1979, engine);
car.getEngine().info();
```

### ▼ Why is there an error?

Composition is a one-way street. The composite class (the Car class) has access to all of the attributes and methods of the component classes (the Engine class). However, component classes cannot access the attributes and methods of the composite class.

## Composition versus Inheritance

Assume you have the class MyClass. You want to use this class in your program, but it is missing some functionality. Do you use inheritance and extend the parent class, or do you use composition and create a component class? Both of these techniques can accomplish the same thing. This is a

complex topic, but you can use a simple test to help you decide. Use inheritance if there is an “is a” relationship, and use composition if there is a “has a” relationship.

For example, you have the `Vehicle` class and you want to make a `Car` class. Ask yourself if a car has a vehicle or if a car is a vehicle. A car is a vehicle; therefore you should use inheritance. Now imagine that you have a `Phone` class and you want to represent an app for the phone. Ask yourself if a phone is an app or if a phone has an app. A phone has an app; therefore you should use composition.

# Represent an Object as a String

---

## The `toString` Method

When you print out the instance of a user-created class, Java returns only the class name and its location in memory.

```
//add class definitions below this line

class Animal {
    private int age;

    public Animal(int a) {
        age = a;
    }
}

//add class definitions above this line
```

Instantiate an `Animal` object and print it.

```
//add code below this line

Animal a = new Animal(3);
System.out.println(a);

//add code above this line
```

This is not very helpful. That is why we have seen code examples where classes have a method called `describe` or `display` that print out a description of the object. However, a better way of representing an object as a string is to override the `toString` method. **Note**, it is not necessary to explicitly call the `toString` method. This is automatically done with `System.out.println`.

```
//add class definitions below this line

class Animal {
    private int age;

    public Animal(int a) {
        age = a;
    }

    public String toString() {
        return getClass().getName() + "[age=" + age + "]";
    }
}

//add class definitions above this line
```

## String Representation and Inheritance

The example above uses `getClass().getName()` to print the name of the class instead of manually printing `Animal`. This makes printing a string representation of a subclass much easier. Create the `Dog` class which extends the `Animal` class.

```

//add class definitions below this line

class Animal {
    private int age;

    public Animal(int a) {
        age = a;
    }

    public String toString() {
        return getClass().getName() + "[age=" + age + "]";
    }
}

class Dog extends Animal {
    private String name;
    private String breed;

    public Dog(String n, String b, int a) {
        super(a);
        name = n;
        breed = b;
    }
}

//add class definitions above this line

```

Change the object creation from an Animal to a Dog. When you run the program, Java will print Dog as the class name. However, it only prints out the age attribute but not the name or breed attributes. That is because `toString` is defined in Animal which does not have name or breed attributes.

```

//add code below this line

Dog d = new Dog("Rocky", "Pomeranian", 3);
System.out.println(d);

//add code above this line

```

To get a string representation of the Dog class, override the `toString` method. Call `toString` from the superclass and append the attributes (name and breed) from the subclass. You should see the name of the class (Dog), a set of square brackets with the attribute from the superclass, and another set of square brackets with attributes from the subclass.

```
class Dog extends Animal {
    private String name;
    private String breed;

    public Dog(String n, String b, int a) {
        super(a);
        name = n;
        breed = b;
    }

    public String toString() {
        return super.toString() + "[name= " + name + ", breed=" +
            breed + "]";
    }
}
```

# Interfaces

---

## Interfaces

Interfaces are similar to abstract classes in that they cannot be instantiated and methods must be defined by subclasses. However, interfaces force the user to implement (write code for) all of the methods. The Dog interface is defined in the IDE to the left. Notice that there is no access modifier for the bark method. Methods in an interface are designed to be used by other classes, so they are public by default. There is no need to add the public access modifier.

To use an interface, create a class with the `implements` keyword. The Chihuahua class **must** override the bark method. Be sure to use the `public` keyword so objects can call the method.

```
//add class definitions below this line

class Chihuahua implements Dog {
    public String bark() {
        return "woof woof";
    }
}

//add class definitions above this line
```

Instantiate a Chihuahua object and print the output of the bark method.

```
//add code below this line

Chihuahua c = new Chihuahua();
System.out.println(c.bark());

//add code above this line
```

challenge

## Try this variation:

- Comment out the definition for the bark method in the Chihuahua class.

```
//add class definitions below this line
```

```
class Chihuahua implements Dog {  
    // public String bark() {  
    //     return "woof woof";  
    // }  
}
```

```
//add class definitions above this line
```

### ▼ Why does this cause an error?

The Dog interface requires that classes override the bark method. Since the method is commented out, Java throws an error.

## Extending a Class and Implementing an Interface

Another benefit to interfaces is the ability to inherit from a class as well as implement an interface. For example, some people consider their pets to be a member of their family. Create the FamilyMember class with attributes for name and age and the info method. Then have the Chihuahua class extend the FamilyMember class and implement the Dog class.

```
//add class definitions below this line

class FamilyMember {
    private String name;
    private int age;

    public FamilyMember(String n, int a) {
        name = n;
        age = a;
    }

    public String info() {
        return String.format("%s is %d years old.", name, age);
    }
}

class Chihuahua extends FamilyMember implements Dog {
    public Chihuahua(String name, int age) {
        super(name, age);
    }

    public String bark() {
        return "woof woof";
    }
}

//add class definitions above this line
```

```
//add code below this line

Chihuahua c = new Chihuahua("Henry", 5);
System.out.println(c.bark());
System.out.println(c.info());
```

*//add code above this line*

challenge

## Try This Variation:

- Create the `movieStar` interface with the `movieDetails` method. Modify the `Chihuahua` class so that it extends the `FamilyMember` class and implements the `Dog` and `MovieStar` interfaces. Add the `film` and `revenue` attributes to help with overriding the `movieDetails`

method.

```
//add class definitions below this line

interface MovieStar {
    String movieDetails();
}

class FamilyMember {
    private String name;
    private int age;

    public FamilyMember(String n, int a) {
        name = n;
        age = a;
    }

    public String info() {
        return String.format("%s is %d years old.", name, age);
    }
}

class Chihuahua extends FamilyMember implements Dog,
    MovieStar {
    private String film;
    private String revenue;

    public Chihuahua(String n, int a, String f, String r) {
        super(n, a);
        film = f;
        revenue = r;
    }

    public String bark() {
        return "woof woof";
    }

    public String movieDetails() {
        return String.format("The move %s grossed %s
worldwide.", film, revenue);
    }
}

//add class definitions above this line
```

Change the instantiation of the Chihuahua object so that the new arguments are passed to the constructor. Then call the movieDetails method.

```

//add code below this line

Chihuahua c = new Chihuahua("Henry", 5, "Beverly Hills
    Chihuahua", "$149,281,606");
System.out.println(c.bark());
System.out.println(c.info());
System.out.println(c.movieDetails());

//add code above this line

```

## Interface vs Abstract Class

You may have noticed that there are many similarities between interfaces and abstract classes. The table below highlights some of the similarities and differences.

Category	Abstract Class	Interface
<b>Keyword:</b>	extends	implements
<b>Inheritance:</b>	Extend one superclass	Can extend a superclass and implement an interface
<b>Access Modifier:</b>	Use public, private, and/or abstract	All methods are public by default, no need to use an access modifier
<b>Implementation:</b>	Must override abstract methods	Must override <b>all</b> methods

With these concepts being so much alike, how do you know when to use one or the other? First, we need to understand two words: behavior and implementation. Behavior means the name but not the code of a method, while implementation means pre-written code for a method. Abstract classes allow for behavior (abstract methods) **and** implementation (concrete methods). Interfaces, however, only allow for behavior. You cannot create a concrete method in an interface. So if you want to define only behavior, use an interface. If you want to define behavior and implementation, use an abstract class.

# Object Equality

---

## Object Equality

The equality operator (==) is overloaded, which means it can compare two integers, two floats, etc. It can even compare two objects. Create the ExampleClass that has two attributes.

```
//add class definitions below this line

class ExampleClass {
    private int attribute1;
    private String attribute2;

    public ExampleClass(int a1, String a2) {
        attribute1 = a1;
        attribute2 = a2;
    }
}

//add class definitions above this line
```

Create the ExampleClass object example1 with 7 and "hello" as the attributes. Now make a copy of example1 and save it to the variable example2. Comparing the two objects with == should return true.

```
//add code below this line

ExampleClass example1 = new ExampleClass(7, "hello");
ExampleClass example2 = example1;
System.out.println(example1 == example2);

//add code above this line
```

Instead of making example2 a copy of example1, create a new object with the same values passed to the constructor. The program should now print false.

```

//add code below this line

ExampleClass example1 = new ExampleClass(7, "hello");
ExampleClass example2 = new ExampleClass(7, "hello");
System.out.println(example1 == example2);

//add code above this line

```

If the objects have the same type and the same values for their attributes, are they not the same? When comparing user-defined functions, the equality operator compares memory addresses. When `example2` is a shallow copy of `example1`, the two objects share the same memory address. When `example2` is created with the constructor, Java gives this object its own memory address.

Override the `equals` method if you want to compare attributes of different objects. If each attribute in one object is the same as the corresponding attribute in another object, the method should return `true`. Use a compound boolean expression to compare both attributes. Since `attribute1` is an integer, you can use `==` for the comparison. `attribute2` is a string, so you need to use the `equals` method.

```

//add class definitions below this line

class ExampleClass {
    private int attribute1;
    private String attribute2;

    public ExampleClass(int a1, String a2) {
        attribute1 = a1;
        attribute2 = a2;
    }

    public boolean equals(ExampleClass other) {
        return attribute1 == other.attribute1 &&
               attribute2.equals(other.attribute2);
    }
}

//add class definitions above this line

```

Finally, check for equality using the `equals` method. Java should return `true`.

```
//add code below this line

ExampleClass example1 = new ExampleClass(7, "hello");
ExampleClass example2 = new ExampleClass(7, "hello");
System.out.println(example1.equals(example2));

//add code above this line
```

challenge

## Try this variation:

- Change the values passed to the constructor for the example2 object.

```
//add code below this line

ExampleClass example1 = new ExampleClass(7, "hello");
ExampleClass example2 = new ExampleClass(-32,
    "goodbye");
System.out.println(example1.equals(example2));

//add code above this line
```

# **Advanced Topics Formative Assessment 1**

---

# **Advanced Topics Formative Assessment 2**

---

# **Learning Objectives - Casting**

---

**Learners will be able to...**

- **explain upcasting**
- **write code that performs upcasting**
- **explain downcasting**
- **write code that performs downcasting**

# Upcasting

Upcasting is when you cast an object to its superclass type. It is always safe and doesn't require an explicit cast.

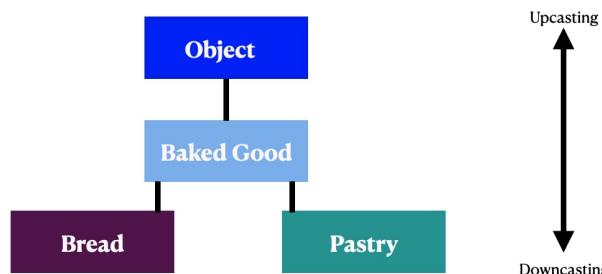


Image depicting classes derived from object and showing that upcasting is when you go up the inheritance tree and downcasting is when you go down

You can see an example of upcasting here:

We're taking a Bread object:

```
class Bread extends BakedGood {  
    public Bread(int bakingTemperature) {  
        super(bakingTemperature);  
    }  
    public void slice() {  
        System.out.println("Slicing the bread...");  
    }  
}
```

and treating it as its superclass type, BakedGood:

```
class BakedGood {  
    private int bakingTemperature;  
  
    public BakedGood(int bakingTemperature) {  
        this.bakingTemperature = bakingTemperature;  
    }  
  
    public void prepare() {  
        System.out.println("Preheat oven to " +  
bakingTemperature);  
    }  
}
```

### Try this:

- Add the code below to try to call a Bread method with that upcasted object
- ```
bg.slice();
```
- You get an error because the object that is a BakedGood doesn't know about the method slice.
  - You cannot call methods of the subclass when you have upcasted an object.
  - If you want to call a method of the subclass, you have to downcast

challenge

### Try this variation:

- Downcast the BakedGood object and try again. Replace the line you added with the following lines:

```
if (bg instanceof Bread){  
    ((Bread) bg).slice();  
}
```

- Now test it out again - it works!

---

info

## Downcasting

We'll learn more about this on the next page but you always want to test that an object is an instance of a class before downcasting.

# Downcasting

## Downcasting:

Downcasting is when you cast an object to one of its subclass/child types. This process is not always safe and requires an explicit cast.

### Correct downcasting

It first checks to see that the reference is an instance of the class it is being cast down to:

```
if (bg instanceof Bread)
```

### Incorrect downcasting

#### This will throw an exception!

```
BakedGood bg = new Bread(); // Here Bread is upcasted to  
BakedGood
```

- bg was upcasted to BakedGood but then below it's trying to downcast it to Pastry which it is not.
- It's not first checking to see if bg is an instance of Pastry (which it's not) so an error is generated when you try to downcast.

```
Pastry pastryItem = (Pastry) bg;
```

# **Formative Assessment 1 - Upcasting**

## **Formative Assessment 2 - Downcasting**

# **Learning Objectives**

**Learners will be able to...**

- **Create bounded and unbounded classes and methods**
- **Pass multiple unknown values to a generic**
- **Identify what to use (generic or wildcard) with collections**

# Limits of Typed Methods

## Many Similar Classes

Before we talk about generics, we are going to first recreate a situation where using generics provides a large benefit. To do this, we are going to create a very simple class that repeats an integer value. The behavior of the class is less important than the problems coming from a multitude of classes.

Let's start by create the `RepeatInt` class. It has one attribute, an integer, whose value is set with a constructor. There is one method that prints the value of the lone attribute in the class.

```
// add classes below this line
class RepeatInt {
    int repeatValue;

    public RepeatInt(int repeatValue) {
        this.repeatValue = repeatValue;
    }

    public void repeat() {
        System.out.println(this.repeatValue);
    }
}
// add classes above this line
```

We can now create the `main` method and instantiate a `RepeatInt` object and call its `repeat` method. Running the code should print 7.

```
// add main method below this line
public static void main (String[] args) {
    RepeatInt repeater = new RepeatInt(7);
    repeater.repeat();
}
// add main method above this line
```

If we find ourselves liking the functionality of the `RepeatInt` class but wanting to do something similar for a double, then we would have to create the `RepeatDouble` class. Add this class to the IDE on the left.

```

// add classes below this line
class RepeatInt {
    int repeatValue;

    public RepeatInt(int repeatValue) {
        this.repeatValue = repeatValue;
    }

    public void repeat() {
        System.out.println(this.repeatValue);
    }
}

class RepeatDouble {
    double repeatValue;

    public RepeatDouble(double repeatValue) {
        this.repeatValue = repeatValue;
    }

    public void repeat() {
        System.out.println(this.repeatValue);
    }
}

// add classes above this line

```

We can then use the `RepeatDouble` class in the `main` method. Update this method so it instantiates a `RepeatDouble` object and calls its `repeat` method. You should now see 7 and 3.25 as the output.

```

class Limits {
    // add main method below this line
    public static void main(String[] args) {
        RepeatInt repeater = new RepeatInt(7);
        repeater.repeat();

        RepeatDouble doubleRepeater = new RepeatDouble(3.25);
        doubleRepeater.repeat();
    }
    // add main method above this line
}

```

These classes are very simple, but you see how the size of our code is quickly growing. Imagine if we wanted to add similar functionality for strings, booleans, floats, and user-defined classes. Now imagine that each of these classes is sufficiently complex.

Our program will be extremely long by lines of code, but the differences between each of these classes is minimal. The only meaningful difference is the data type used in each class.

This is why Java has generics. It allows you to have a single class that works with many different data types.

# Generic Classes

## Generic Syntax

Unlike interfaces and abstract classes, you do not use a keyword to differentiate a generic class from a traditional class. Instead, use a type parameter. The type parameter should come after the class name and before the curly braces. The type parameter is used to signify the type used in the generic. Again, generics are designed to work with multiple data types, so the type parameter is not going to specify an exact data type. Instead it is going to use `<T>` to represent the unknown type.

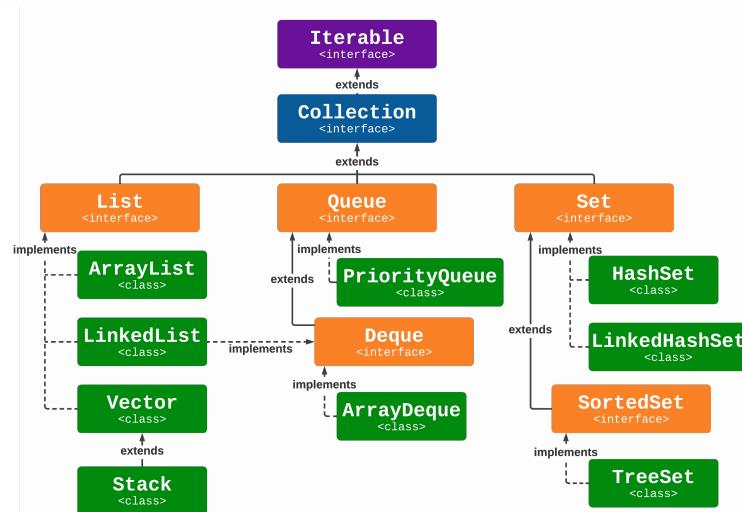
Generics enable classes and interfaces or types to be parameters when defining classes, interfaces and methods. A type variable can be any non-primitive type you specify: any class type, any interface type, any array type, or even another type variable.

By convention, type parameter names are single, uppercase letters. The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value

Code that uses generics:

- \* Have stronger type checks at compile time
- \* Allows elimination of casts
- \* Enables implementation of generic algorithms



The Iterable interface is extended by the Collection interface. The collection interface is extended by the List, Queue and Set interfaces.

Remove the RepeatDouble class. Then update the RepeatInt class to be named Repeater. Since we want this to be a generic class, use <T> as the type parameter. Just as our RepeatInt class was consistent in its use of integers, our Repeater generic must be consistent with the unknown type. Change the type of the repeatValue attribute to T. Note, the angle brackets are only used in the type parameter. All other references made to the unknown type are done without the angle brackets. Finally, change the type of the parameter for the constructor to T as well.

```
// add classes below this line
class Repeater<T> {
    T repeatValue;

    public Repeater(T repeatValue) {
        this.repeatValue = repeatValue;
    }

    public void repeat() {
        System.out.println(this.repeatValue);
    }
}
// add classes above this line
```

In the main method, instantiate a Repeater object. This is the moment where you are going to explicitly indicate the type of data you want to use with the generic. Since angle brackets are used with the type parameter, we are going to use angle brackets when stating the type during instantiation. Running the program should produce 7 as the output.

```
// add main method below this line
public static void main (String[] args) {
    Repeater<Integer> r1 = new Repeater<>(7);
    r1.repeat();
}
// add main method above this line
```

Because the Repeater class is now a generic, we can instantiate different objects, each with their own data type. All of this can be done without having to duplicate similar classes. Update the main method so there are three Repeater instances that work with integers, strings, and booleans.

```

// add main method below this line
public static void main (String[] args) {
    Repeater<Integer> r1 = new Repeater<>(7);
    Repeater<String> r2 = new Repeater<>("Generics");
    Repeater<Boolean> r3 = new Repeater<>(false);

    r1.repeat();
    r2.repeat();
    r3.repeat();
}

// add main method above this line

```

challenge

## Try these variations:

- Update the Repeater generic to use a different type parameter:

```

// add classes below this line
class Repeater<BlahBlahBlah> {
    BlahBlahBlah repeatValue;

    public Repeater(BlahBlahBlah repeatValue) {
        this.repeatValue = repeatValue;
    }

    public void repeat() {
        System.out.println(this.repeatValue);
    }
}

// add classes above this line

```

### ▼ What is happening?

Java does not dictate that a certain type parameter be used. You can use any valid identifier. However, you must be consistent in your use of the type parameter within the generic. <T> is a common convention, so we will use it for all other examples of generics.

- Update the instantiation to use a primitive:

```
// add main method below this line
public static void main (String[] args) {
    Repeater<int> r = new Repeater<>(42);
    r.repeat();
}
// add main method above this line
```

### ▼ What is there an error?

Generics do not work with primitive data types. If you want to use an integer, double, or any other primitive data type, be sure to use the wrapper class, i.e. Integer, Float, etc.

Generics are a compile-time feature, and uses something called type erasure. There is not data type called T so the compiler erases all instances of T and replaces them with type Object. Primitive data types are not a subclass of Object which is why Java throws an error.

## Familiar Syntax

If you are looking at the code above and think that you have seen this before, it is because you have. Here is how you declare two ArrayLists, one for integers and the other for strings:

```
ArrayList<Integer> integerList = new ArrayList<>();
ArrayList<String> stringList = new ArrayList<>();
```

ArrayLists are generics. That is why you are able to have the same data structure (class) work with a variety of data types. It is also why we have angle brackets, and why ArrayLists do not work with primitive data types.

# Generic Methods

## Adding Type Flexibility to Methods

Just as you can create a generic class, you can also create a generic method. Note, generic methods do not have to appear in generic classes. This flexibility is crucial when you wish to implement functionality that can operate across a diverse range of object types. Assume we want to make a method called echo. It takes a value of any type as well as an integer. The method prints out the value the same number of times as the integer parameter. Use `T` value to represent the value of an unknown type as a parameter. However, running your program like that would cause an error. We need to tell Java that `T` is a generic type, which means we need to use `<T>`. This time, however, the `<T>` is placed before the method's return type.

```
// add generic method below this line
private static <T> void echo(T type, int n) {
    for (int i = 0; i < n; i++) {
        System.out.print(type + " ");
    }
    System.out.println();
}
// add generic method above this line
```

Next, go into the `main` method and call the `echo` method a few times. Be sure to pass different data types as the first parameter.

```
// add main method below this line
public static void main (String[] args) {
    echo(1, 3);
    echo("dog", 7);
}
// add main method above this line
```

challenge

## Try these variations:

- Create the method `stringRepresentationLength` that uses the `Strings.valueOf` method to convert the parameter of any give type into a string and then returns the length of the string representation.

Test your method with the following `main` method:

```
// add main method below this line
public static void main (String[] args) {
    System.out.println(stringRepresentationLength(3.1459));
    System.out.println(stringRepresentationLength(false));
    System.out.println(stringRepresentationLength('x'));
}
// add main method above this line
```

You should see the following output:

```
tex -hide-clipboard 6 5 1
```

### ▼ Solution

Here is one possible solution:

```
private static <T> int stringRepresentationLength(T type)
{
    String stringRep = String.valueOf(type);
    return stringRep.length();
}
```

# Multiple Values

## Multiple Values in Classes

Using `<T>` to allow for an unknown type is quite useful, but you may find yourself needing more than unknown type for your generic class. Find the type parameter and add `V`. Be sure to use a comma to separate the values. Then add a second attribute to the class of type `V`. The constructor and the repeat method should also use the set and print this new attribute.

```
// add classes below this line
class Repeater<T, V> {
    T repeatValue1;
    V repeatValue2;

    public Repeater(T repeatValue1, V repeatValue2) {
        this.repeatValue1 = repeatValue1;
        this.repeatValue2 = repeatValue2;
    }

    public void repeat() {
        System.out.println(this.repeatValue1);
        System.out.println(this.repeatValue2);
    }
}
// add classes above this line
```

### ▼ Common type variables

When using multiple values in a generic there are common conventions used for the variables representing the different values:

- `T` – General type
- `E` – Element in a collection
- `K` – Key in a map
- `V` – Value in a map
- `S, U` - Additional general types

Again, these are just conventions. You can use any valid identifier to represent your type as long as you are consistent throughout your code. For this discussion, we will be using `<S>` for the second value.

In the `main` method, we are going to update our instantiation of the `Repeater` object. The class is expecting two unknown types, so we need two types between the angle brackets. In this case, use `Boolean` and `String`. Then pass a boolean and a string to the constructor.

```
// add main method below this line
public static void main (String[] args) {
    Repeater<Boolean, String> r1 = new Repeater<>(false, "Dog");
    r1.repeat();
}
// add main method above this line
```

challenge

## Try these variations:

- Change the instantiation of the `Repeater` object so that two integers are passed to the constructor:

```
// add main method below this line
public static void main (String[] args) {
    Repeater<Integer, Integer> r1 = new Repeater<>(3325,
        42);
    r1.repeat();
}
// add main method above this line
```

### ▼ What is happening?

Even though the type parameters are `<T, S>`, that does not mean that you must pass two values of different types to the constructor.

- Update the `Repeater` class so that it takes two unknown values both of type `T`:

```
// add classes below this line
class Repeater<T, T> {
    T repeatValue1;
    T repeatValue2;

    public Repeater(T repeatValue1, T repeatValue2) {
        this.repeatValue1 = repeatValue1;
        this.repeatValue2 = repeatValue2;
    }

    public void repeat() {
        System.out.println(this.repeatValue1);
        System.out.println(this.repeatValue2);
    }
}
// add classes above this line
```

### ▼ Why is there an error?

Java will not let you reuse the same identifier for a type parameter. If you want multiple unknown types, you must use different identifiers for each.

- Update the Repeater class so that it takes and repeats three unknown values.

### ▼ Solution

Here is one possible solution:

```

// add classes below this line

class Repeater<T, S, U> {
    T repeatValue1;
    S repeatValue2;
    U repeatValue3;

    public Repeater(T repeatValue1, S repeatValue2, U
                    repeatValue3) {
        this.repeatValue1 = repeatValue1;
        this.repeatValue2 = repeatValue2;
        this.repeatValue3 = repeatValue3;
    }

    public void repeat() {
        System.out.println(this.repeatValue1);
        System.out.println(this.repeatValue2);
        System.out.println(this.repeatValue3);
    }
}

// add classes above this line

class MultipleValues {

    // add generic method below this line

    // add generic method above this line

    // add main method below this line
    public static void main (String[] args) {
        Repeater<Boolean, String, Double> r1 = new Repeater<>(
            false, "Dog", 3.1459);
        r1.repeat();
    }
}

// add main method above this line
}

```

## Multiple Values in Methods

Multiple unknown values can also be used with generic methods. Add the generic method echo to the IDE. This method takes two unknown values and an integer. For each unknown value, the method loops *n* number of times, printing the unknown value.

```
// add generic method below this line
private static <T, S> void echo(T type, S value, int n) {
    for (int i = 0; i < n; i++) {
        System.out.print(type + " ");
    }
    System.out.println();
    for (int i = 0; i < n; i++) {
        System.out.print(value + " ");
    }
    System.out.println();
}
// add generic method above this line
```

Update the `main` method to call the `echo` method twice, each time with different arguments.

```
// add main method below this line
public static void main (String[] args) {
    echo(true, 5.6789, 4);
    echo("Dog", 42, 2);
}
// add main method above this line
```

# Bounded Generics

## Limiting Types for Generics

Let's assume we are going to create a wrapper class that takes an unknown type, and calls some methods on the type. For this exercise, we referencing the `MobileApp` and `DesktopApp` classes which inherit from the `Application` class.

### ▼ Class code

If you are interested in the code for the `Application`, `DesktopApp`, and `MobileApp` classes, see below. These are skeleton classes with a few attributes and some getters.

### Application Class

```

public class Application {
    private String name;
    private String developer;
    private String operatingSystem;

    public Application (String n, String d, String os) {
        name = n;
        developer = d;
        operatingSystem = os;
    }

    public String getName() {
        return name;
    }

    public String getDeveloper() {
        return developer;
    }

    public String getOperatingSystem() {
        return operatingSystem;
    }

    public String interact() {
        return "Click or tap on the application";
    }
}

```

## DesktopApp Class

```

public class DesktopApp extends Application {

    public DesktopApp(String n, String d, String os) {
        super(n, d, os);
    }

    public String interact() {
        return "Click, double-click, or right-click";
    }
}

```

## MobileApp Class

```

public class MobileApp extends Application {

    public MobileApp(String n, String d, String os) {
        super(n, d, os);
    }

    public String interact() {
        return "Tap, swipe left, or swipe right";
    }

    public String confirmMobile() {
        return "This is a mobile app";
    }
}

```

We are going to add the `printInfo` method that prints out information in a more friendly manner.

```

// add classes below this line

class Wrapper <T> {
    T type;

    public Wrapper(T type) {
        this.type = type;
    }

    public void printInfo() {
        System.out.println("Application name: " + type.getName());
        System.out.println("Application developer: " +
                           type.getDeveloper());
        System.out.println("Application OS: " +
                           type.getOperatingSystem());
        System.out.print("Application interactions: ");
        System.out.println(type.interact());
    }
}

// add classes above this line

```

Modify the `main` method such that we create two instances of the `Wrapper` class, one based on the `DesktopApp` class and the other based on the `MobileApp` class.

```

// add main method below this line
public static void main(String args[]) {
    Wrapper<DesktopApp> be1 = new Wrapper<>(new
        DesktopApp("Lutris", "Mathieu Comandon", "Linux"));
    Wrapper<MobileApp> be2 = new Wrapper<>(new
        MobileApp("Overcast", "Marco Arment", "iOS"));

    be1.printInfo();
    System.out.println("-----");
    be2.printInfo();
}
// add main method above this line

```

Click the button below to run your program. You should see several errors.

Why would Java throw these error messages? Recall that Java is a statically-typed language. The idea behind this is type safety. That is, you are not performing actions that would otherwise cause an error. For example, you can use `.length` with an array, but you cannot use it with an integer.

Using generics means we can pass an unknown data type to a class or method. In the example above, what would happen if we were to pass an `Integer` or a `String` to the `Wrapper` class? Do those data types have methods like `getName` or `getDeveloper`? No, they don't.

Java introduced bounded generics so that we can limit the types of data that work with generics. This way, we can guarantee type safety and still get type flexibility. As mentioned above, the `DesktopApp` and `MobileApp` class inherit from the `Application` class. Using the `extends` keyword, we want to limit the types of data that work with the `Wrapper` class to be instances of the `Application` class (this includes subclasses).

```
class Wrapper <T extends Application> {
```

Run the program again. Everything should work this time. Because we are setting boundaries for the types of data the `Wrapper` class accepts, we can guarantee that each object has the `getName`, `getDeveloper`, `getOperatingSystem` and `interact` methods.

challenge

## Try this variation:

Update the `printInfo` method to call `confirmMobile`.

```
public void printInfo() {  
    System.out.println("Application name: " +  
        type.getName());  
    System.out.println("Application developer: " +  
        type.getDeveloper());  
    System.out.println("Application OS: " +  
        type.getOperatingSystem());  
    System.out.print("Application interactions: ");  
    System.out.println(type.interact());  
    System.out.println(type.confirmMobile());  
}
```

### ▼ Why is there an error?

The `confirmMobile` method is only found in the `MobileApp` class. It is not in the `Application` or `DesktopApp` classes. Even though your generic is bounded, that does not mean you can call any method. The methods invoked must be found in the superclass and all subclasses.

# Interfaces as Boundaries

## Using an Interface as a Boundary

Just as we can set boundaries for a generic with inheritance, we can also use interfaces. However, you do not use the `implements` keyword even though you are working with an interface. Instead, you use `extends`.

You have already been given the `Seniority` interface, which requires the overriding of the `calculateSeniority` method. Basically, we need to calculate the difference between the start date of an employee and today's date. This gives us the seniority of an employee. Let's start by creating the `Employee` class that implements the `Seniority` interface. There are two attributes (both of type `LocalDate`) which represent the date of hire and today's date. The start date is passed to the constructor, while `LocalDate.now` returns the date when the code is executed. Using `Period.between` we can calculate the amount of time that has passed between these two dates.

```
// add classes below this line
class Employee implements Seniority {
    private LocalDate startDate;
    private LocalDate today;

    public Employee(String s) {
        startDate = LocalDate.parse(s);
        today = LocalDate.now();
    }

    public Period calculateSeniority() {
        Period duration = Period.between(startDate, today);
        return duration;
    }
}
// add classes above this line
```

### ▼ Working with time

This example works with time, which is why we imported the `time` package. This is not a formal introduction to time, but you can find more information about the objects and methods used with the links below:

- [Period](#)
- [LocalDate](#)

Next, create the `printSeniority` generic method. This method accepts an unknown data type bound by the `Seniority` interface. Call the `calculateSeniority` method which returns a `Period` object. This object contains the number of years, months, and days between the two dates. We can access this data with some getter methods. Lastly, print out a sentence that provides context to the duration.

```
// add generic method below this line
public static <T extends Seniority> void printSeniority(T
    employee) {
    Period duration = employee.calculateSeniority();
    int years = duration.getYears();
    int months = duration.getMonths();
    int days = duration.getDays();
    String sentence = String.format(
        "They have worked here for %d year(s), %d month(s), and %d
        day(s).",
        years, months, days);
    System.out.println(sentence);
}
// add generic method above this line
```

The final step is to add the `main` method. Instantiate an `Employee` object, and pass it the date of hire for the employee. Java expects the date to be a string in the format YY-MM-DD (year, month, day). Pass the `Employee` object to the `printSeniority` method.

```
// add main method below this line
public static void main(String[] args) {
    Employee e = new Employee("2019-08-19");
    printSeniority(e);
}
// add main method above this line
```

## Multiple Bounds

You can add more than one boundary to a generic. This allows us to further narrow down the scope of types that can work with the generic method or class. Let's update the type signature of the `printSeniority` method. We want it to accept only types that inherit from `Employee` and implement the `Seniority` interface. Use a `&` to separate the class and the interface in the type parameter.

```
public static <T extends Employee & Seniority> void  
printSeniority(T employee) {
```

The program should run just as before. Functionally speaking, this change had no real effect on the code.

challenge

## Try this variation:

Change the order of the superclass and interface in the type parameter:

```
public static <T extends Seniority & Employee> void  
printSeniority(T employee) {
```

### ▼ Why is there an error?

When you have multiple boundaries, you must put the superclass first and then list any interfaces in the type parameter. Be sure to correct the order of the type parameter before continuing:

```
public static <T extends Employee & Seniority> void  
printSeniority(T employee) {
```

However, let's introduce another class that implements the `Seniority` interface. Create the `Employer` class that overrides the `calculateSeniority` method. We are going to keep this simple, so return a `Period` object of 1 year, 1 month, and 1 day.

```
class Employer implements Seniority {  
    public Period calculateSeniority() {  
        return Period.of(1, 1, 1);  
    }  
}
```

Now update the `main` method to instantiate an `Employer` object and pass it to the `printSeniority` method.

```
// add main method below this line
public static void main(String[] args) {
    Employer e = new Employer();
    printSeniority(e);
}
// add main method above this line
```

This time we should see an error. The `Employer` class implements the `Seniority` interface, but it is *not* a subclass of the `Employee` class. Because this does not meet the requirements of the type parameter, Java throws an error.

Java does not support multiple inheritance, so you cannot set boundaries for a generic to be two or more classes. You can only extend one class. However, you can extend a class and implement more than one interface when declaring the type parameter. You can also just implement several interfaces. Be sure to use a `&` to separate the superclass and any interfaces in the type parameter.

```
class MyClass <T extends Superclass & Interface1 & Interface2 &
Interface3>
```

# Wildcards

## Unknown Values

So far, we have used generics to create classes and methods that take unknown values. Java also allows you to use the ? symbol as a wildcard. For example, assume you have the method below. All the method does is print out the elements in a list.

```
// add method below this line
public static <E> void printList(List<E> list) {
    for (int i = 0; i < list.size(); i++) {
        System.out.println(list.get(i));
    }
}
// add method above this line
```

In the main method, create a list of integers and pass it to the `printList` method.

```
// add main method below this line
public static void main(String[] args) {
    List<Integer> list = Arrays.asList(1, 2, 3, 4);
    printList(list);
}
// add main method above this line
```

We can rewrite the method with a wildcard. Wild cards do not need a type parameter. We can also replace `List<E>` with `List<?>`. Running the program will work and produce the same results as the generic method.

```
public static void printList(List<?> list) {
    for (int i = 0; i < list.size(); i++) {
        System.out.println(list.get(i));
    }
}
```

So, does this mean we should use wildcards as opposed to generics? Relying solely on wild card can cause problems. Let's take a look at the `reverseList` method. This method will not compile.

```
public static void reverseList(List<?> list) {  
    List<?> newList = new ArrayList<?>();  
    for (int i = list.size() - 1; i >= 0; i--) {  
        newList.add(list.get(i));  
    }  
}
```

In particular, pay attention to the line of that says `newList.add(list.get(i));`. Wildcards are used for both `list` and `newList`, however there is no concrete link between the two data types. That means you could add an invalid data type from `list` to `newList`. Therefore Java does not allow this.

Take a look at the same method that uses generics instead of wildcards:

```
public static <E> void reverseList(List<E> list) {  
    List<E> newList = new ArrayList<?>();  
    for (int i = list.size() - 1; i >= 0; i--) {  
        newList.add(list.get(i));  
    }  
}
```

This is valid Java code because both `list` and `newList` have the data type `E`. This type may be unknown, but it will be the same for both lists. So while wildcards can be used to represent unknown data types, they cannot always be used in place of a generic.

## Bounded Wildcards

Using just a `?` is an unbounded wildcard. That means, it can be any value at all. Similar to generics, you can set boundaries on wildcards. You can even set boundaries that do not exist with generics.

### Upper Bounded Wildcards

Upper bounded wildcards work exactly like bounded generics. That is you use the `extends` keyword and then indicate a class. This means you can use any instances of the listed class or any of its subclasses. By using `extends`, you are setting the upper limit. That is, you can not use a data type that is a superclass of the extended class.

For example, let's look at the `printList` method. It takes a list of an unknown type (as long as the type is a `Number` or one of its subclasses) and prints out each element in the list.

```
// add method below this line
public static void printList(List<? extends Number> list) {
    for (int i = 0; i < list.size(); i++) {
        System.out.println(list.get(i));
    }
}
// add method above this line
```

Modify the `main` method to instantiate a list of integers and a list of doubles. Pass both of them to the `printList` method. Everything should work as expected.

```
// add main method below this line
public static void main(String[] args) {
    List<Integer> ints = Arrays.asList(1, 2, 3, 4);
    List<Double> dbls = Arrays.asList(5.0, 6.0, 7.0, 8.0);
    printList(ints);
    printList(dbls);
}
// add main method above this line
```

However, if we change the `main` method to create a list of type `Object`, passing this list to the `printList` method will cause an error. The `Object` class is the superclass of `Number`, which violates the upper boundary of the wildcard.

```
// add main method below this line
public static void main(String[] args) {
    List<Object> objs = new ArrayList<>();
    objs.add("y");
    objs.add(7);
    printList(objs);
}
// add main method above this line
```

## Lower Bounded Wildcards

You can also set the lower boundary for a wildcard with the `super` keyword. Update the `printList` method so that `super` replaces `extends`. Recall that our previous code example would not work because `Object` is a superclass to `Number`. Since the lower boundary is now `Number`, we should now be able to print a list of type `Object`.

```
// add method below this line
public static void printList(List<? super Number> list) {
    for (int i = 0; i < list.size(); i++) {
        System.out.println(list.get(i));
    }
}
// add method above this line
```

challenge

## Try this variation:

Update the `main` method so that a list of integers is passed to the `printList` method.

```
// add main method below this line
public static void main(String[] args) {
    List<Integer> ints = Arrays.asList(1, 2, 3, 4);
    printList(ints);
}
// add main method above this line
```

### ▼ Why is there an error?

The class `Integer` is a subclass of `Number`. Because we set the lower boundary to be `Number`, Java will only accept values of type `Number` or its superclass.

# Wildcards and Collections

## Producer Extend, Consumer Super (PECS)

When working with collections you need to understand when to use a wildcard, what kind of wildcard to use, and when to use a generic. The mnemonic PECS will help us decide what to use. PECS stands for “Producer extend, consumer super”. It is important to understand that the acts of producing and consuming are defined from the collection’s point of view.

### Producing

When we say producing, we mean that the collection is producing some kind of action or end result that we want. As an example, let’s create the `total` method which takes a list of numbers (the literal `Number` class) and returns the sum of the list as an integer. We can say that the elements of the list are used to produce the total. According to PECS, we need to use the `extends` keyword for the wildcard.

```
// add method below this line
public static int total(List<? extends Number> numbers) {
    int sum = 0;
    for (Number number : numbers) {
        sum += number.intValue();
    }
    return sum;
}
// add method above this line
```

Now create the `main` method in which you create a two lists, one of integers and another of doubles. Pass both lists to the `total` method. Running this code should produce an error.

```

// add main method below this line
public static void main(String[] args) {
    List<Integer> list1 = Arrays.asList(1, 2, 3, 4);
    int sum = total(list1);
    System.out.println(sum);

    List<Double> list2 = Arrays.asList(2.1, 3.1, 4.1, 5.1);
    sum = total(list2);
    System.out.println(sum);
}
// add main method above this line

```

## Try this variation:

Rewrite the `total` method so that it works only with a list of type `Number`. Notice how Java throws an error because lists of type `Integer` and `Double` cannot be converted to type `Number`.

```

// add method below this line
public static int total(List<Number> numbers) {
    int sum = 0;
    for (Number number : numbers) {
        sum += number.intValue();
    }
    return sum;
}
// add method above this line

```

## Consuming

When the collection is consuming, that means an element is being added to the collection. According to PECS, we should use the `super` keyword. Create the `appendNumber` method that takes a list of integers or its superclass.

```

// add method below this line
public static void appendNumber(List<? super Integer> numbers)
{
    numbers.add(42);
}
// add method above this line

```

Modify the `main` method so that you create a list of integers and a list of numbers (both integers and doubles). Pass both of these lists to the `appendNumber` method and print the lists.

```
// add main method below this line
public static void main(String[] args) {
    List<Integer> intList = new ArrayList<>(Arrays.asList(1, 2,
        3, 4));
    appendNumber(intList);
    System.out.println(intList);

    List<Number> numList = new ArrayList<>(Arrays.asList(3325,
        3.1459));
    appendNumber(numList);
    System.out.println(numList);
}
// add main method above this line
```

## Try this variation:

Change the list type from the superclass of `Integer` to just `Integer`. Notice that Java throws an error because it cannot convert a list of type `Number` to a list of type `Integer`.

```
// add method below this line
public static void appendNumber(List<Integer> numbers) {
    numbers.add(42);
}
// add method above this line
```

## Producing and Consuming

Let's create the method `appendTotal` which takes a list of integers, calculates the sum of the list, and then appends the sum to the end of the list. The collection is both a producer and a consumer. According to PECS we should use both of the `extends` and `super` keywords. What does this mean exactly? If we extend an integer, that means integers and the subclasses are allowed, but the superclasses are not. The `super` keyword means the superclasses are allowed, but the subclasses are not. In effect, `super` and `extends` cancel each other out. All we are left with is the `Integer` type itself.

When a collection is both a producer and a consumer, you should use an unbounded generic for the collection type. For the appendTotal method, have it take a list of integers. Then calculate the sum of the list and append it to the list itself.

```
// add method below this line
public static void appendTotal(List<Integer> numbers) {
    int total = 0;
    for (Integer number : numbers) {
        total += number;
    }
    numbers.add(total);
}
// add method above this line
```

Update the main method so that you create a list of integers. Pass the list to appendTotal, and then print out the list.

```
// add main method below this line
public static void main(String[] args) {
    List<Integer> intList = new ArrayList<>(Arrays.asList(1, 2,
            3, 4));
    appendTotal(intList);
    System.out.println(intList);
}
// add main method above this line
```

## Try this variation:

Change the parameter for appendTotal from an unbound generic to an unbound wildcard. Notice that Java throws an error because it cannot convert a list of unknown type to a list of type integer.

```
// add method below this line
public static void appendTotal(List<?> numbers) {
    int total = 0;
    for (Integer number : numbers) {
        total += number;
    }
    numbers.add(total);
}
// add method above this line
```

## Independence

You may find that a collection is neither producing nor consuming. The collection itself is completely independent of the actions being taken. PECS does not really help us determine what should be used. However, because the collection is not producing or consuming, then we do not need to place any restrictions upon it. A generic wildcard works best.

The `echoList` method take a list of any type at all and prints it. That's it. None of the elements are accessed, the list is not altered in any way. The list is neither producing nor consuming.

```
// add method below this line
public static void echoList(List<?> list) {
    System.out.println(list);
}
// add method above this line
```

Change the `main` method so that we have three lists of different data types. Pass each of them to the `echoList` method.

```
// add main method below this line
public static void main(String[] args) {
    List<Integer> intList = new ArrayList<>(Arrays.asList(1, 2,
        3, 4));
    List<Boolean> boolList = new ArrayList<>(Arrays.asList(true,
        false, false, true));
    List<String> strList = new ArrayList<>(Arrays.asList("cat",
        "dog", "fish"));

    echoList(intList);
    echoList(boolList);
    echoList(strList);
}
// add main method above this line
```

## Try this variation:

Change the parameter for echoList from an unbound wildcard to an unbound generic of type Object. Notice that Java throws an error because it cannot convert lists of type Integer, Boolean, and String to lists of type Object.

```
// add method below this line
public static void echoList(List<Object> list) {
    System.out.println(list);
}
// add method above this line
```

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives - Advanced Topics**

**Learners will be able to...**

- **Understand and Define Abstract Data Types (ADTs)**
- **Create and Implement Abstract Data Types using Java**
- **Distinguish between Concrete and Abstract Data Types**
- **Implement basic operations on ADTs like insertion, deletion, and traversal**

# Understanding Abstract Data Types

An **Abstract Data Type (ADT)** is a high-level description of a collection of data and the operations that can be performed on that data. It does not specify how the data will be organized in memory or what algorithms will be used for implementing the operations. It is called “abstract” because it gives an abstraction of data and does not concern itself with the implementation details.

## Components:

An ADT has two main components:

1. **Data:** It represents the values that are to be stored.
2. **Operations:** They are the procedures or functions performed on the data, such as unique ways to perform insertion, deletion, modification, etc.

## Example:

Consider a List ADT.

- \* **Data:** It stores elements in a sequence, allowing duplicates.
- \* **Operations:**
  - \* Add an element.
  - \* Remove an element.
  - \* Retrieve an element at a specific position.
  - \* Find the size of the list.

More examples of ADTs include String , Stack, Queue, Tree, Graph, and Set. Each of these defines a set of operations that can be performed on it, such as insertion, deletion, and traversal, but does not provide any implementation details about how these operations are carried out.

## Why Abstract Data Types?

- **Encapsulation:** ADTs encapsulate the data and the operations that operate on them, allowing for a modular approach to programming.
- **Abstraction:** They provide a clear separation between the abstract properties of a data type and the concrete details of its implementation.
- **Reusability:** The implementation of ADTs can be reused across different applications and programs, improving software design and reducing development time.

## **Abstract Classes and Interfaces in Java:**

Understanding Abstract Data Types, abstract classes, and interfaces is crucial for designing robust, modular, and reusable software. By defining the structure and operations of data separately from the implementation details, ADTs enable developers to focus on solving problems at a higher, more abstract level, promoting clear thinking and effective problem-solving in software development. Abstract Data Types can be represented using abstract classes and interfaces.

### **Abstract Classes:**

- An abstract class in Java is a class that cannot be instantiated, but it can have fields, methods, and constructors.
- It can have both abstract (without implementation) and concrete (with implementation) methods.
- It allows for code inheritance, as a subclass can inherit the characteristics of the abstract class.

```
abstract class Shape {  
    abstract void draw();  
    void display() {  
        System.out.println("Displaying shape.");  
    }  
}
```

### **Interfaces:**

- An interface in Java is a completely abstract type that can include abstract methods and constants.
- All methods in an interface are implicitly public and abstract.
- A class can implement multiple interfaces, enabling Java to compensate for the absence of multiple inheritances.

```
interface Drawable {  
    void draw();  
}
```

### **Usage:**

- **Abstract Classes:** Use when you want to share code among several closely related classes, and when you expect classes that extend your abstract class to have many common methods or fields.
- **Interfaces:** Use when you want to define a contract for classes to implement, allowing for a modular and loosely coupled design, especially when you expect the classes to implement multiple

interfaces.

# String ADT

A **String ADT** is an Abstract Data Type representing a sequence of characters. It is one of the most fundamental and widely used ADTs in computer science. The String ADT defines a set of operations that can be performed on the sequence of characters without revealing the internal structure and implementation details of the string.

## Components:

Understanding the String ADT is crucial as strings are pervasive in programming. Being able to manipulate and operate on strings efficiently is a key skill in software development, whether it is for processing user input, managing file paths, or handling textual data in various encoding formats.

### #### Data:

- \* A sequence of characters.

## Operations:

Some of the essential operations that can be performed on a String ADT include:

- \* **Concatenation:** Appending one string to the end of another.
- \* **Substring:** Extracting a sequence of characters from a string.
- \* **Comparison:** Comparing two strings for equality or lexicographical order.
- \* **Length:** Finding the number of characters in a string.
- \* **Character At:** Retrieving the character at a specific position.
- \* **Index Of:** Finding the position of a character or substring within a string.

## String ADT in Java:

In Java, the `String` class provides an implementation of the String ADT. The `String` class is immutable, which means once a `String` object is created, it cannot be changed. If any operation appears to change the `String`, a new `String` object is created.

```

public class StringExample {
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "World";

        // Concatenation
        String concatenatedString = str1.concat(str2);
        System.out.println("Concatenated String: " +
                           concatenatedString);

        // Substring
        String substring = str1.substring(1, 4); // "ell"
        System.out.println("Substring: " + substring);

        // Comparison
        boolean isEqual = str1.equals("Hello"); // true
        System.out.println("Strings are equal: " + isEqual);

        // Length
        int length = str1.length(); // 5
        System.out.println("Length of String: " + length);

        // Character At
        char charAt = str1.charAt(1); // 'e'
        System.out.println("Character at index 1: " + charAt);

        // Index Of
        int indexOf = str1.indexOf('l'); // 2
        System.out.println("Index of character 'l': " +
                           indexOf);
    }
}

```

The String Abstract Data Type is fundamental in computer science, representing sequences of characters and providing various operations to manipulate them. The String class offers a robust and comprehensive implementation of this ADT, with a variety of methods that allow developers to perform common string operations efficiently and effectively. Familiarity with String ADT and the String class in Java is essential for anyone aiming to develop software in a modern programming environment.

# More on String ADT

## Immutable Nature

In Java, String objects are immutable, meaning once a String object is created, it cannot be altered. Any operation that seems to modify the string instead creates a new String object.

```
String str = "Hello";
str = str + " World"; // Creates a new String object
```

This immutable nature helps in memory efficiency and security but can lead to overhead when numerous modifications are needed, leading to the creation of many temporary objects.

## Operations on String ADT

### 1. Concatenation:

This operation combines two strings to form a new one.

```
java String str1 = "Java"; String str2 = "Programming";
String result = str1.concat(str2); // "JavaProgramming"
System.out.println(result);
```

### 2. Substring:

This operation extracts a sequence of characters from the string.

```
java String str = "Computer"; String sub = str.substring(1,4);
// "omp" System.out.println(sub);
```

### 3. Search:

This operation determines if a character or substring exists within a string and returns the index.

```
java String str = "Introduction"; int index =
str.indexOf("tro"); // 3 System.out.println(index);
```

### 4. Comparison:

It allows the comparison of two strings, determining their lexicographical ordering.

```
java String str1 = "Apple"; String str2 = "Orange"; int
result = str1.compareTo(str2); // -14 (Negative because "Apple" is
lexicographically less than "Orange") System.out.println(result);
```

## 5. Length:

This operation retrieves the number of characters in a string.

```
java String str = "Hello"; int len = str.length(); // 5
System.out.println(len);
```

## Memory Efficiency: StringBuilder and StringBuffer

Given the immutability of String objects, when multiple modifications are required, it is efficient to use `StringBuilder` or `StringBuffer` classes. Both are mutable sequences of characters, but `StringBuffer` is thread-safe, ensuring synchronized access.

```
StringBuilder builder = new StringBuilder("Hello");
builder.append(" World"); // Efficiently modifies the existing
                         object
System.out.println(builder);
```

## Use Cases

String ADT is ubiquitous in software development. It is extensively used in:

1. **Text Processing and Analysis:** String ADT is vital in reading, analyzing, and processing textual data, used in natural language processing, search engines, etc.
2. **Pattern Matching:** It is fundamental in developing algorithms for matching and searching patterns in textual data, used in data mining, bioinformatics, etc.
3. **Formatting Outputs:** String ADT aids in formatting textual output in console applications, web applications, and more.

# Formative Assessment 1

Below is a piece of Java code that incorporates concepts from the lessons on ADTs, particularly focusing on the String ADT, and its related operations. However, the code contains several errors. Your task is to identify these mistakes and fix them. When fixing them think of why they are considered errors based on your understanding of Java strings, ADTs, and associated operations.

```
public class TestStringOperations {
    public static void main(String[] args) {

        // Intended to create a new string that appends " World"
        // to "Hello"
        String greeting = "Hello";
        greeting.concat(" World"); // Error

        // Intended to extract a substring "Computer" from
        // "SuperComputer" starting from index 5
        String fullWord = "SuperComputer";
        String partWord = fullword.substring(5); // Error

        // Intended to compare two strings lexicographically
        String first = "Java";
        String second = "Kotlin";
        int comparisonResult;
        comparisonResult = first.compareTo(second); // Error

        // Intended to create a mutable sequence of characters
        // and append a string to it
        StringBuilder builder;
        builder = StringBuilder("Start"); // Error
        builder.append("End"); // Error
    }
}
```

After fixing your code , the code should output the following :

```
Hello World
Computer
-1
StartEnd
```

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- **Define a data structure**
  - **Cover Basic operations of data structures**
  - **Perform Basic Operations with arrays**
  - **Differentiate Linear and Non-Linear Data Structures**
-

# Data Structures

## What is a Data Structure?

Let's break down the two fundamental words in **Data Structures**: Data and Structure.

**Data:** In the simplest terms, data is information. In the context of computer science, data refers to the quantities, characters, or symbols on which operations are performed by a computer. Data can exist in various forms - it could be numeric values, text, images, audio, video, and even complex structured types like lists, arrays, and so forth.

**Structure:** Structure implies the arrangement or organization of components. In the world of computer science, when we refer to the structure, we're talking about how we are organizing or arranging our data in the computer's memory for efficient access and modification.

definition

**Data Structures** are particular ways of organizing data in a computer so that we can perform different operations. Some of those basic operations include:

- \* Insertion
- \* Deletion
- \* Merging
- \* Searching
- \* Sorting
- \* Traversal

We can categorize data types into two categories:

- \* **Primitive data types**
- \* **Non-primitive data types**

The primitive type of data types includes the predefined data structures such as char, float, int, and double. Non-Primitive data types are composed of primitive data types.

Simple data structures are types of data structures that are generally built from primitive data types like int, float, double, string, char.

Compound data structures are types that the user can build by combining simple data structures.

The non-primitive data structures are used to store the collection of elements. This data structure can be further categorized into **Linear and Non-Linear data structures**.

**Linear data structure** – It is a type of data structure where the arrangement of the data follows a linear fashion. It is a type of data structure where data is stored and managed in a linear sequence. Some of the popular linear data structures that we widely use in Java are arrays, stacks, queues, and linked lists

**Non-Linear data Structures** – It is a type of data structures are basically multilevel data structures. the data are arranged in a manner that does not follow a linear fashion. Some of the popular non-linear data structures that we widely use are trees and graphs.

# Algorithms

## Algorithms

Now that we are more or less familiar with Data structures bit of “data structures and algorithms”. We can tackle the question “what is an algorithm?”

### definition

An **algorithm** is a step-by-step procedure or a set of rules to be followed in calculations or other problem-solving operations, especially by a computer. In simpler terms, it's a recipe for accomplishing a task.

For example, an algorithm could be a process for sorting a list of names alphabetically, calculating the square root of a number, finding the shortest path in a graph, or any other sequence of operations that can be programmed and implemented.

The study of algorithms and data structures go hand in hand. Data structures are often designed to facilitate specific algorithms, and algorithms are often created to process specific data structures. Together, they form the foundation of computer programming and are essential for the design of efficient software.

Lets try an example with arrays that we should be familiar with. An array is a data structure, therefore we are looking to be able to perform the following operation to it: traversal, insertion, deletion, merging.

We are going to try a basic array traversal. Our step would be given an array we will check each element one by one until the very last element. Our recipe in this case will be to use a loop and traverse our array.

```
public class ArrayTraversal {

    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};

        // Iterate through the array using a for loop.
        for (int i = 0; i < array.length; i++) {
            System.out.println(array[i]);
        }
    }
}
```

In this course, "**Data Structures and Algorithms**", we will explore the symbiotic relationship between these two essential aspects of computer science. We'll dive deeper into various types of data structures, understand their internal workings, learn when to use which data structure, and examine their space and time complexities. For the rest of this lesson we will tackle different operations with the array data structure.

# Insertion

## Insertion in Arrays

When it comes to arrays, **insertion** refers to the process of adding a new element to the array at a specific position. However, since arrays in Java are of fixed size, we don't have a direct method to add an element to it.

To perform an insertion operation, we usually have to define a new array with a size larger than the original array by one and then copy elements from the old array to the new one. After copying, we can add the new element at the desired location.

Here's an example of how you could do it:

```

public class ArrayInsertion {

    public static void main(String[] args) {

        int[] array = {1, 2, 3, 4, 5};
        int insertValue = 10;
        int insertIndex = 2;

        // print original array
        System.out.println("Original Array:");
        for (int i : array) {
            System.out.print(i + " ");
        }
        System.out.println();

        // create a new array of size array.length + 1
        int[] newArray = new int[array.length + 1];

        // copy elements from old array to new array, leaving
        // space for new element
        for (int i = 0; i < insertIndex; i++)
            newArray[i] = array[i];

        // add new element
        newArray[insertIndex] = insertValue;

        // copy the rest of elements from old array to new array
        for (int i = insertIndex + 1; i < newArray.length; i++)
            newArray[i] = array[i - 1];

        // print new array
        System.out.println("Array after insertion:");
        for (int i : newArray) {
            System.out.print(i + " ");
        }
    }
}

```

However, this method is not efficient, especially for large arrays or for multiple insertions. We would typically use a dynamic data structure, such as an `ArrayList` in Java, for these types of operations, which we will explore later in the course.

# **Deletion**

## **Deletion in Arrays**

Deletion in an array refers to the operation of removing an element from the array. Like insertion, deletion is also tricky with arrays because arrays in Java are of a fixed size.

To delete an element from an array, we don't have a direct method. Instead, we'll typically create a new array with a size smaller than the original array by one, and then copy all elements excluding the one to be deleted from the old array to the new one.

Let's look at an example where we delete an element from a specific index in an array:

```

public class ArrayDeletion {

    public static void main(String[] args) {

        int[] array = {1, 2, 3, 4, 5};
        int deleteIndex = 2;

        // print original array
        System.out.println("Original Array:");
        for (int i : array) {
            System.out.print(i + " ");
        }
        System.out.println();

        // create a new array of size array.length - 1
        int[] newArray = new int[array.length - 1];

        // copy elements from old array to new array excluding
        the element to be deleted
        for (int i = 0, k = 0; i < array.length; i++) {

            // this condition will skip the element to be
            deleted
            if (i == deleteIndex) {
                continue;
            }

            // if the index is not the deletion index, add the
            element to the new array
            newArray[k++] = array[i];
        }

        // print new array
        System.out.println("Array after deletion:");
        for (int i : newArray) {
            System.out.print(i + " ");
        }
    }
}

```

As with insertion, this method is not very efficient, especially for large arrays or multiple deletions. It involves shifting the elements and reallocating the memory which takes a lot of time and resources. This issue is one of the reasons why we often use more complex data structures like linked lists or dynamic arrays (ArrayList in Java) for tasks involving frequent insertions and deletions. We will learn about these data structures later in this course.

# Merging

## Merging Two Arrays

Merging in the context of arrays involves combining two arrays into a single array. Unlike insertion and deletion, merging does not alter the original arrays but creates a new one that holds all elements from both arrays.

The simplest way to merge two arrays in Java is to first create a new array of the correct size (which should be the sum of the lengths of the two input arrays), then copy over the elements from both arrays.

Here is an example:

```
public class ArrayMerge {

    public static void main(String[] args) {

        int[] array1 = {1, 2, 3, 4, 5};
        int[] array2 = {6, 7, 8, 9, 10};

        // create a new array of size array1.length +
        // array2.length
        int[] mergedArray = new int[array1.length +
array2.length];

        // copy elements from array1 to the new array
        for (int i = 0; i < array1.length; i++)
            mergedArray[i] = array1[i];

        // copy elements from array2 to the new array
        for (int i = 0; i < array2.length; i++)
            mergedArray[i + array1.length] = array2[i];

        // print merged array
        System.out.println("Merged Array:");
        for (int i : mergedArray) {
            System.out.print(i + " ");
        }
    }
}
```

In this code, we first create a new array of size equal to the sum of sizes of the input arrays. Then we copy all elements from the first array followed by all elements from the second array into the new array. As a result, the new array contains all elements from both input arrays in their original order.

However, this approach does not take into account any sorting or ordering of elements in the input arrays. If your input arrays are sorted and you want the merged array to be sorted as well, you'll need to implement a more complex merging algorithm, such as the one used in merge sort, or sort the merged array afterwards.

We'll explore such algorithms in later parts of this course, where we'll learn about sorting algorithms and their applications.

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

- Compare Linear and Binary search
- Analyze coding problems in five steps
- Learn to write more efficient code
- Use Big O analysis to determine algorithm runtime

# Algorithm Analysis

## Introduction to Algorithm Analysis

After learning about data structures and their operations, we'll now shift our focus towards **Algorithm Analysis**. The goal of algorithm analysis is to predict the resources that the algorithm requires. It can be both time and space.

Often in computer science, there are many ways to solve a problem. All these solutions, or algorithms, can vary in their efficiency. Some algorithms might run faster, while others might use less memory. To make an informed choice about which algorithm to use, we need to understand how to analyze and compare algorithms based on their efficiency. This is the essence of algorithm analysis.

In the process of algorithm analysis, we are mostly interested in the efficiency with respect to time and space:

1. **Time Complexity:** This refers to the total count of operations an algorithm will perform in its lifetime. As the size of the input data increases, the running time of the algorithm also grows. We express time complexity as a function of the input size ( $n$ ), often using **Big O notation**.
2. **Space Complexity:** This is the amount of memory space the algorithm needs to run to completion. Just like time complexity, space complexity is also expressed as a function of the input size ( $n$ ), often using Big O notation.

In this part of the course, we will learn about the concept of **Big O notation**, which is a theoretical measure of the execution of an algorithm or the amount of space it requires. It provides us with an upper bound of the complexity in the worst-case scenario, helping us to understand the worst-case scenario for an algorithm.

We will go through different complexities like  $O(1)$ ,  $O(n)$ ,  $O(\log n)$ ,  $O(n^2)$ , etc., and understand their impact on the running time and space of an algorithm. We will also learn how to calculate the time and space complexity of an algorithm and compare different algorithms.

# Big O

**Big O** is probably going to be one of the most important topics in your computer science career. Previously, we talked about how we wanted to optimize our code. By *optimize*, we mean how we make our code more efficient (or run faster). Turns out, there is a metric we use to describe the efficiency of algorithms and that is **Big O**. Understanding this will help better determine when your algorithm is more efficient, running faster, taking less space, etc.

definition

**Big O** notation is a way of describing how the performance of an algorithm or program changes as the size of the input grows. It is a mathematical notation used to represent the time complexity or space complexity of an algorithm.

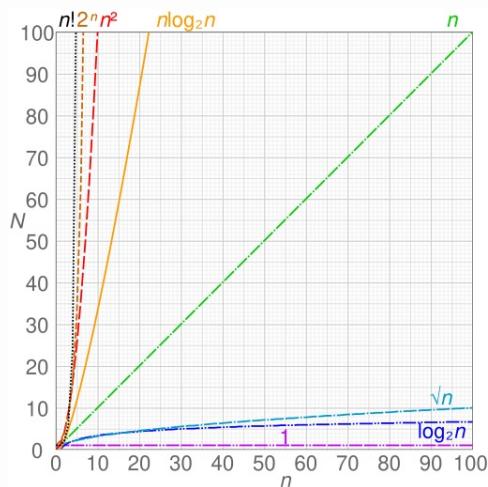
## Time Complexity

**Time complexity** is commonly estimated by counting the number of elementary operations performed by the algorithm. It's not about the time in seconds or milliseconds it takes for your program to run. Its primary focus is the number of operations relative to the size of the input. Suppose each elementary operation takes the same amount of time to perform. If each operation takes the same amount of time to perform, then we can think of it as the **more operations** performed by an algorithm, the **longer** that algorithm took to get to completion. We can think of the number of operations and the completion time as being related by a constant factor. In short, if one increases the other one does as well.

An algorithm's running time may vary with different inputs of the same size. The most common way to tackle this dilemma is to instead focus on the **worst case** scenario. Given an example, what's the longest it can take to perform all the operations it needs to in relation to its size. Sometimes, you will encounter cases where you are looking for the average case complexity, but that will be specified.

Time complexity can be measured by the number of elementary operation performed. If told to look for time complexity, we are looking for something in relation to the worst case scenario unless told otherwise. When we evaluate complexity of algorithms, we count the number of computation steps, or of memory locations.

The time complexity is generally expressed as a function of the size of the input. More specifically, it is based on the behavior of the complexity when the input size increases. The time complexity is commonly expressed using *Big O* notation, typically  $O(n)$ ,  $O(n \log n)$ ,  $O(2^n)$ , etc., where  $n$  is the size of the input. The image below is a representation of some of the common  $O$  notations.



### Common runtimes

- $O(\log n)$ ,  $O(n \log n)$ ,  $O(n)$ ,  $O(n^2)$ ,  $O(2^n)$
- $O(1)$

Let's start focusing on the purple line at the bottom.  $O(1)$  is **constant** time which occurs when the algorithm does not depend on the size of the input. Regardless of how much we change our array, the function will take the same amount of operations to get the answer. For example, imagine we are told to write a function that returns the first element of the array. Regardless of how large the array is, it will only take *one* step to do.

$O(n)$  or **linear** time is when the algorithm is proportional to the size of the input. In the image above, it is represented by a green line. We can see that the size of the input ( $n$  on the x-axis) is directly correlated with the number of operations ( $N$  on the y-axis). In the graph, everything below the green line is considered to be more optimal. Another way to think of it is as the

size of the array increases there is less change in terms of how many operation will be taken. Ideally, we want something below the linear time (green line) and avoid any times above it since they would take too long (making them much less efficient).

---

# Searching

**Searching** is a fundamental operation in computer science. Given an array of elements, a common task might be to find if a particular value exists within this array. We have different strategies(algorithms) to perform this operation each strategy with different time complexities, and the two most common are **Linear Search** and **Binary Search**. Each of these techniques uses a different approach and has different performance characteristics.

## Linear Search

The simplest approach to searching an array is with a linear search. As the name suggests, this algorithm works by checking each element in the array one-by-one, starting at the first element and continuing until it finds a match or until it has checked all elements. This method is straightforward and doesn't require the array to be sorted.

Here's a simple Java implementation of a linear search. The `target` variable represents the search target, and the `main` method prints the index of the target or that it was not found. Copy and paste the following in the top left panel:

```

public class LinearSearch {

    public static int linearSearch(int[] array, int target) {
        for (int i = 0; i < array.length; i++) {
            if (array[i] == target) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};
        int target = 3;

        int result = linearSearch(array, target);

        if (result == -1)
            System.out.println("Element not found in the
array");
        else
            System.out.println("Element found at index: " +
result);
    }
}

```

In this code, the `linearSearch` function iterates through each element in the array. If it finds the target element, it returns the index of the element. If it does not find the target, it returns -1.

## Binary Search

Binary search is a more efficient algorithm than linear search, but it requires the array to be sorted first. Binary search works by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, the algorithm continues the search on the lower half. Otherwise, it continues on the upper half.

Here's a simple Java implementation of a binary search. Copy and paste the following in the bottom left panel:

```

public class BinarySearch {

    public static int binarySearch(int[] array, int target) {
        int left = 0;
        int right = array.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            // Check if target is present at the mid
            if (array[mid] == target)
                return mid;

            // If target greater, ignore left half
            if (array[mid] < target)
                left = mid + 1;

            // If target is smaller, ignore right half
            else
                right = mid - 1;
        }

        // if we reach here, then element was not present
        return -1;
    }

    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};
        int target = 3;

        int result = binarySearch(array, target);

        if (result == -1)
            System.out.println("Element not found in the array");
        else
            System.out.println("Element found at index: " + result);
    }
}

```

In this code, the `binarySearch` function repeatedly checks the middle element of the array (or sub-array) until it finds the target or the search interval is empty. The beauty of binary search lies in its ability to cut the search space in half with each iteration.

However, keep in mind that binary search only works if the array is sorted. If it's not, you'll need to sort it first or use a different search algorithm.

Through these two search methods, we can begin to see how different approaches to the same problem can greatly affect the efficiency of our programs. We'll further explore these topics when we dive deeper into time complexity and Big O notation.

# Linear vs. Binary Search

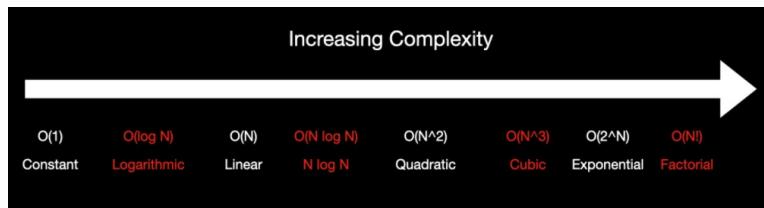
## Comparing Linear and Binary Searches

### Linear Search

In the worst case scenario, we have to check every single element in our array (see the animation on the top-left). The **Big O** for Linear Search is  $O(n)$ , linear time. The algorithm complexity increases as the size of our array increases. The algorithm takes an additional step for each additional element added to our array.

### Binary Search

In the worst case scenario, we are cutting our list in half with each iteration (see the animation on the bottom-left). The “halving” with each iteration can be expressed as the mathematical operation **logarithm**. The **Big O** for Binary Search is  $\log n + 1$ . We can ignore the  $+1$  because it is negligible in the grand scheme of time complexity. Binary Search is so efficient because doubling the length of our array means the algorithm would only do one additional step.



The image shows common Big O notations ordered by complexity. The notations are:  $O(1)$  - constant,  $O(\log N)$  - logarithmic,  $O(N)$  - linear,  $O(N \log N)$  -  $N \log N$ ,  $O(N^2)$  - quadratic,  $O(N^3)$  - cubic,  $O(2^N)$  - exponential, and  $O(N!)$  - factorial.

When dealing with *Big O*, we drop the non-dominant terms (yes, including constants). *Big O* notation only describes the growth rate of algorithms in terms of mathematical function, rather than the actual running time of algorithms.

- \*  $O(n^2 + n)$  is the same as  $O(n^2)$
- \*  $O(n + \log n)$  is the same as  $O(n)$
- \*  $O(\log n + 323123)$  is the same  $O(\log n)$

Using the image above, in terms of *Big O*, it is more efficient for us to use Binary Search instead of Linear Search because it is more efficient as our input size becomes larger. Remember that  $\log n$  is smaller than  $n$ . While

linear search is straightforward and works well on small, unsorted arrays, binary search is far more efficient for larger, sorted arrays. However, binary search requires the array to be sorted, which might be a limitation if you frequently add new elements to the array. Linear search does not have this limitation.

# Problem Solving

When tackling a problem in computer science and for the rest of this course, always do these **five steps**:

1. **Read/Listen**
2. **Think of Examples**
3. **BRUTE FORCE**
4. **Optimize**
5. **Test**

We are going to employ these five steps with the following prompt below.

## Prompt:

Create a function given a sorted array of  $n$  strings, that will return the location of a given element.

## Read/Listen

This goes with paying close attention to the problem we are trying to solve.

- \* What is our goal? What are we looking for? What do we want to make happen?
- \* What do we start with? What information is given? What can we extract from what we are given?

**Example:** We are given an array of strings, and we want to return the index at which the element of the search target.

### ## Think of Examples

This goes with picturing how our code works. It helps us visualize what we are about to do. Think of special cases and normal examples.

- \* Normal case: search for "cat" given the following ["bat", "cat", "dog", "eagle", "fish"]
- \* Special cases: search for "cat" given [] or ["ape", "donkey", "monkey", "raccoon", "snake"]

### ## BRUTE FORCE

This goes with thinking of the easiest or most straightforward way to obtain the solution. Let's start by using a linear search, since this is the easiest and most straightforward way of search an array.

```
public class ProblemSolving {

    public static int search(String[] array, String target) {
        for (int i = 0; i < array.length; i++) {
            if (array[i].equals(target)) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        String[] array = {"ape", "bat", "cat", "dog", "eagle"};
        String target = "dog";

        int result = search(array, target);

        if (result == -1)
            System.out.println("Element not found in the
array");
        else
            System.out.println("Element found at index: " +
result);
    }
}
```

# Optimize

The fourth thing is to **optimize**. In other words, pay attention to how to make your code better. What does a better code look like? For that, let's compare linear search and binary search.

## Prompt:

Create a function given a sorted array of  $n$  strings, that will return the location of a given element.

## Linear vs. Binary Search

The two approaches for searching for an element that we have covered are linear (or sequential) search and binary (or half-interval) search. Let's go over the differences.

### How they start:

- \* Linear : start from the beginning of our array
- \* Binary : start from the middle of our array

### Best case:

- \* Linear : the first element is what we are looking for
- \* Binary : the middle element is what we are looking for

### Worst case:

- \* Linear : last element of the array
- \* Binary : the value is the first or last element of the array

## What Do We Mean by “Best” and “Worst” Case?

We are going to think of **best case** as the situation where our program will take the least number of steps to return the result. On the other hand, the **worst case** situation will take the most number of steps to return the result.

Looking above, the worst case for binary and the best case for linear are the same. Does that mean that linear is a better search function? No, it does not. As a matter of fact, more often than not, binary search will be faster than linear search for a given large array. We will not know in advance

where the search target is located in the array. We need to choose the code that on average will be faster, regardless of what the array actually looks like.

Given the discussion above, we know on average that a binary search is more efficient than a linear search. As such, let's optimize and update our search method to use a binary search algorithm.

```
public static int search(String[] array, String target) {  
    int left = 0;  
    int right = array.length - 1;  
  
    while (left <= right) {  
        int mid = left + (right - left) / 2;  
        int result = target.compareTo(array[mid]);  
  
        if (result == 0)  
            return mid;  
        if (result > 0)  
            left = mid + 1;  
        else  
            right = mid - 1;  
    }  
    return -1;  
}
```

# Test

The fifth and final step is to test your work. This does not mean that you write up a formal unit test (though thorough testing is never a bad idea), but you should at least give your code a few sample inputs to which you already know the answer. The previous search target was "dog", which we know is an element of the test array. Now, let's try a test case where we know the algorithm should not find the search target.

```
String target = "fish";
```

Run the code again. You should see a message stating the element is not found in the array.

Back in the **Think of Examples** step, we talked about special cases like an empty array. Let's update our code to see how it handles an empty array.

```
String[] array = {};
String target = "fish";
```

Just as before, we should see the message about the element not being in the list.

Finally, let's consider a few more edge cases. What happens if we search for "Dog" instead of using a lowercase letter? Update the values for `array` and `target` so they match the example below.

```
String[] array = {"ape", "bat", "cat", "dog", "eagle"};
String target = "Dog";
```

Because Java is case sensitive, it should print the message that "Dog" is not in the array.

Ignoring capitalization is a common search practice, and we want to do the same with our search algorithm. Find the line of code that calculates the `result` variable. We want the string comparison to force the search target and element to be uppercase.

```
int result =
    target.compareToIgnoreCase(array[mid].toUpperCase());
```

Java provides us with the `compareToIgnoreCase()` method that compares the values of two strings while ignoring case. As long as the spelling matches, the algorithm should return the index of search target.

challenge

### Try these variations:

Update `target` to have the different spellings below. The search algorithm should still return the index 3 despite the odd capitalizations.

\* "DOG", "dOG", and "d0g"

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- **Define Big O**
- **Differentiate between Big O, Big Theta, and Big Omega**

# Big O Notation: The Language of Algorithm Efficiency

## Defining Performance

**Big O notation** is the language we use in computer science to describe the performance of an algorithm. Specifically, it provides an upper bound on the time complexity or space complexity of an algorithm in terms of the size of the input data, giving us a measure of the worst-case scenario. Big O notation abstracts the details of the machine where the algorithm runs, allowing us to focus on the algorithm's inherent efficiency.

It's important to understand that Big O notation does not give us an exact count of operations, memory used, or time taken by an algorithm. Instead, **it provides a measure of the rate of growth of these resources as the input size increases.**

It helps us answer questions like "If we double the size of the input, will the algorithm take twice as long, or will it take four times as long, or perhaps not increase significantly at all?" This understanding is vital when we compare different algorithms and choose the most efficient one for our specific problem.

In Big O notation, we only keep the term that grows the fastest as the input size increases and discard the rest. For example, if we have a time complexity of  $2n^2 + 3n + 4$ , in Big O notation, we would represent this as  $O(n^2)$ , as  $n^2$  grows the fastest for large inputs.

Throughout this course, we'll frequently use Big O notation to analyze and compare the efficiency of various data structures and algorithms. It is a fundamental concept that every computer scientist and software engineer should master.

# Big O, Big Theta, Big Omega

Aside from **Big O** there are some other notations that we use to describe the performance of an algorithm. Big O is by far the most popular and the one we will be interacting with in the course. However, we want you to be informed of the other notations *Big $\Theta$ (Theta)*, and *Big $\Omega$  (Omega)*. While they might appear similar, they are used to denote different aspects of algorithmic performance.

## Big O Notation

**Big O** notation is the most commonly used notation to represent the time complexity or space complexity of an algorithm. It gives an upper bound on the time or space needed by an algorithm in the worst-case scenario, hence it's often viewed as the worst-case scenario analysis.

If the running time of an algorithm is  $O(f(n))$ , it means that the running time grows at most linearly with  $f(n)$ . For example, if we say an algorithm has a time complexity of  $O(n)$ , it means that in the worst case, the algorithm's running time increases linearly with the size of the input.

## Big $\Theta$ (Theta) Notation

**Big Theta** notation is used when we want to give both an upper and lower bound on the time complexity of an algorithm. In other words, if a function's growth rate is  $\Theta(g(n))$ , there exist constants  $n_0, c_1$  and  $c_2$  such that at and beyond  $n_0$ , the value of the function is always between  $c_1g(n)$  and  $c_2g(n)$  for all  $n > n_0$ . Hence, if we say that the running time of an algorithm is  $\Theta(n^2)$ , it means the algorithm's running time grows exactly like  $n^2$  in the average-case scenario.

## Big $\Omega$ (Omega) Notation

**Big Omega** notation is used to provide a lower bound on the time complexity or space complexity of an algorithm. It's often seen as the best-case scenario analysis. If the running time of an algorithm is  $\Omega(g(n))$ , it means that the running time grows at least as quickly as  $g(n)$ .

For instance, if we say an algorithm has a time complexity of  $\Omega(n)$ , it means that in the best case, the algorithm's running time increases linearly with the size of the input.



# Understanding Best, Worst, Average, and Expected Cases

While **Big O** gives us an upper bound (worst case), **Big Ω** gives us a lower bound (best case), and **Big Θ** provides us with both an upper and lower bound (tight case). All these notations are invaluable tools when we analyze and compare the efficiency of algorithms, as they allow us to estimate an algorithm's performance as the input size grows.

We keep mentioning best, worst cases, let's us use linear search to help us explain and better visualize it. Why linear search? The Linear Search algorithm is one of the simplest searching algorithms, and it provides an excellent opportunity to understand how the best, worst, average, and expected cases work in algorithm analysis

This linear search is modified from what we have seen in the past. The count variable keeps track of how many iterations were run before returning a result.

```

public class LinearSearch {

    public static int[] linearSearch(int[] array, int target) {
        int count = 0;
        for (int i = 0; i < array.length; i++) {
            count++;
            if (array[i] == target) {
                int[] result = {i, count};
                return result;
            }
        }
        int[] result = {-1, count};
        return result;
    }

    public static void main(String[] args) {
        int[] array = {6, 2, 0, 5, 1, 7, 8, -1, 4};
        int target = 8;

        int[] result = linearSearch(array, target);

        if (result[0] == -1) {
            System.out.println("Element not found in the
array");
            System.out.println(result[1] + " calculations were
needed");
        } else {
            System.out.println("Element found at index: " +
result[0]);
            System.out.println(result[1] + " calculations were
needed");
        }
    }
}

```

Let's dive into each case scenario, now that we are done reviews on linear search :

### **Best Case**

The best case scenario for linear search occurs when the target value is located at the very beginning of the array (i.e., the first element is the target). In this situation, only one comparison is necessary, so the best-case time complexity is  $O(1)$ . That would be the scenario where the 8 was in the first element.

Set the `target` variable to the value 6, which is the first element. The search algorithm should only take one iteration.

```
int target = 6;
```

## Worst Case

The worst-case scenario for linear search happens when the target value is at the very end of the array or is not in the array at all. In both cases, the algorithm needs to traverse the entire array, resulting in a time complexity of  $O(n)$ , where  $n$  is the number of elements in the array. That would be the scenario where the 8 is not in our array or that it was the very last element.

Set the target variable to the value 4, which is the last element. The search algorithm should take nine iterations.

```
int target = 4;
```

## Average Case

The average case assumes that the target value has an equal chance of being in any position in the array. On average, the algorithm needs to check half of the elements in the array before finding the target. Thus, the average-case time complexity is also  $O(n)$ , but on average, it performs roughly  $O(\frac{n}{2})$  operations, which is more efficient than the worst case but still scales linearly with  $n$ .

Before we can show the average case, we need to make some modifications to the `main` function. We are going to calculate the total number of iterations to find each of the elements in the array and then divide by 9, the number of elements in the array. This gives us the average number of calculations to find any element in the list.

```

public static void main(String[] args) {
    int iterations = 0;
    int[] array = {6, 2, 0, 5, 1, 7, 8, -1, 4};

    for (int i = 0; i < 9; i++) {
        int target = array[i];

        int[] result = linearSearch(array, target);

        if (result[0] == -1) {
            System.out.println("Element not found in the
array");
            System.out.println(result[1] + " calculations
were needed");
        } else {
            System.out.println("Element found at index: " +
result[0]);
            System.out.println(result[1] + " calculations
were needed");
        }
        iterations += result[1];
    }
    System.out.println("The average search took " +
iterations/9 + " iterations");
}

```

## Expected Case

The expected case often aligns with the average case, particularly when there's an equal probability of the target value being at each position in the array. In more complex scenarios, the expected case might vary based on the known or assumed probability distribution of the input. In the case of a linear search, if we don't have any additional information about the data, the expected case would also be  $O(n)$ .

info

### **Please Note:**

Linear search serves as an excellent introduction to algorithm analysis. Despite its simplicity, understanding the time complexity in different scenarios helps us to appreciate the trade-offs involved in choosing algorithms. While linear search is useful due to its simplicity and versatility (as it works on unsorted data), it might not be the optimal choice for large datasets or when performance is a key factor, as other algorithms like binary search or a hash-based search can offer better time complexity.

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- **Define space complexity**
- **Compare and contrast Arrays and ArrayLists**
- **Analyze insertion and merge sorting algorithms**

# Space Complexity Analysis

## Introduction to Space Complexity Analysis

After delving into time complexity, it's essential to understand another crucial aspect of algorithm analysis, **Space Complexity**. Space complexity provides us with a measure of the amount of memory an algorithm needs to run to completion. It's as crucial as time complexity, especially in systems with limited memory resources or where memory cost is a significant factor.

### info

The **space complexity** of an algorithm quantifies the amount of space or memory taken by an algorithm to execute as a function of the length of the input. It includes the memory required for the input data itself and any additional space needed for the algorithm to transform the input and produce the output.

Space complexity is sometimes overlooked in favor of time complexity, but it's particularly important in today's computing world. As datasets become larger, an algorithm that was previously feasible could become impractical due to high memory requirements. So understanding space complexity is essential to ensure our solutions scale well.

In this assignment, we will explore different algorithms, their space complexity, and how to analyze them. Furthermore, we'll also discuss the trade-offs between time and space complexity, as optimizing for one often comes at the expense of the other. **Please note:** more often than not, when you are asked about the complexity of an algorithm (unless specifically ask for space) they are referring to time complexity or both.

# Fundamental Concepts

Before we get into different data structures and the space complexity of some sorting algorithms, we need to cover some fundamental concepts first. This will help us understand and quantify the space complexity of algorithms.

## Fixed Space and Variable Space

Space complexity deals with memory usage of an algorithm. We can categorize this into two ideas:

1. **Fixed Space:** This is the space required to store certain data and variables, which does not change during the execution of the algorithm. For instance, simple variables and constants used in the algorithm fall into this category.
2. **Variable Space:** This is the space needed by variables that dynamically change during the execution of the algorithm. It includes things like dynamic data structures, recursion stack space, etc.

The total space required by an algorithm is the sum of the fixed space and the variable space.

Consider the following Java code snippet:

```
public class SpaceComplexity {

    public static void main(String[] args) {
        int n = 100;      // fixed space
        int[] array = new int[n];    // variable space

        for (int i = 0; i < n; i++) {
            array[i] = i;
        }
    }
}
```

In this program, the integer `n` takes up a fixed space, while the `array` is variable space, which depends on the value of `n`.

When calculating space complexity, we usually focus on the variable space as it is the one that grows with the input size.

## **Big O Notation for Space Complexity**

Just like time complexity, we use Big O notation to represent space complexity. This allows us to express the space complexity of an algorithm in relation to the size of the input.

In the example above, the space complexity of the program is  $O(n)$ . That's because the amount of space used by the array scales linearly with the input  $n$ .

In the following sections, we will look into more complex examples and also compare the space complexities of two commonly used data structures in Java: Arrays and ArrayLists. This will provide us with a practical understanding of how data structure choices can impact the memory utilization of our programs.

# Auxiliary Space

Space complexity is often confused for auxiliary space. Auxiliary space is any additional storage space used to assist in the execution of a program.

Space Complexity = Auxiliary space + Space use by input values

Space complexity is dependent upon auxiliary space. As auxiliary space increases, so does space complexity.

## Diving Deeper

To fully understand the significance of auxiliary space, let's take a look at an example:

```
public class AuxiliarySpace {

    public static int findSum(int[] array) {
        int sum = 0;      // auxiliary space

        for (int i = 0; i < array.length; i++) {
            sum += array[i];
        }

        return sum;
    }

    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};      // space used by input

        int sum = findSum(array);

        System.out.println("Sum = " + sum);
    }
}
```

In the `findSum` method above, the space used by the `array` is not considered auxiliary space, as it's part of the input to the method. However, the `sum` variable, which is used to keep track of the total sum of the array, is considered auxiliary space because it's extra space used by the algorithm, independent of the input size.

The concept of auxiliary space becomes especially important when we discuss algorithms that might take in large amounts of data as input, but only use a small, fixed amount of extra space to perform their operations.

In this example, no matter how large our input array is, our `findSum` function only ever needs a single integer's worth of extra space to perform its calculation. Hence, the auxiliary space complexity is  $O(1)$ , which means it uses constant extra space.

Being able to minimize auxiliary space can be a significant advantage when dealing with large data, as it allows for more memory-efficient algorithms. Understanding the difference between space complexity and auxiliary space complexity helps you write algorithms that not only perform well but are also resource-efficient.

In the next section, we'll take this concept further, comparing the space complexities and auxiliary space used by two commonly used data structures in Java: Arrays and ArrayLists.

# Array vs ArrayList

## Comparing Space Complexity

One of the essential factors to consider while choosing the right data structure is its space complexity. Understanding how much space a data structure consumes can significantly impact the overall efficiency of an algorithm. Let's compare Arrays and ArrayLists in Java to understand this better.

### Arrays in Java

An array in Java is a static data structure that holds a fixed number of values of a single type. The length of the array is established when the array is created, and it cannot be changed.

Here's an example of an array. The variable `array` is instantiated with enough memory to hold five integers. The program then iterates over the array and assigns a random integer (0-99) to each element. Finally, the program uses the `.add()` method to add a sixth element to the array.

```
import java.util.Random;

public class ArrayVsArrayList {

    public static void main(String args[]) {
        Random rand = new Random();
        int[] array = new int[5];

        for (int i = 0; i < 5; i++) {
            array[i] = rand.nextInt(100);
        }

        array.add(0);
    }
}
```

As expected, this code produces an error. Arrays have a fixed length, and trying to invoke a method to change its size causes a problem for the compiler. Once you instance an array with a given size, it cannot change. Because of its fixed length, we can say the space complexity for an array is  $O(n)$ , where  $n$  is the number of elements in the array.

## ArrayLists in Java

On the other hand, an ArrayList in Java is a dynamic data structure, which means it can grow or shrink as needed. When elements are added to an ArrayList, its capacity grows automatically.

Here's an example of how an ArrayList. The same actions are performed on it as the array example from above.

```
import java.util.ArrayList;
import java.util.Random;

public class ArrayVsArrayList {

    public static void main(String args[]) {
        Random rand = new Random();
        ArrayList<Integer> numbers = new ArrayList<Integer>(5);

        for (int i = 0; i < 5; i++) {
            numbers.add(rand.nextInt(100));
        }

        numbers.add(0);
    }
}
```

When an element is added to an ArrayList, if it is full, a new, larger array is created, and the old array is copied into the new one. Therefore, in addition to the space required for the elements themselves, extra space is required during the resizing operation. The space complexity of an ArrayList is generally  $O(n)$ , but the auxiliary space complexity during the resize operation is  $O(n)$  as well, making ArrayLists less memory efficient than arrays during insertions when resizing is required.

While both Arrays and ArrayLists have a space complexity of  $O(n)$ , ArrayLists require more auxiliary space because of the resizing operations. Therefore, if the size of the data structure is known in advance and will not change, an array would be a more memory-efficient choice. However, if you need a flexible size data structure, ArrayList provides this functionality at the cost of additional memory. The choice between using an Array or an ArrayList often comes down to a trade-off between memory efficiency and flexibility. Understanding these trade-offs is crucial for effective programming.

# Insertion sort

In order to better help visualize the space complexities. We are going to compare the space complexities of two sorting algorithms: **insertion sort** and **merge sort**.

When interacting with **insertion sort**, we use only a constant amount of additional memory apart from the input array, the space complexity is  $O(1)$ . A constant Big O value in terms of space complexity means that our algorithm will take the same amount of space regardless of the input size. In **Merge sort**, we require an auxiliary array to store the merged subarray. The size of this auxiliary array is at most  $n$ , and as a result, the space complexity of Merge sort is  $O(n)$ .

## Insertion Sort

An insertion sort is a simple sorting algorithm that works by repeatedly inserting an unsorted element into a sorted portion of an array until the entire array is sorted. A visual representation is provided below:



## Space Complexity in Context

Insertion sort is an efficient sorting algorithm for small data sets. It sorts the input list by treating the leftmost part of the array as a sorted segment. It then iterates through the rest of the list and inserts the element in its correct position in the sorted segment.

Let's take a look at a simple implementation of Insertion Sort in Java:

```

public class InsertionSort {

    // Method to perform insertion sort on the given array
    public static void sort(int[] arr) {
        int n = arr.length;

        // Traverse through the array from the second element
        for (int i = 1; i < n; i++) {
            int key = arr[i]; // Store the current element to be inserted into the sorted part
            int j = i - 1; // Initialize the pointer to the last element of the sorted part

            // Shift elements greater than the key to the right
            // to create space for inserting the key in the correct position
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }

            // Insert the key into the correct position in the sorted part
            arr[j + 1] = key;
        }
    }
}

```

## Space Complexity of Insertion Sort

In terms of space complexity, insertion sort is quite efficient. The space complexity of an algorithm is determined by the amount of memory it requires in relation to the size of the input. In the case of insertion sort:

- **Fixed Space:** This includes variables and constants. In the implementation above, the fixed space is occupied by variables such as `n`, `i`, `key`, and `j`.
- **Variable Space:** This includes dynamically allocated space. In this case, there's no dynamically allocated space, as we're not using any additional data structures, like arrays or linked lists.

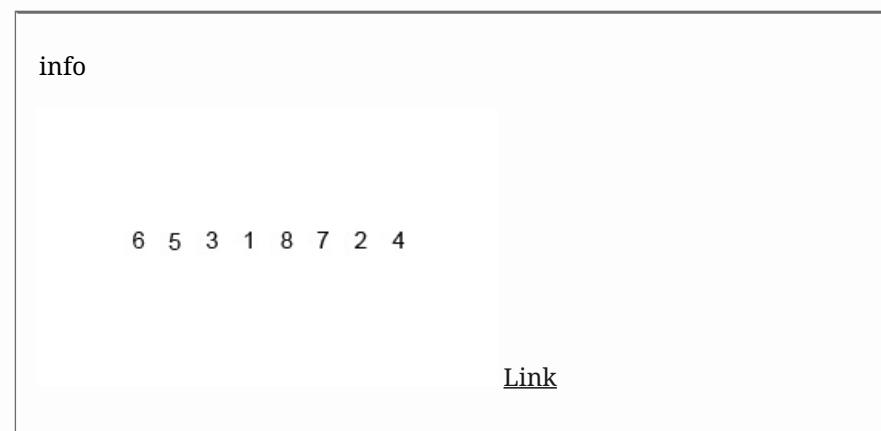
Therefore, the total space complexity of insertion sort is the sum of the fixed and variable space. Given that the variable space is zero, and the fixed space does not depend on the size of the input array, we can conclude that the Insertion Sort algorithm has a **constant space complexity**, denoted as  $O(1)$ . This means the space requirement does not change with the size of the input array.

Understanding the space complexity of sorting algorithms like insertion sort is crucial. Even though the insertion sort might not be the most efficient in terms of time complexity, especially for larger data sets, it's very space-efficient because it sorts the list in place, without needing additional storage, making it an excellent choice when memory is a consideration.

# Merge Sort

**Merge sort** involves breaking down an unsorted array into  $n$  subarrays, each consisting of one element, which can be considered sorted in a straightforward manner. Then, we merge these sorted subarrays together repeatedly, forming new sorted subarrays, until we end up with only one sorted array of size  $n$ .

A visual representation is provided below:



## Space Complexity in Context

Merge sort is another common sorting algorithm that leverages the divide-and-conquer approach to sort an array or list of elements. We will delve more into the divide and conquer in a bit. In short, this means that an algorithm divides the current problem into subproblems. It then solves the subproblems in order to solve the whole problem. The algorithm works by repeatedly splitting the list into two halves until we have sublists with one element each. We then merge these sublists back together such that they are sorted.

Here is a simple implementation of merge sort:

```
public class MergeSort {

    // Merges two subarrays of arr[].
    // First subarray is arr[left..mid].
    // Second subarray is arr[mid+1..right].
    private static void merge(int arr[], int left, int mid, int
        right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;
```

```

// Create temporary arrays to store the two subarrays
int leftArray[] = new int[n1];
int rightArray[] = new int[n2];

// Copy data to temporary arrays
for (int i = 0; i < n1; ++i)
    leftArray[i] = arr[left + i];
for (int j = 0; j < n2; ++j)
    rightArray[j] = arr[mid + 1 + j];

// Merge the two temporary arrays back into arr[left..right]

int i = 0, j = 0; // Initialize pointers for the two
temporary arrays
int k = left; // Initialize pointer for the merged array

// Compare elements from leftArray and rightArray and
put the smaller one into arr
while (i < n1 && j < n2) {
    if (leftArray[i] <= rightArray[j]) {
        arr[k] = leftArray[i];
        i++;
    } else {
        arr[k] = rightArray[j];
        j++;
    }
    k++;
}

// Copy remaining elements from leftArray, if any
while (i < n1) {
    arr[k] = leftArray[i];
    i++;
    k++;
}

// Copy remaining elements from rightArray, if any
while (j < n2) {
    arr[k] = rightArray[j];
    j++;
    k++;
}

// Main method that sorts arr[left..right] using merge()
public static void sort(int arr[], int left, int right) {
    if (left < right) {

```

```

int mid = (left + right) / 2;

// Sort the first half and the second half
separately
sort(arr, left, mid);
sort(arr, mid + 1, right);

// Merge the sorted halves
merge(arr, left, mid, right);
}

}

```

## Space Complexity of Merge Sort

The space complexity of an algorithm involves the amount of memory it needs in proportion to the input size. In the case of Merge Sort:

- **Fixed Space:** This includes variables and constants. In the above implementation, the fixed space is occupied by variables such as `left`, `right`, `mid`, `n1`, `n2`, `i`, `j`, and `k`.
- **Variable Space:** This includes dynamically allocated space. Here, the variable space is used by the temporary arrays `leftArray` and `rightArray`.

As such, the total space complexity of Merge Sort is the sum of the fixed and variable space. Since the variable space in this algorithm scales with the size of the input array (specifically, we need to create temporary arrays for each element in the array), merge sort has a **linear space complexity**, denoted as  $O(n)$ .

This means that the space required by the merge sort algorithm increases linearly with the size of the input. Although merge sort is efficient in terms of time complexity ( $O(n \log n)$ ), its space complexity is higher than that of some other sorting algorithms, such as insertion sort, which has a space complexity of  $O(1)$ . Understanding this trade-off is critical when dealing with large datasets and limited memory resources.

# **Formative Assessment 1**

## **Formative Assessment 2**

## **Learning Objectives: Recursion**

**Learners will be able to...**

- **Define recursion**
- **Identify the base case**
- **Identify the recursive pattern**
- **Identify when to use recursion**

# Recursion in Data Structures

---

## What is Recursion?

---

**Recursion** is a programming concept where a function calls itself inside its own definition. Recursion refers to the method where the solution to a problem depends on solutions to smaller instances of the same problem. This may seem like a strange idea, but it can be used to solve problems that would be difficult or impossible to solve with a traditional, iterative approach.

To understand recursion, it's helpful to think of a problem that can be broken down into smaller and smaller subproblems (these are sometimes referred to as self-similar problems). In data structures and algorithms, recursion plays an essential role in breaking down complex problems into more manageable parts.

## When to Use Recursion

---

Recursion is especially potent in scenarios where:

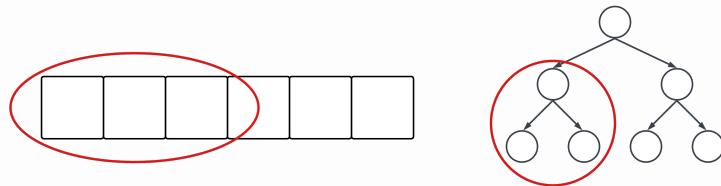
1. **Natural Hierarchy Exists:** For instance, in tasks like traversing a file directory.
2. **Problem Can Be Broken Down:** Such as sorting, where a large list can be split into smaller ones.
3. **Solution Requires Multiple Steps:** For example, calculating factorial values or Fibonacci numbers.

Applying recursion requires a mental shift because it can initially seem counterintuitive. Instead of seeing the solution in a linear, step-by-step manner, one needs to trust that smaller versions of the problem will be solved, culminating in the solution to the original issue. The smallest problem we know how to solve in recursion is the base case.

## Recursive Algorithms

---

Recursion is fundamental when working with data structures and algorithms. There are many data structures that are self-similar. These can be linear data or non-linear structures. If you look at the first half of an array, we can think of this half as being its own, smaller array. The same thing can be done to a tree. A subset of a tree can be thought of being its own, smaller tree.



The image depicts an array and a tree. A smaller, self-similar portion of each structure is outlined with a red circle or oval.

Because of the inherent, self-similar structure of some of the data structures, common algorithms often traverse these structures in a recursive manner. This could be for searching, for sorting, or for something entirely different. Regardless, recursion and data structures and algorithms go hand in hand.

---

The mathematical concept of factorial ( $n!$ ) is the product of all consecutive, positive integers less than or equal to  $n$ . In other words:

Let's look at the concrete example of  $5!$ . This can be expressed as

$5 * 4 * 3 * 2 * 1$ . If we look closely at a subset of this, we can see that

$5 * 4 * 3 * 2 * 1$  is actually  $5 * 4 * 3 * 2$ . So we can say that  $5 * 4 * 3 * 2 = 5 * 4 * 3 * 2 + 1$ . If we continue with this logic, then  $5 * 4 * 3 * 2 + 1 = 5 * 4 * 3 * 2 * 1$ . We can continue breaking down this larger problem down into smaller, self-similar subproblems. Mouse over the image below to see the recursive structure of  $5!$ .

$$5! = 5 * 4 * 3 * 2 * 1$$

Because  $5!$  is self-similar, recursion can be used to calculate the answer.

Copy the following code into the text editor on your left and click [TRY IT](#) to test the code. You should see [120](#) as the result.

```
public class Factorial {  
  
    public static int factorial(int n) {  
        if (n == 1) { //base case  
            return 1;  
        } else { //recursive step  
            return n * factorial(n - 1);  
        }  
    }  
  
    public static void main(String[] args) {  
        System.out.println(factorial(5));  
    }  
}
```

Recursion is an abstract and difficult topic, so it might be a bit hard to follow what is going on here. When `n` is 5, it starts a multiplication problem of `5 * factorial(4)`. The method runs again and the multiplication problem becomes `5 * 4 * Factorial(3)`. This continues until `n` is 1. It returns the value `1`, and Java solves the multiplication problem `5 * 4 * 3 * 2 * 1`. The video below should help explain how `5!` is calculated recursively.



---

Each recursive method has two parts: the recursive case (where the method calls itself with a different parameter) and the base case (where the method stops calling itself and sometimes returns a value).

The base case is the most important part of a recursive method. Without it, the method will never stop calling itself. Like an infinite loop, Java will stop the program with an error. Replace the `factorial` method in your code with the one below and see what happens.

```
// This recursive method returns an error. THE BASE CASE IS  
MISSING.  
public static int factorial(int n) {  
    return n * factorial(n - 1);  
}
```

When creating a recursive algorithm, always start with the base case. That is, determine when your algorithm should stop running. Each time the method is called recursively, the program should get one step closer to the base case.

### challenge

---

If you were to call the `factorial` method with `0` as the argument, you will see an error since the method would never stop calling itself.

```
System.out.println(factorial(0));
```

Modify the `factorial` method so that it will return `1` if it is passed any non-positive integer.

► **Solution**

# A Deep Dive with Merge Sort

---

## Recursive Merge Sort

---

Let's take another look at the merge sort, which we briefly saw in the previous module. A merge sort lends itself nicely to recursion because of the way it tackles the sorting problem. Instead of trying to sort an entire list (which is too big), a merge sort divides the array in half and tries to sort the smaller arrays.

The algorithm keeps dividing the arrays until they are small enough to be sorted. These sorted arrays are combined with other arrays and sorted once more. A merge sort keeps combining all of these sorted arrays until there is only one array left, which is completely sorted.

Dividing a big problem into smaller, easier to solve problems is a hallmark of recursion. Here is a recursive implementation of a merge sort:

```
public class MergeSort {

    private static void merge(int arr[], int left, int
        mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        int leftArray[] = new int[n1];
        int rightArray[] = new int[n2];

        for (int i = 0; i < n1; ++i)
            leftArray[i] = arr[left + i];
        for (int j = 0; j < n2; ++j)
            rightArray[j] = arr[mid + 1 + j];

        int i = 0, j = 0;
        int k = left;
        while (i < n1 && j < n2) {
            if (leftArray[i] <= rightArray[j]) {
                arr[k] = leftArray[i];
                i++;
            } else {
                arr[k] = rightArray[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            arr[k] = leftArray[i];
            i++;
            k++;
        }

        while (j < n2) {
            arr[k] = rightArray[j];
            j++;
            k++;
        }
    }

    public static void sort(int arr[], int left, int
        right) {
        if (left < right) {
            int mid = (left + right) / 2;
            sort(arr, left, mid);
            sort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }
}
```

In particular, let's focus on the `sort` method where recursion is actually used. Unlike the factorial example, we do not see two branches in our conditional. It is much harder to differentiate between the base case and the recursive case because we have only one branch in our conditional. The only branch is the recursive case. As long as `left` is less than `right`, the method will keep recursing. When `left` is no longer less than `right`, the method skips the recursive calls and simply ends. This is the base case.

```
void sort(int arr[], int left, int right) {  
    if (left < right) {  
        int mid = (left + right) / 2;  
        sort(arr, left, mid);           // Recursive call  
        on left half  
        sort(arr, mid + 1, right);     // Recursive call  
        on right half  
        merge(arr, left, mid, right);  
    }  
}
```

The algorithm continually splits the array in half; `left` to `mid` is one half, while `mid + 1` to `right` is the other half. This splitting continues until each sub-array has a single element. In an array with a length of one, `left` and `right` will have the same value, which means `left` is no longer less than `right`. These sublists are then merged together, in a recursive manner, to produce a sorted list.

# **Formative Assessment 1**

---

## **Formative Assessment 2**

---

# **Learning Objectives**

**Learners will be able to...**

- **Define divide and conquer**
- **Implement divide and conquer algorithms.**
- **Analyze the time and space complexities associated with these divide and conquer algorithms.**

# Divide and Conquer

## Merge Sort Revisited

In the last assignment, we talked about the merge sort and why it is suited to a recursive implementation. In short, the ability to create subproblems out of a larger problem means recursion is a good choice for the algorithm. Below is the sort method from a merge sort. We see how the left side of the original array is sorted into arrays of only one element. From there, the right side of the original array is divided into arrays of only one element.

```
void sort(int arr[], int left, int right) {  
    if (left < right) {  
        int mid = (left + right) / 2;  
        sort(arr, left, mid);           //divide array into the  
        left half  
        sort(arr, mid + 1, right);     //divide array into the  
        right half  
        merge(arr, left, mid, right); //combine the sorted  
        arrays  
    }  
}
```

Computer scientists refer to this problem solving technique as **divide and conquer**. It should be noted that the divide and conquer technique often includes a third step, one that combines all of the small subproblems into a single solution. We saw how a merge sort uses the `merge` method after all of the dividing has been done to generate the final, sorted array.

An algorithm for a merge sort is similar to an algorithm for factorial. Both of them use recursion to divide the bigger problem into smaller subproblems.  $5!$  is defined as  $5 \times 4!$ , and  $4!$  is defined as  $4 \times 3!$  and so on. Both of the recursive algorithms for a merge sort and factorial rely on divide and conquer.

```
public static int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        //calculate the factorial of a smaller number  
        return n * factorial(n - 1);  
    }  
}
```

Because of this, people often have the incorrect belief that the terms recursion and divide and conquer can be used interchangeably. While these two terms are similar, there is an important difference. Divide and conquer is the problem solving technique, while recursion is the way in which divide and conquer is expressed as code.

## When to Use Divide and Conquer

The Divide and Conquer algorithmic design paradigm is based on multi-branched recursion, where a problem is solved by:

1. **Dividing it into smaller sub-problems.**
2. **Conquering the sub-problems recursively.**
3. **Combining these solutions to craft the answer to the original problem.**

This strategy leans on the idea that a complex problem can often be solved more easily by splitting it into smaller, easier, and more manageable parts. The characteristics of divide and conquer are:

1. **Self-similarity:** Each sub-problem is essentially a smaller version of the general problem.
2. **Independence:** The sub-problems are solved independently.
3. **Recursive Nature:** The approach repeatedly breaks down a problem into sub-problems.

# Binary Search

## Iterative Binary Search

**Binary Search** is a quintessential example of the divide and conquer approach in action. Given a sorted list and a target value, binary search aims to find the target's position in the list. Instead of searching sequentially, binary search divides the list in half repeatedly until the target value is found or the range is empty.

Here's a simplified outline of how Binary Search works:

1. Compare the target value to the middle element.
2. If they are not equal, the half in which the target cannot lie is eliminated.
3. Continue the search on the remaining half.

This continuous halving is a perfect demonstration of the divide and conquer strategy. It should be noted, however, that you do not have to use recursion to implement a divide and conquer algorithm. The code sample below uses a while loop instead of recursion. **Important**, do not try to run the code below in the IDE. It will cause errors.

```
public static int binarySearch(int[] array, int target) {  
    int left = 0;  
    int right = array.length - 1;  
  
    while (left <= right) {  
        int mid = left + (right - left) / 2;  
  
        if (array[mid] == target)  
            return mid;  
  
        if (array[mid] < target)  
            left = mid + 1;  
        else  
            right = mid - 1;  
    }  
    return -1;  
}
```

## Recursive Binary Search

Let's implement a recursive solution for a binary search. The base case is when the array is empty (`right < left`) or when you have found the search target (`target == array[mid]`). In all other cases, we are going to recurse. However, we want to ignore the half of the list that we know cannot contain the search target. When you call `binarySearch` with `left` and `mid - 1`, you are ignoring the right half of the array. When you call the function with `mid + 1` and `right`, you are ignoring the left half.

```
public static int binarySearch(int[] array, int target, int left, int right) {  
    int mid = left + (right - left) / 2;  
  
    if (right < left)  
        return -1;  
  
    if (target == array[mid])  
        return mid;  
    else if (target < array[mid])  
        return binarySearch(array, target, left, mid - 1);  
    else  
        return binarySearch(array, target, mid + 1, right);  
}
```

This approach is significantly faster than a linear search, especially for larger datasets. For a list with  $n$  elements, Binary Search can find an element in about  $O(\log n)$  steps, whereas linear search would take, on average,  $O(n/2)$  steps.

# Binary Search Analysis

## Time Complexity

The primary reason binary search is efficient lies in its mechanism: With each comparison, it halves the search space. Let's break this down:

- **First iteration:** We have  $n$  elements.
- **Second iteration:** We have  $n/2$  elements.
- **Third iteration:** We have  $n/4$  elements.

... and so on. We can see this process when we ran the following:

```
if (target < array[mid])
```

To determine how many steps it would take to get down to 1 element (since  $\$$ ). This tells us that the binary search has a time complexity of  $O(\log n)$ .

This logarithmic time complexity ensures that even for large datasets, the number of steps to find an element (or determine its absence) grows very slowly.

## Space Complexity

Binary search requires only a constant amount of additional memory to perform, which includes storage for a single variable: `mid`. Therefore, regardless of how large our input array is, the space we need (apart from the array itself) remains the same. This translates to  $O(1)$ .

That means that a binary search has a constant space complexity, even when you search large arrays.

## Best, Worst, and Average Cases

- **Best Case:** The best-case scenario for binary search occurs when the target element is right in the middle of the sorted list, meaning we have found the element in just one comparison. Best case time complexity is  $O(1)$ .
- **Worst Case:** The worst-case scenario happens when the target element is either one of the endpoints of the list or is not in the list at all. In such cases, the search would make the maximum number of comparisons,

i.e.,  $O(\log n)$  comparisons. Thus, the worst-case time complexity is  $\Theta(\log n)$ .

- **Average Case:** On average, binary search will have to make multiple comparisons before locating the desired index or concluding the absence of an element. The average case, too, will be bound by  $O(\log n)$

info

In comparison to linear search, which operates in linear time, binary search can vastly outperform it, especially as the dataset grows. However, it's essential to note that binary search **requires** a sorted list, and the time taken to sort a list (if it is not already sorted) should be considered when assessing the overall efficiency for specific applications.

# Max Value

## Finding the Max Value

Let's practice the divide and conquer technique once more. This time, we are going to find the element with the largest value in an array of integers. Create the method `maxValue` that takes an array of integers, and integer representing the left-most element in the array, and another integer representing the right-most element in an array.

```
public static int maxValue(int[] array, int left, int right)
{
}
```

Our base case is when our array has a length of 2. We determine this by subtracting `right` from `left`. If the difference is 1 or less, than the array has two elements. If the element at index `left` is less than the element at index `right`, then return the element at index `left`. Otherwise, return the element at index `right`.

```
public static int maxValue(int[] array, int left, int right)
{
    if (right - left <= 1) {
        return array[left] > array[right] ? array[left] :
array[right];
    }
}
```

For the recursive case, we need to create a few variables. Calculate the mid-point of the array and assign it to `mid`. Create `leftMax` and assign it the value of recursively calling `maxValue` with the left-half of the array (the elements from index `left` to index `mid`). Then create `rightMax` and assign it the value of recursively calling `maxValue` with the right-half of the array (the elements from `mid + 1` to index `right`). Finally return the greater value between `leftMax` and `rightMax`.

```

public static int maxValue(int[] array, int left, int right)
{
    if (right - left <= 1) {
        return array[left] > array[right] ? array[left] : array[right];
    } else {
        int mid = left + (right - left) / 2;
        int leftMax = maxValue(array, left, mid);
        int rightMax = maxValue(array, mid + 1, right);
        return leftMax > rightMax ? leftMax : rightMax;
    }
}

```

This algorithm displays the divide and conquer technique because it repeatedly divides the original array into smaller and smaller arrays. Once these arrays reach a small enough size, then the max value is calculated for each one. These values are then compared to find the largest value in the array.

challenge

## Try This Variation

Create the `minValue` method that takes an array of integers and returns the element with the smallest value. Use recursion and the divide and conquer technique in your algorithm.

### ▼ Solution

```

public static int minValue(int[] array, int left, int right) {
    if (right - left <= 1) {
        return array[left] < array[right] ? array[left] : array[right];
    } else {
        int mid = left + (right - left) / 2;
        int leftMin = minValue(array, left, mid);
        int rightMin = minValue(array, mid + 1, right);
        return leftMin < rightMin ? leftMin : rightMin;
    }
}

```

# Max Value Analysis

In short, the time complexity for all scenarios (worst, best, and average) is  $O(n)$ , while the space complexity for all these scenarios is  $O(\log n)$ . A more in-depth analysis is provided below. We are going to be referencing a recursion tree. In our code for finding the maximum element in an array, the recursion tree provides a good way to understand the execution flow and complexity. Let's consider an example array of length  $n = 8$ , with elements [1, 5, 3, 9, 2, 4, 7, 6].

The root of the recursion tree represents the initial call, which covers the entire array from index 0 to 7. The root then has two children: one representing the left half of the array (from index 0 to 3) and the other representing the right half of the array (from index 4 to 7).

Here is how the recursion tree would look like in this example:

```
tex -hide-clipboard
[1, 5, 3, 9, 2, 4,
7, 6] (0-7)
/
\ [1, 5, 3, 9] (0-3) [2, 4,
7, 6] (4-7) / \
/ \ [1, 5] (0-1) [3, 9] (2-3)
[2, 4] (4-5) [7, 6] (6-7) / \ /
\ / \ \ [1] (0-0) [5]
(1-1) [3] (2-2) [9] (3-3) [2] (4-4) [4] (5-5) [7] (6-6)
[6] (7-7)
```

## Time Complexity:

- The **worst-case** scenario for this algorithm would involve traversing the entire array to find the maximum value. In this case, the function will be called ( $n$ ) times, where ( $n$ ) is the size of the array. Each call takes constant time  $O(1)$  to execute the comparisons and returns. Therefore, the worst-case time complexity would be  $O(n)$ .
- The **best-case** time complexity is also  $O(n)$ . Even if the maximum element is the very first or last element of the array, the algorithm must still traverse the entire array to guarantee that it is indeed the maximum. Just like in the worst-case, each function call takes  $O(1)$  time and there are  $n$  such function calls.
- The **average-case** scenario would also necessitate going through all the elements of the array to find the maximum element, resulting in a time complexity of  $O(n)$ .

## Space Complexity:

- The **worst-case** space complexity of this algorithm is  $O(\log n)$ . This occurs when the recursion tree is deepest, which would happen when you need to go through all the elements in the array. Each recursive call takes up space on the call stack. The maximum depth of the recursion tree would be  $(\log n)$ , resulting in a worst-case space complexity of  $O(\log n)$ .
- The **best-case** space complexity would still be  $O(\log n)$ . This is because, even if the maximum element is found in the first or last position, you would still make the same number of recursive calls and the depth of the recursive call stack would still be  $(\log n)$ .
- Just like the worst and best cases, the **average-case** space complexity is  $O(\log n)$ . This is because the algorithm must traverse the array in a divide-and-conquer manner, regardless of the actual values in the array. Each level of recursion adds a new layer to the call stack, resulting in a maximum depth of  $(\log n)$ .

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- **Define and formulate recurrence relations**
- **Differentiate the master theorem and recurrence relations**
- **Identify the significance of space complexity in addition to time complexity**

# Analyzing Algorithms

We have previously talked about the importance of things like space and time complexity of algorithms. Using these valuable pieces of information we can compare algorithms. Which ones are faster? Which ones take up less memory? This allows us to select the best one for a given situation.

Analyzing non-recursive are fairly straightforward. We often employ techniques like counting the number of operations, assessing loop iterations, etc. This is relatively easy as the code is easy to follow.

## Recursive Algorithms: A Different Beast

Recursion, however, is quite common in many algorithms due to the self-similarity of many data structures. These algorithms present unique challenges. Because the function calls itself with different input sizes, understanding the growth of these calls over time is essential to understanding an algorithm's efficiency.

Key aspects to consider when analyzing recursive algorithms include:

- **Base Case(s):** Every recursive algorithm must have one or more conditions that allow it to terminate. These are known as base cases. Understanding these is crucial, as they determine when the recursion stops.
- **Recurrence Relation:** This defines the relationship between the size of a problem and the size of its sub-problems. The recurrence relation provides a mathematical model that describes the algorithm's behavior.
- **Divide and Conquer:** Many recursive algorithms employ a 'divide and conquer' approach. This means they break the problem down into smaller sub-problems. The manner and efficiency with which an algorithm divides its tasks can greatly affect its overall performance.
- **Stack Space:** Recursive calls utilize stack memory. Thus, it's crucial to consider the space implications of recursion. An algorithm that delves too deep recursively might consume all available stack space, leading to a stack overflow.

# Recurrence Relation

## Defining Recurrence Relation

A **recurrence relation** is an equation that describes the relationship between the values of a function at different input sizes. It is often used to analyze the time complexity of recursive algorithms.

In a recursive algorithm, a problem is broken down into smaller subproblems, which are then solved recursively. The recurrence relation for a recursive algorithm can be used to determine the number of times the recursive function is called, which is a measure of the algorithm's time complexity.

## Fundamentals of Recurrence Relation

Recurrence relations are often used when studying data structures to analyze the time complexity of different operations. A recurrence relation for a function  $T(n)$  expresses its value in terms of one or more smaller values of  $T$ . Here is a simple example for calculating the recurrence relation for a recursive function:

$$T(n) = T(n - 1) + 1$$

The  $T(n - 1)$  represents the smaller subproblem created by the recursive algorithm. This relation suggest that the work done for problem for a size of  $n$  is dependent on the work done for a problem of size  $(n - 1)$ , plus some constant work (denoted by the 1).

## Fibonacci Sequence

Let's further explore recurrence relation with a recursive algorithm we have already used, the Fibonacci sequence:

```
public int fibonacci(int n) {
    if (n <= 1) return n; // Base case
    return fibonacci(n-1) + fibonacci(n-2);
}
```

In this particular algorithm, we are recursing twice — once for `fibonacci(n-1)` and again for `fibonacci(n-2)`. This recurrence relation states that the time complexity of computing the Fibonacci number for a

positive integer  $n$  is equal to the sum of the time complexities of computing the Fibonacci numbers for  $n-1$  and  $n-2$ .

$$T(n) = T(n - 1) + T(n - 2)$$

However, the reality is a bit more nuanced than the equation above. If you look at the fibonacci method, you will see that there are other actions besides recursion. The algorithm is making a comparison, adding two integers, returning 1, etc. All of this work is a constant.

A better expression of the recurrence relation can be described by the equation below. The  $c$  represents the constant time taken for operations other than the recursive calls.

$$T(n) = T(n - 1) + T(n - 2) + c$$

Notice something important: The method makes two recursive calls for each non-base case input. This leads to an exponential growth in the number of operations as  $n$  grows, making this a very inefficient way to compute Fibonacci numbers. We will cover some ways to increase the efficiency for calculating the  $n$ th number of the Fibonacci sequence in another assignment.

Recurrence relations are a powerful tool for analyzing the time complexity of recursive algorithms. They can be used to determine the asymptotic behavior of an algorithm, which can help us to choose the most efficient algorithm for a particular problem.

---

# Master Theorem

## Calculating Time Complexity

We have seen how recurrence relations are used to calculate the time complexity of recursive algorithms. We have also talked about how divide and conquer is a specific kind of recursion. In light of this, there is a specific recurrence relation that applies to divide and conquer recursive algorithms.

This specific recurrence relation is called the **master theorem**. It gives us a direct way to determine the time complexity based on:

- How many subproblems there are.
- The size of each subproblem relative to the original problem.
- The cost of dividing the original problem and combining the subproblem solutions.

We can express the master theorem as a recurrence relation with the following formula:

$$T(n) = a * T(n/b) + f(n)$$

Where:

- $n$  is the size of the input. This is the parameter that determines the asymptotic behavior of the algorithm.
- $a$  is a constant greater than 1. This is the growth factor of the recursive calls.
- $b$  is a constant greater than 1. This is the factor used to reduce the size of each subproblem.
- $n/b$  is the size of each new subproblem.
- $f(n)$  is a function that represents the amount of work done outside of the recursive calls. This commonly entails the time it takes to create subproblems and combine their results into the solution.

## Recursive Fibonacci Algorithm

Although the recursive Fibonacci algorithm does not fit perfectly into the classic mold for the master theorem, we can still use the theorem's principles to understand the algorithm's nature. Recall that we previously expressed the recurrence relation for finding the  $n$ th number of the Fibonacci sequence as:

$$T(n) = T(n - 1) + T(n - 2) + c$$

The  $(n - 1)$  and  $(n - 2)$  represent the creation of smaller subproblems. According to the master theorem, we can divide the size of the input  $(n)$  by a factor  $(b)$  to get the size of the smaller subproblem. If we make this substitution, the recurrence relation becomes:

$$T(n) = T(n/b) + T(n/b) + c$$

Adding two of the same value is equivalent to multiplying that value by 2. We can make this substitution to our recurrence relation, which becomes:

$$T(n) = 2 \times T(n/b) + c$$

The constant  $c$  is defined as the non-recursive actions the algorithm performs. This aligns with  $f(n)$  in the master theorem. Our recurrence relation now becomes:

$$T(n) = a \times T(n/b) + f(n)$$

We can see how the original recurrence relation of the recursive Fibonacci algorithm closely maps to the master theorem. While this is not a rigorous application of the master theorem, it mirrors the theorem's emphasis on the branching factor of recursive algorithms to determine their growth.

However, it is worth noting that there are more efficient algorithms, like the iterative approach or using matrix exponentiation, which can compute the Fibonacci sequence in polynomial or even logarithmic time.

# Merge Sort Analysis

The master theorem provides three distinct cases which can help ascertain the nature of  $T(n)$ , the time complexity of the recursive algorithm. The first thing we need to do is calculate the value for  $d$  such that  $a^d > b^d$ .

**Note**, the values for  $a$  and  $b$  come from the master theorem.

The 3 cases are as follows:

- \* **Case 1:** If  $d < \log_b a$ , then  $T(n) = O(n^d)$ .
- \* **Case 2:** If  $d = \log_b a$ , then  $T(n) = O(n^d \log n)$ .
- \* **Case 3:** If  $d > \log_b a$ , then  $T(n) = O(f(n))$ .

# Merge Sort Analysis

The master theorem can be used to solve a wide variety of recurrence relations that arise in the analysis of divide-and-conquer algorithms. Lets try it with our typical divide and conquer example, merge sort. We know that a merge sort works by recursively dividing the input array into two halves, sorting each half, and then merging the two sorted halves back together.

```
// code snippet from a merge sort
void sort(int arr[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        sort(arr, left, mid);          //create the left half
        sort(arr, mid + 1, right);    //create the right half
        merge(arr, left, mid, right);
    }
}
```

If we apply the master theorem to a merge sort, we can describe its recurrence as:

$$T(n) = 2 \times T(n/2) + O(n)$$

## ▼ Original master theorem

Remember, the original master theorem can be expressed as:

$$T(n) = a * T(n/b) + f(n)$$

Where:

- $n$  is the size of the input.
- $a$  is a constant representing the growth factor of the recursive calls.
- $b$  is a constant representing the factor used to reduce the size of each subproblem.
- $n/b$  is the size of each new subproblem.
- $f(n)$  is a function representing the amount of non-recursive work.

Here is a quick explanation for the substitutions made to the original master theorem:

- We can substitute 2 for  $a$  because we divide the array into two subproblems.
- We can substitute 2 for  $b$  since each subproblem is half the size the original problem.
- We can substitute  $O(n)$  for  $f(n)$  because of the linear time merging process.

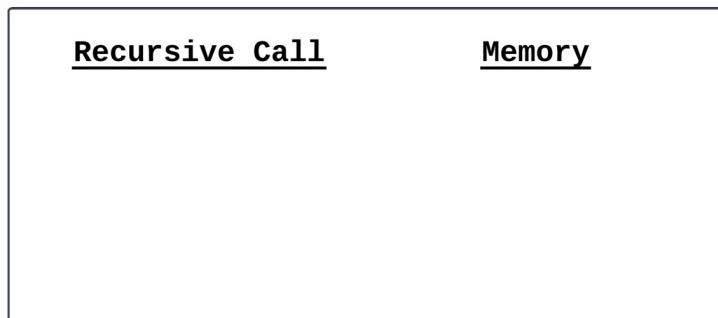
Using the cases at the top of this page, we see that a merge sort falls under **case 2**,  $d = \log_b a$ . The function  $F(n) = O(n^{\log_b a})$  which can be simplified to  $O(n)$ . Therefore, the time complexity of merge sort is  $T(n) = O(n \log n)$

# Space Complexity of Recursive Algorithms

While time complexity often takes center stage in algorithm analysis, understanding the space requirements, especially for recursive algorithms, is equally vital. Every recursive call consumes additional memory, primarily due to the stack frame created during the call. As such, a deep recursive algorithm can quickly lead to a stack overflow if it exhausts the available stack space.

## Stack Frames and Recursion

Every time a method (recursive or not) is called, a new stack frame is created to store local variables, parameters, and bookkeeping information for the method. For recursive methods, each recursive call generates a new stack frame. The total stack space consumed by a recursive algorithm is proportional to the maximum depth of its recursive calls.



The gif shows a stack frame being added to memory with every recursive call for the factorial method.

To determine the space complexity of a recursive algorithm, focus on two main factors:

- 1. Non-recursive space:** This includes space for local variables, constants, and temporary storage, which is not part of the recursive call.
- 2. Recursive space:** The space required for each recursive call, which is directly linked to the depth of recursion.

The overall space complexity is the sum of non-recursive and recursive space.

## Examples

Let's look at some recursive algorithms we already know and calculate the space complexity for each.

### 1. Factorial Function

```
public int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n-1);
}
```

For this function, each call results in a new recursive call (except the base case). Hence, in the worst case, the depth of recursion is  $n$ . Thus, the space complexity is  $O(n)$ .

### 2. Binary Search

```
public int binarySearch(int[] arr, int l, int r, int x) {
    if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x) return mid;
        if (arr[mid] > x) return binarySearch(arr, l, mid -
1, x);
        return binarySearch(arr, mid + 1, r, x);
    }
    return -1;
}
```

Binary search divides the problem size in half with each recursive call. As a result, the maximum depth of recursion is  $\log n$ . The space complexity is thus  $O(\log n)$ .

## Optimization with Tail Recursion

Tail recursion occurs when the recursive call is the last operation in the function. Here is the factorial algorithm. This is an example of tail recursion because the recursive call comes in the last line of the method.

```
public static int factorial(int num) {

    if (num == 1)
        return 1;
    else
        //recursive call is the last operation
        return num * factorial(num - 1);
}
```

Many compilers and interpreters can optimize tail recursive calls to reuse the current function's stack frame for the next call, effectively transforming the recursion into iteration and dramatically reducing space requirements.

For example, the factorial function can be rewritten as a tail-recursive function with an accumulator. When optimized, its space complexity reduces to  $O(1)$ .

```
public static int factorial(int num, int accumulator) {  
  
    if (num == 1)  
        return accumulator;  
    else  
        return factorial(num - 1, num * accumulator);  
}
```

In the vast realm of recursive algorithms, space often matters as much as time. Being adept at analyzing both provides a holistic understanding of an algorithm's efficiency and constraints, allowing for more informed and optimal solutions.

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- Define greedy algorithms
- Identify situations where a greedy approach is appropriate
- Implement a basic greedy solution for the coin change problem
- Define the knapsack problem and differentiate between its 0/1 and fractional versions.

# Greedy Algorithms

## Building Up Instead of Breaking Down

**Greedy algorithms** build up a solution piece by piece, always choosing the next piece that offers the most immediate benefit or profit. This approach is mainly used to solve optimization problems.

Imagine you're at a buffet. A greedy approach would be to always pick the dish that looks most appetizing right now without thinking about what you might choose next or if you're leaving enough room for dessert. You're aiming to get the best immediate satisfaction with each choice.

This can sometimes lead to suboptimal solutions, but it can also be very efficient for problems that have a lot of local optima. Some of the benefits include:

- **Efficiency:** Greedy algorithms often offer a faster solution because they make decisions based on current information without worrying about the consequences. This means they can often reach a solution faster than methods that consider all possibilities before moving forward.
- **Simplicity:** Greedy algorithms are generally easier to describe and implement. They make the most obvious choice at each step of the decision-making process.
- **Optimality:** For some problems, greedy algorithms can find the global optimal solution. However, it's worth noting that while a greedy approach can provide a solution that's good enough, it does not always guarantee the best possible solution for all problems.

## Binary Search vs Greedy Algorithm

Binary search **is not** a greedy algorithm. While both binary search and greedy algorithms make decisions at each step, the rationale and purpose behind those decisions are different.

Binary search is a divide-and-conquer algorithm. In binary search, the goal is to find a specific value in a sorted list. At each step, binary search divides the list in half by comparing the target value to the middle element:

- If the target value is equal to the middle element, you have found the value.
- If the target value is less than the middle element, continue the search on the left half.

- If the target value is greater than the middle element, continue the search on the right half.

This division continues until the value is found or the list is exhausted.

A greedy algorithm, on the other hand, makes the best local choice at each step with the hopes of finding a global optimum. It does not divide the problem into subproblems like binary search but instead makes a series of choices.

While both methods can be efficient for their respective problems, their underlying strategies and purposes are distinct. Binary search is specifically designed for searching in sorted collections, while greedy algorithms are a general approach for solving optimization problems. In some cases, binary search can be used as part of a greedy algorithm. However, binary search itself is not a greedy algorithm.

# Coin Change Problem

## Greedy Algorithm Example

Now that we know what a greedy algorithm is not, lets try to tackle an example of one that is. The main example, that is often introduced when working with a greedy algorithms is the coin change problem.

The **coin change problem** is a problem where you have a certain amount of money and you need to find the minimum number of coins that you need to make up that amount of money. The greedy algorithm for the coin change problem works by choosing the largest coin that does not exceed the amount of money that you need.

Let's walk through the coin change problem using a greedy approach in Java. Note that this greedy method only works when the coin denominations have certain properties. For instance, with US coin denominations (1, 5, 10, 25), the greedy method works. However, it's not always optimal for other denominations.

## Code

Given different coin denominations and a total amount, find the minimum number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

Here are the steps to solve this problem with a greedy approach:

- Always select the largest coin denomination less than the remaining total.
- Deduct the value of the coin from the total.
- Repeat the process until the total is zero or no coins can be taken.

Start by creating the `CoinChangeGreedy` class. **Note**, we are importing the `Arrays` utility so we can sort an array. In the class, create the `coinChange()` method that takes an array of integers (the denominations for the coins) and another integer (the target amount). We also need a `main()` method.

```

import java.util.Arrays;

public class CoinChangeGreedy {

    public static int coinChange(int[] coins, int amount) {

    }

    public static void main(String[] args) {

    }
}

```

In the `main()` method, create an array of integers that has the values for the coin denominations, an integer for the target amount, and a second integer for the result of our greedy algorithm. Finally, print out some context for the results.

```

public static void main(String[] args) {
    int[] coins = {1, 5, 10, 25}; // US coin denominations
    int amount = 63;
    int result = coinChange(coins, amount);

    if (result != -1) {
        System.out.println("Minimum coins needed: " +
                           result);
    } else {
        System.out.println("The amount cannot be represented
                           using the given denominations.");
    }
}

```

In the `coinChange()` method, first sort the array of coins into ascending order. Create a variable to count the number of coins used and initialize its value to `0`. Then calculate the length of the array of coins.

```

public static int coinChange(int[] coins, int amount) {
    // Sort the coins in descending order
    Arrays.sort(coins);
    int count = 0;
    int n = coins.length;

}

```

Remember, greedy algorithms select for the most immediate profit. In this case, that means the algorithm should use as many of the coins with the largest denomination. So, we need to iterate backwards over the array `coins`. If we finish making change before we finish iterating over `coins`, then we want to stop the loop. There is no need to continue if the correct amount has been created.

```
public static int coinChange(int[] coins, int amount) {
    Arrays.sort(coins);
    int count = 0;
    int n = coins.length;

    // Starting from the largest coin denomination
    for (int i = n - 1; i >= 0 && amount > 0; i--) {

    }

    return amount == 0 ? count : -1;
}
```

For each coin in `coins`, we are going to keep adding that coin as long as the value of that coin is greater than or equal to the amount remaining. If we can add a coin, subtract the value of the coin from the amount remaining and increment `count`. Finally, after both loops have run, return `count` if the amount remaining is `0`. If not, return `-1` to indicate that this amount cannot be calculated.

```
public static int coinChange(int[] coins, int amount) {
    Arrays.sort(coins);
    int count = 0;
    int n = coins.length;

    // Starting from the largest coin denomination
    for (int i = n - 1; i >= 0 && amount > 0; i--) {
        while (amount >= coins[i]) { // while we can still
            use coin[i]
                amount -= coins[i];
                count++;
        }
    }

    return amount == 0 ? count : -1;
}
```

## ▼ Code

Your code should look like this:

```

import java.util.Arrays;

public class CoinChangeGreedy {

    public static int coinChange(int[] coins, int amount) {
        // Sort the coins in descending order
        Arrays.sort(coins);
        int count = 0;
        int n = coins.length;

        // Starting from the largest coin denomination
        for (int i = n - 1; i >= 0 && amount > 0; i--) {
            while (amount >= coins[i]) { // while we can still
                use coin[i]
                amount -= coins[i];
                count++;
            }
        }

        return amount == 0 ? count : -1;
    }

    public static void main(String[] args) {
        int[] coins = {1, 5, 10, 25}; // US coin denominations
        int amount = 63;
        int result = coinChange(coins, amount);

        if (result != -1) {
            System.out.println("Minimum coins needed: " +
result);
        } else {
            System.out.println("The amount cannot be represented
using the given denominations.");
        }
    }
}

```

For an amount of 63, the output will be 6 because two quarters (50 cents), one dime (10 cents), and three pennies (3 cents) make 63 cents.

challenge

## Try This Variation

Let's assume that we live in a country who has the following coins: 2, 15, and 25. Modify the `main()` method to reflect these coins, and then try to find the minimum number of coins needed to total 30.

```
public static void main(String[] args) {
    int[] coins = {2, 15, 25}; // US coin denominations
    int amount = 30;
    int result = coinChange(coins, amount);

    if (result != -1) {
        System.out.println("Minimum coins needed: " +
                           result);
    } else {
        System.out.println("The amount cannot be
                           represented using the given denominations.");
    }
}
```

### ▼ What is happening?

A greedy algorithm will always choose the option that provides the most immediate profit. It does not stop and consider the ramifications of the choice. So, the algorithm selects the coin worth 25, which leaves 5 as the remaining amount. There is no combination of the available coins that produces a total of exactly 5.

Again, it's important to emphasize that this greedy approach does not guarantee an optimal solution for all coin denominations. In cases where it's not suitable, other techniques, such as dynamic programming, would be more appropriate.

# Knapsack Problems

## 0/1 Knapsack Problem

Another classic and simple greedy algorithm is the 0/1 knapsack problem. The problem can be understood in terms of resource allocation with a constraint on total capacity, where items have values and weights. The goal is to maximize the value stored in the knapsack without going over the weight limit. The “0/1” part means that you can only take entire items. This is similar to binary, where values can be 0 or they can be 1, but they cannot be some decimal between the two.

The greedy algorithm for the knapsack problem works by iteratively adding the item with the highest value-to-weight ratio to the knapsack until the knapsack is full or no more items can be added. This algorithm is not always optimal, but it is often very efficient.

Here is a step-by-step description of the greedy algorithm for the knapsack problem:

1. Sort the items in decreasing order of value-to-weight ratio.
2. Iterate over the items in sorted order.
3. If the item can be fit into the knapsack, add it to the knapsack.
4. If the item cannot be fit into the knapsack, skip it.
5. Repeat steps 3 and 4 until the knapsack is full or no more items can be added.

## Fractional Knapsack

This variation of the knapsack problem allows you to take a fraction of an item and place it in the knapsack.

The greedy algorithm for the knapsack problem is not always optimal. For example, consider the following set of items:

- \* Item 1: Value = 10, Weight = 5
- \* Item 2: Value = 20, Weight = 10
- \* Item 3: Value = 30, Weight = 15

Let's evaluate how the greedy algorithm for the fractional knapsack problem would behave with the given items. Notice how all items have the same value-to-weight ratio.

- \* Item 1: Value = 10, Weight = 5. ( $\{\text{Value}/\text{Weight}\} == 2$ ) \* Item2 :  
 $Value = 20, Weight = 10. (\{\text{Value}/\text{Weight}\} == 2 \$)$
- \* Item 3: Value = 30, Weight = 15. ( $\{\text{Value}/\text{Weight}\} == 2 \$$ )

If the knapsack has a capacity  $W$ :

- **If  $W \geq 30$ :** The knapsack can fit all items. The maximum value will be  $10 + 20 + 30 = 60$ .
- **If  $15 \leq W < 30$ :** The knapsack can fit Item 1 and Item 2 completely, and a fraction of Item 3. The maximum value will be  $10 + 20 + (\text{fraction of Item 3's value})$ .
- **If  $10 \leq W < 15$ :** The knapsack can fit Item 1 completely and a fraction of Item 2. The maximum value will be  $10 + (\text{fraction of Item 2's value})$ .
- **If  $5 \leq W < 10$ :** The knapsack can only fit Item 1 completely. The maximum value will be 10.
- **If  $W < 5$ :** The knapsack can fit only a fraction of Item 1.

To provide a concrete example: If the knapsack capacity is  $W = 20$ , the algorithm will select Item 1 and Item 2 completely (as they both fit). The value in the knapsack is  $10 + 20$  so far. The remaining weight is 5 but the weight of Item 3 is 15. If we divide the remaining weight in the knapsack by the weight of Item 3 ( $\frac{5}{15}$ ), we get a ratio of  $\frac{1}{3}$ . Apply this ratio to the value of Item 3 ( $\frac{30}{3}$ ) and we get a total value of 40 ( $10 + 20 + 10$ ) in the knapsack.

## Advantages and Disadvantages

Here are some of the advantages and disadvantages of using the greedy algorithm for the knapsack problem:

### Advantages:

- The greedy algorithm is very efficient. It can be used to solve knapsack problems with a large number of items in a reasonable amount of time.
- The greedy algorithm is easy to implement. It does not require any complex data structures or algorithms.

### Disadvantages:

- The greedy algorithm is not always optimal. It can sometimes produce suboptimal solutions.
- The greedy algorithm does not work for all knapsack problems. It only works for knapsack problems where the items can be easily sorted in decreasing order of value-to-weight ratio.

# Knapsack Code

## Fractional Example

Now that we have discussed how the fractional knapsack problem works, let's create a working example. Given weights and values of  $n$  items, we need to put these items in a knapsack of capacity  $W$  to get the maximum total value in the knapsack.

The basic idea of the greedy approach is to calculate the value per unit weight for each item, then choose items based on their value-to-weight ratio, starting with the item that has the highest ratio.

In order for our algorithm to work, we need to first need to create the `Item` class, which has `weight` and `value` attributes. We also need to import the `Arrays` utility for sorting our array of `Items`.

```
import java.util.Arrays;

class Item {
    int weight;
    int value;

    Item(int weight, int value) {
        this.weight = weight;
        this.value = value;
    }
}
```

Create the `FractionalKnapsack` class as well as the `main()` and `fractionalKnapsack()` methods. The `fractionalKnapsack()` method takes integers representing the capacity of the knapsack and the number of possible items. It also takes an array containing objects of type `Item`.

```

import java.util.Arrays;

class Item {
    int weight;
    int value;

    Item(int weight, int value) {
        this.weight = weight;
        this.value = value;
    }
}

public class FractionalKnapsack {

    public static double fractionalKnapsack(int W, Item[] arr,
        int n) {

    }

    public static void main(String[] args) {

    }
}

```

In the `main()` method, set a capacity (`W`) for the knapsack and create an array of `Items` with differing weights and values. Then calculate the length of the array. Finally, print some context for the result of the `fractionalKnapsack()` method.

```

public static void main(String[] args) {
    int W = 50;    // Knapsack capacity
    Item[] arr = {new Item(10, 60), new Item(20, 100), new
    Item(30, 120)};
    int n = arr.length;

    System.out.println("Maximum value in knapsack = " +
        fractionalKnapsack(W, arr, n));
}

```

For the `fractionalKnapsack()` method, first sort the array based on the value-to-weight ratio for each element. The array should be in descending order. Create a `totalValue` variable that will be used to keep track of the value stored in the knapsack. Finally, iterate over the array. For each element, ask if the entire object will fit in the knapsack. If yes, subtract the weight of the element from the capacity of the knapsack, and add the value of the element to `totalValue`.

```

public static double fractionalKnapsack(int W, Item[] arr,
int n) {
    // Sort items based on value-to-weight ratio
    Arrays.sort(arr, (a, b) -> Double.compare((double)
        b.value / b.weight, (double) a.value / a.weight));
    double totalValue = 0;

    for (int i = 0; i < n; i++) {
        // Add entire item if it doesn't exceed capacity
        if (W - arr[i].weight >= 0) {
            W -= arr[i].weight;
            totalValue += arr[i].value;
        }
    }

    return totalValue;
}

```

If the entire element cannot fit into the knapsack, we need to create an `else` branch for the conditional. Calculate the fraction of the element that could fit into the knapsack. Multiple the `value` and `weight` attributes by the fraction, and add the fractional value to `totalValue` and the fractional weight to the knapsack. After adding the fractional item to the knapsack, break out of the loop. Finally, return `totalValue`.

```

public static double fractionalKnapsack(int W, Item[] arr,
int n) {
    // Sort items based on value-to-weight ratio
    Arrays.sort(arr, (a, b) -> Double.compare((double)
        b.value / b.weight, (double) a.value / a.weight));
    double totalValue = 0;

    for (int i = 0; i < n; i++) {
        // Add entire item if it doesn't exceed capacity
        if (W - arr[i].weight >= 0) {
            W -= arr[i].weight;
            totalValue += arr[i].value;
        } else { // Else add a fraction of it
            double fraction = (double) W / arr[i].weight;
            totalValue += arr[i].value * fraction;
            W = (int) (W - (arr[i].weight * fraction));
            break;
        }
    }

    return totalValue;
}

```

## ▼ Code

Your code should look like this:

```
import java.util.Arrays;

class Item {
    int weight;
    int value;

    Item(int weight, int value) {
        this.weight = weight;
        this.value = value;
    }
}

public class FractionalKnapsack {

    // Function to get the maximum value in the knapsack
    public static double fractionalKnapsack(int W, Item[] arr,
  int n) {
        // Sort items based on value-to-weight ratio
        Arrays.sort(arr, (a, b) -> Double.compare((double)
            b.value / b.weight, (double) a.value / a.weight));

        double totalValue = 0;
        for (int i = 0; i < n; i++) {
            // If adding the entire item doesn't exceed
            // capacity, add it
            if (W - arr[i].weight >= 0) {
                W -= arr[i].weight;
                totalValue += arr[i].value;
            } else { // Else add a fraction of it
                double fraction = (double) W / arr[i].weight;
                totalValue += arr[i].value * fraction;
                W = (int) (W - (arr[i].weight * fraction));
                break;
            }
        }

        return totalValue;
    }

    public static void main(String[] args) {
        int W = 50;      // Knapsack capacity
        Item[] arr = {new Item(10, 60), new Item(20, 100), new
                      Item(30, 120)};
        int n = arr.length;
```

```
        System.out.println("Maximum value in knapsack = " +  
            fractionalKnapsack(W, arr, n));  
    }  
}
```

For the given example, the output will be 240.0. The maximum value is obtained by choosing the entire first item (60), the entire second item (100), and 2/3 of the third item (80).

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- Differentiate dynamic programming from recursive and iterative solutions
- Differentiate between memoization and tabulation
- Compare iterative and recursive approach to a problem

# Limits of Recursion

## Fibonacci Series

Let's return to a recursive algorithm for the Fibonacci sequence. Copy and paste this code into the IDE to the left.

```
public static int fib(int n) {  
    if (n == 1 || n == 0) {  
        return n;  
    }  
    return fib(n-1) + fib(n-2);  
}
```

First, run the program when asking for the fifth number in the Fibonacci sequence. The program compiles and runs relatively quickly.

Now, let's find the fortieth number in the sequence. In the `main` method, change the value of `n` to `40`. Run the program once more. You should notice that it is a bit slower than before.

```
int n = 40;
```

Finally, let's call the `fib` method several times, each time calculating the fortieth number in the Fibonacci sequence. The results should be dramatically slower than the ones from above.

```
System.out.println("Fibonacci number at position " + n +  
" is: " + fib(n));  
System.out.println("Fibonacci number at position " + n +  
" is: " + fib(n));  
System.out.println("Fibonacci number at position " + n +  
" is: " + fib(n));  
System.out.println("Fibonacci number at position " + n +  
" is: " + fib(n));  
System.out.println("Fibonacci number at position " + n +  
" is: " + fib(n));
```

Despite the usefulness of recursive algorithms, they are not always the most performant solution. However, there are things that we can do to increase the performance of recursion.

challenge

## Try This Variation

Change `n` to a large number like `100`. You do not need to print out the results of the function call multiple times.

```
int n = 100;
```

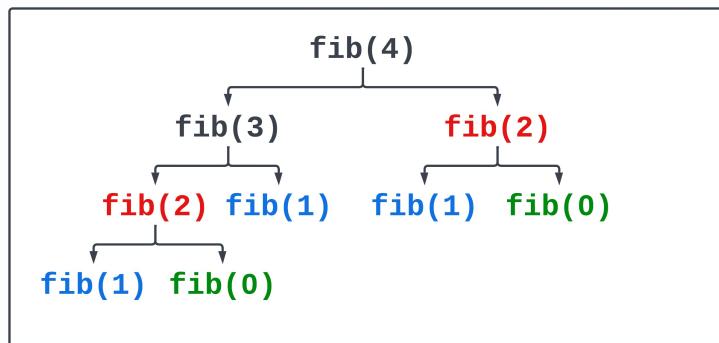
### ▼ What is happening?

We previously discussed that algorithms that make too many recursive calls can fill up the memory on the stack. The timeout message you see is a safety precaution that keeps programs from causing a stack overflow error.

# Dynamic Programming

## Why is `fib()` so Slow?

The time complexity of the `fib` algorithm from the previous page is approximately  $O(2^n)$ . Here is a recursive tree for `fib(4)`:



The image shows a tree representation for the method call “`fib(4)`”. Several of the subsequent method calls are repeated. `fib(2)` is repeated twice and is colored red. `fib(1)` appears three times and is colored blue. `fib(0)` appears twice and is colored green.

Even though we are only looking for the fourth number in the sequence, our algorithm is making nine recursive calls. This number is so big because several calculations are being repeated. We see `fib(2)` two times, `fib(1)` three times, and `fib(0)` two times. If we did not have to repeat ourselves so many times, our algorithm would not be so slow. That is where dynamic programming comes into play.

# Dynamic Programming

**Dynamic programming** is a technique for solving problems by breaking them down into smaller subproblems, storing the solutions to the subproblems, and then using the stored solutions to solve the original problem. This allows us to solve the original problem more efficiently, since we do not have to repeat any calculations.

Here is an example of how dynamic programming can be used to solve the Fibonacci sequence problem. Break down the problem into subproblems, identifying the base case(s) and the recursive case.

- The first Fibonacci number is 0 (base case).

- The second Fibonacci number is 1 (base case).
- The  $n$ th Fibonacci number is the sum of the  $(n - 1)$ th and  $(n - 2)$ th Fibonacci numbers (recursive case).

We do not need to perform any calculations for the base case. Those solutions are hard-coded into our algorithm. For every recursive case, we are going to perform the calculation and then store that value. For this example, we are going to use a table to represent storing the calculations. The table might look like this:

| <b>Value of <math>n</math></b> | <b>Stored Value</b> |
|--------------------------------|---------------------|
| 2                              | 1                   |
| 3                              | 2                   |
| 4                              | 3                   |
| 5                              | 5                   |

Once we have the solutions to the first few subproblems, we can solve the original problem by looking up the solution for the desired  $n$ th Fibonacci number in the table. For example, the solution to the 5th Fibonacci number is 5, which can be found in the table.

Dynamic programming is a powerful technique that can be used to solve many different kinds of problems. It is often much more efficient than other techniques, such as brute force or backtracking. We are going to use Fibonacci to show dynamic programming using recursion and iteration.

# Memoization

## Applying Dynamic Programming Principles

To increase the performance of our algorithm, we are going to apply some dynamic programming principles. In particular, we are going to use memoization. **Memoization** is an optimization technique used to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. This is commonly referred to as a top-down approach.

Let's start by creating the `FibonacciRecursive` class. In it, we need a `main` method and a `fib` method. In addition, we are going to create the global variable `memo` (an array of integers) for the memoization.

```
public class FibonacciRecursive {
    private static int[] memo;

    public static int fib(int n) {

    }

    public static void main(String[] args) {
    }
}
```

Next, we are going to create the variable `n` and set it to 40. Remember, we saw how finding the 40th number in the Fibonacci sequence caused our code to slow down. Set the size of `memo` to one larger than the size of `n`. To help us keep track of which elements have been filled with a Fibonacci number, initialize all of the elements of `memo` to -1. Finally, print a message that provides some context to the output of `fib`.

```
public class FibonacciRecursive {
    private static int[] memo;

    public static int fib(int n) {

    }

    public static void main(String[] args) {
        int n = 40;
        memo = new int[n+1];

        for (int i = 0; i <= n; i++) {
            memo[i] = -1;
        }

        System.out.println("Fibonacci number at position " + n +
            " is: " + fib(n));
    }
}
```

Create the `fib` method that takes an integer as a parameter. Before we check our base case, we first are going to see if the calculation was previously performed and stored in the `memo` array. If the element at `memo[n]` is not a `-1`, then it is the result of `fib(n)`. Return this value. Now we can get to the base and recursive cases. If `n` is less than or equal to 1, then store `n` in `memo[n]`. In the recursive case, set `memo[n]` to the traditional recursive calls for the Fibonacci sequence. Finally, return `memo[n]`.

```

public class FibonacciRecursive {
    private static int[] memo;

    public static int fib(int n) {
        if (memo[n] != -1) {
            return memo[n];
        }

        if (n <= 1) {
            memo[n] = n;
        } else {
            memo[n] = fib(n-1) + fib(n-2);
        }

        return memo[n];
    }

    public static void main(String[] args) {
        int n = 40;
        memo = new int[n+1];

        for (int i = 0; i <= n; i++) {
            memo[i] = -1;
        }

        System.out.println("Fibonacci number at position " + n +
            " is: " + fib(n));
    }
}

```

If we run our code now, it should have no trouble at all with calculating `fib(100)`.

challenge

## Try This Variation

Modify algorithm so that it can calculate the 100th number in the Fibonacci sequence. Print it out several times. **Hint**, you need to change the data type for the `fib()` method to be a `long` since the 100th number is too big for the `int` data type.

### ▼ Solution

If `fib()` now returns a `long`, then `memo` also needs to be of type `long`. This includes when you create the variable and assign it a length.

```

public class FibonacciRecursive {
    private static long[] memo;

    public static long fib(int n) {
        if (memo[n] != -1) {
            return memo[n];
        }

        if (n <= 1) {
            memo[n] = n;
        } else {
            memo[n] = fib(n-1) + fib(n-2);
        }

        return memo[n];
    }

    public static void main(String[] args) {
        int n = 100;
        memo = new long[n+1];

        for (int i = 0; i <= n; i++) {
            memo[i] = -1;
        }

        System.out.println("Fibonacci number at position " +
n + " is: " + fib(n));
        System.out.println("Fibonacci number at position " +
n + " is: " + fib(n));
        System.out.println("Fibonacci number at position " +
n + " is: " + fib(n));
        System.out.println("Fibonacci number at position " +
n + " is: " + fib(n));
        System.out.println("Fibonacci number at position " +
n + " is: " + fib(n));
        System.out.println("Fibonacci number at position " +
n + " is: " + fib(n));
    }
}

```

## Analysis

The use of dynamic programming principles means that the time complexity of `fib()` is now  $O(n)$ . The space complexity becomes a bit more complicated as we now introduce greater auxiliary space due to memoization. The space complexity for the memoization table is  $O(n)$ , and the space complexity for the recursive call stack is also  $O(n)$ .

Although the recursive approach seems more complicated, it's a good illustration of the power of memoization and how a naive (it is called "naive" because calculations are forgotten after each recursive call) recursive solution can be optimized using dynamic programming principles.

# Iterative Dynamic Programming

## Iterative Fibonacci Sequence

Recursion is not the only way to calculate a number from the Fibonacci sequence, and an iterative approach also stands to benefit from implementing dynamic programming. Instead of starting from a high number and breaking it down into smaller sub-problems (like recursion), we begin from the smallest problems and build-up. Create the method `fib()` that returns an integer and also takes an integer as a parameter. Just like the recursive approach, we know that `fib(0)` or `fib(1)` do not require any calculations. Return 0 or 1 in these respective cases.

```
public static int fib(int n) {  
    if (n <= 1) {  
        return n;  
    }  
}
```

If `n` is greater than 1, then we need to create an array of integers of size `n + 1` to store our calculations. **Note**, since there is no recursion, we do not need to make this array a global variable. Instead, it can reside in the method because `fib()` will only be called once. Set `fib(0)` to 0 and `fib(1)` to 1. This way we are ready to start calculating values for when `n` is greater than or equal to 2.

```
public static int fib(int n) {  
    if (n <= 1) {  
        return n;  
    }  
  
    int[] memo = new int[n + 1];  
    memo[0] = 0;  
    memo[1] = 1;  
}
```

Finally, iterate from 2 (because the first two elements in the array already have values) up to and including `n`. For each value `i`, calculate the Fibonacci number using the two previous elements in the array. After the loop has finished, return the element at index `n`.

```
public static int fib(int n) {
    if (n <= 1) {
        return n;
    }

    int[] memo = new int[n + 1];
    memo[0] = 0;
    memo[1] = 1;

    for (int i = 2; i <= n; i++) {
        memo[i] = memo[i-1] + memo[i-2];
    }

    return memo[n];
}
```

Just as before, we are able to calculate `fib(40)` without much of a delay.

Compared to the recursive counterpart, the iterative `fib()` method is a more efficient algorithm. The time complexity is still  $O(n)$ , but the space complexity is reduced. You still have a memoization table with a space complexity of  $O(n)$ , but there is no recursive call stack. The iterative `fib()` is only called one time.

The above approach efficiently computes the Fibonacci of  $n$  in linear time. This is much faster than the naive recursive method, especially for large values of  $n$ .

# Tabulation

## Dynamic Programming with a 2D Array

Another way to implement dynamic programming is through tabulation. That is, we use a 2D array to store data in a table, where the elements are arranged into rows and columns. Each cell in the table contains solutions to all of the sub-problems that make up a larger problem. This is commonly referred to as a bottom-up approach.

Two-dimensional arrays (**2D arrays**) are essentially arrays of arrays. They are extremely useful for representing data that naturally fits in a table, such as a chess board or a grid. In dynamic programming, 2D arrays can be used to store values that correspond to solutions of sub-problems in a grid-like fashion. Let's look at an example to illustrate this idea.

## Longest Common Subsequence (LCS)

Suppose we are trying to find the length of the longest common subsequence between two strings A and B. A subsequence is a sequence of characters that appear in the same order in the string, but not necessarily consecutively. For example, look at the following strings:

```
tex -hide-clipboard "BACBDAB" "BDCAB"
```

The longest common subsequence for these strings would be "BCAB". Calculating the length of the longest common subsequence is a common problem in computer science as it lends itself to dynamic programming. Using a naive recursive approach would result in many redundant calculations. To more efficiently solve this problem, we are going to use an iterative approach with tabulation.

With dynamic programming, we can store the solutions to sub-problems in a 2D array `dp`, where `dp[i][j]` will contain the length of LCS of prefixes `A[0..i-1]` and `B[0..j-1]`.

Start by creating the `longestCommonSubsequence` method which takes two strings and returns an integer. Calculate the length of each string, and create the 2D array `dp` where each array is one element longer than their respective strings.

```

public class LCS {
    public static int longestCommonSubsequence(String A, String B) {
        // Get the length of each input string.
        int m = A.length();
        int n = B.length();

        // Initialize a 2D array to store the LCS lengths.
        // dp[i][j] will contain the length of LCS of prefixes
        A[0..i-1] and B[0..j-1].
        int[][] dp = new int[m+1][n+1];

    }
}

```

Iterate over the two arrays. If any of the cells are in the first row ( $i == 0$ ) or first column ( $j == 0$ ) then set the value of the element to 0.

```

public class LCS {
    public static int longestCommonSubsequence(String A, String B) {
        // Get the length of each input string.
        int m = A.length();
        int n = B.length();

        // Initialize a 2D array to store the LCS lengths.
        // dp[i][j] will contain the length of LCS of prefixes
        A[0..i-1] and B[0..j-1].
        int[][] dp = new int[m+1][n+1];

        // Loop through each cell in the 2D array.
        for(int i = 0; i <= m; i++) {
            for(int j = 0; j <= n; j++) {
                // Base case: If either of the strings is empty,
                LCS length is 0.
                if(i == 0 || j == 0) {
                    dp[i][j] = 0;
                }
            }
        }
    }
}

```

It will be helpful to visualize what this table looks like and how it relates to the strings. Here is the table with the first row and column filled with zeros. Each character of the first string represents a row, while the characters in the second string represent the columns.

|   | B | D | C | A | B |
|---|---|---|---|---|---|
| B | 0 |   |   |   |   |
| A | 0 |   |   |   |   |
| C | 0 |   |   |   |   |
| B | 0 |   |   |   |   |
| D | 0 |   |   |   |   |
| A | 0 |   |   |   |   |
| B | 0 |   |   |   |   |

The image depicts a table. The letters “BACBDAB” are along the far left side of table. The letter “BDCAB” are along the top of the table. Cells in the first row and column are filled with zeros. All of the other cells are empty.

If the cell in the table is not in the first row or column, then determine if the two characters are the same. If so, set the value of the current cell to the value of the cell up one row and one column to the left plus 1.

```

public class LCS {
    public static int longestCommonSubsequence(String A, String B) {
        // Get the length of each input string.
        int m = A.length();
        int n = B.length();

        // Initialize a 2D array to store the LCS lengths.
        // dp[i][j] will contain the length of LCS of prefixes
        // A[0..i-1] and B[0..j-1].
        int[][] dp = new int[m+1][n+1];

        // Loop through each cell in the 2D array.
        for(int i = 0; i <= m; i++) {
            for(int j = 0; j <= n; j++) {
                // Base case: If either of the strings is empty,
                // LCS length is 0.
                if(i == 0 || j == 0) {
                    dp[i][j] = 0;
                }
                // If the corresponding characters in A and B
                // are equal,
                // then we can extend the LCS for A[0..i-2] and
                // B[0..j-2] by 1.
                else if(A.charAt(i-1) == B.charAt(j-1)) {
                    dp[i][j] = dp[i-1][j-1] + 1;
                }
            }
        }
    }
}

```

In our example, the two strings both start with the character “B”. Our current cell (the blue cell) represents the comparison of characters “B” and “B”. Since they are the same, take the value of the red cell (up one row and one column to the left) and increment by 1. That means the value of the blue cell is 1 since the value of the red cell was 0.

|   | B | D | C | A | B |
|---|---|---|---|---|---|
| B | 0 | 0 | 0 | 0 | 0 |
| A | 0 |   |   |   |   |
| C | 0 |   |   |   |   |
| B | 0 |   |   |   |   |
| D | 0 |   |   |   |   |
| A | 0 |   |   |   |   |
| B | 0 |   |   |   |   |

The image depicts a table. The cell at the intersection of “B” and “B” is blue and has the value of 1. The cell up and to the left of the blue cell is red and has the value of 0.

If the two characters that are being compared are different, then the value of the current cell becomes whatever value is the largest between the cell directly to the left and the cell directly above.

```

public class LCS {
    public static int longestCommonSubsequence(String A, String B) {
        // Get the length of each input string.
        int m = A.length();
        int n = B.length();

        // Initialize a 2D array to store the LCS lengths.
        // dp[i][j] will contain the length of LCS of prefixes
        // A[0..i-1] and B[0..j-1].
        int[][] dp = new int[m+1][n+1];

        // Loop through each cell in the 2D array.
        for(int i = 0; i <= m; i++) {
            for(int j = 0; j <= n; j++) {
                // Base case: If either of the strings is empty,
                // LCS length is 0.
                if(i == 0 || j == 0) {
                    dp[i][j] = 0;
                }
                // If the corresponding characters in A and B
                // are equal,
                // then we can extend the LCS for A[0..i-2] and
                // B[0..j-2] by 1.
                else if(A.charAt(i-1) == B.charAt(j-1)) {
                    dp[i][j] = dp[i-1][j-1] + 1;
                }
                // If the corresponding characters in A and B
                // are different,
                // take the maximum LCS length between A[0..i-1]
                // & B[0..j-2] or A[0..i-2] & B[0..j-1].
                else {
                    dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
                }
            }
        }
    }
}

```

The next iteration has the current cell (blue) comparing the characters “B” and “D”. Since they are not the same, look to the cell to the left and the cell directly above (red). Take the larger of these two values and assign it to the current cell.

|   | B | D | C | A | B |
|---|---|---|---|---|---|
| B | 0 | 0 | 0 | 0 | 0 |
| A | 0 |   |   |   |   |
| C | 0 |   |   |   |   |
| B | 0 |   |   |   |   |
| D | 0 |   |   |   |   |
| A | 0 |   |   |   |   |
| B | 0 |   |   |   |   |

The image depicts the next step in the algorithm. The current cell (blue) is comparing the characters “B” and “D”. This cell has the value of 1. The cell to the left is red and has the value 1. The cell above is red and has the value 0.

As you continue to fill out the table in this manner, you will see the following as the final result:

|   | B | D | C | A | B |
|---|---|---|---|---|---|
| B | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 1 | 2 | 2 |
| B | 0 | 1 | 1 | 2 | 2 |
| D | 0 | 1 | 2 | 2 | 2 |
| A | 0 | 1 | 2 | 2 | 3 |
| B | 0 | 1 | 2 | 2 | 3 |

The image depicts the filled out with all of the calculations from the algorithm. The bottom-right cell is blue and has the number 4.

Continue iterating through the array and, when the loops have finished, return the element in the bottom-right corner of the table. This cell represents the length of the longest common subsequence between the two strings.

```

public class LCS {
    public static int longestCommonSubsequence(String A, String
B) {
    // Get the length of each input string.
    int m = A.length();
    int n = B.length();

    // Initialize a 2D array to store the LCS lengths.
    // dp[i][j] will contain the length of LCS of prefixes
    A[0..i-1] and B[0..j-1].
    int[][] dp = new int[m+1][n+1];

    // Loop through each cell in the 2D array.
    for(int i = 0; i <= m; i++) {
        for(int j = 0; j <= n; j++) {
            // Base case: If either of the strings is empty,
            LCS length is 0.
            if(i == 0 || j == 0) {
                dp[i][j] = 0;
            }
            // If the corresponding characters in A and B
            are equal,
            // then we can extend the LCS for A[0..i-2] and
            B[0..j-2] by 1.
            else if(A.charAt(i-1) == B.charAt(j-1)) {
                dp[i][j] = dp[i-1][j-1] + 1;
            }
            // If the corresponding characters in A and B
            are different,
            // take the maximum LCS length between A[0..i-1]
            & B[0..j-2] or A[0..i-2] & B[0..j-1].
            else {
                dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }

    // The value at dp[m][n] contains the length of LCS for
    A and B.
    return dp[m][n];
}

```

In this code, we initialized a 2D array `dp` with dimensions  $(m+1) \times (n+1)$ , where  $m$  and  $n$  are the lengths of strings  $A$  and  $B$ , respectively. We then filled up the array row by row, using the solutions to smaller sub-problems to solve larger sub-problems. Finally,  $dp[m][n]$  contains the length of the LCS of  $A$  and  $B$ .

By using a 2D array, we have eliminated the need for redundant calculations, and this program will run in  $O(m * n)$  time, which is much more efficient than the naive recursive approach.



# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- **Identify the underlying mechanics of each of the simple sorting algorithms.**
- **Implement selection, bubble and insertion sorts in Java.**
- **Analyze the time and space complexities of each algorithm.**
- **Identify practical scenarios best suited for each algorithm.**

# Introduction to Sorting

## Importance of Sorting

Sorting is one of the fundamental operations in computer science, and it plays a crucial role in various applications, ranging from database management to computer graphics. Understanding basic sorting algorithms not only gives insights into how data can be organized efficiently, but also provides a foundation for more complex algorithms and structures.

Recall that a binary search is more efficient than linear search. However, a binary search requires a sorted array in order to work. Understanding how to efficiently sort data structures allows you to make use of other, important algorithms.

## Sorting Algorithms

In this assignment, we will dive into three basic sorting algorithms: **selection sort**, **bubble sort**, and **insertion sort**. You will implement them, compare their performances, and delve deep into their complexity analyses.

### Selection Sort

Selection sort works by repeatedly selecting the smallest (or largest, depending on the ordering) element from the unsorted part of the array and swapping it with the first unsorted element. Click on the gray canvas below to start the selection sort animation.



**Tasks:**

**1. Implementation:**

- Implement a selection sort in Java.

**2. Analysis:**

- Describe the steps involved in a single iteration of the selection sort.
- Analyze and determine the best-case, worst-case, and average-case time complexities for selection sort.

### **Bubble Sort**

Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. The pass through the list is repeated until the list is sorted. Click on the gray canvas below to start the bubble sort animation.



**Tasks:**

**1. Implementation:**

- Implement a bubble sort in Java.

**2. Analysis:**

- Describe the key operation that makes the largest unsorted element “bubble up” to its correct position.
- Analyze and determine the best-case, worst-case, and average-case time complexities for bubble sort.

## **Insertion Sort**

Insertion sort is a simple sorting algorithm that works by repeatedly inserting an unsorted element into a sorted list. The algorithm starts with the first element in the array, which is considered to be sorted. Then, each subsequent element is taken in turn and inserted into the sorted list in the correct position. To do this, the algorithm compares the new element to each element in the sorted list, starting with the first element. If the new element is smaller than the current element, it is swapped with the current element. This process continues until the new element is larger than or equal to all of the elements in the sorted list. Click on the gray canvas below to start the insertion sort animation.



**Tasks:**

**1. Implementation:**

- Implement an insertion sort in Java.

**2. Analysis:**

- Describe the steps involved in a single iteration of the insertion sort.
- Analyze and determine the best-case, worst-case, and average-case time complexities for insertion sort.

# Selection Sort

## Selection Sort: Detailed Implementation

The **Selection Sort** algorithm divides the input list into two parts: a sorted and an unsorted sublist. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm repeatedly selects the smallest (or largest, depending on sorting order) element from the unsorted sublist and swaps it with the leftmost unsorted element, thereby extending the sorted sublist one element at a time. Selection sort is neither a greedy algorithm nor a divide-and-conquer algorithm. It follows a straightforward, iterative approach.

Info

### Sorting Mechanics

The following gif demonstrates how a selection sort works:

|  |   |
|--|---|
|  | 8 |
|  | 5 |
|  | 2 |
|  | 6 |
|  | 9 |
|  | 3 |
|  | 1 |
|  | 4 |
|  | 0 |
|  | 7 |

By [Joestape89](#), CC BY-SA 3.0, [Link](#)

## Steps

1. Start with the first element of the list, assuming it to be the minimum.
2. Move through the list and find the smallest element.

3. Swap the found minimum element with the first element.
4. Move to the next element and repeat the process until the entire list is sorted.

## Java Implementation

Create the `selectionSort` method that takes an array of integers. The method does not return a value. The first step is to calculate the length of the array that is to be sorted.

```
public static void selectionSort(int[] arr) {  
    int n = arr.length;  
  
}
```

Iterate over the array. Assume that the smallest value in the array is located at the same index as the looping variable (in this case `i`).

```
public static void selectionSort(int[] arr) {  
    int n = arr.length;  
  
    for (int i = 0; i < n - 1; i++) {  
        // Assume the smallest value to be at position 'i'  
        int minIndex = i;  
  
    }  
}
```

Loop over the array one more time. This time, however, start at `i + 1`. Be sure to stop the loop one element before the end of the array. If not, you will get an index out of bounds error. If the element at index `j` is less than the element at `minIndex`, set `minIndex` to `j`.

```

public static void selectionSort(int[] arr) {
    int n = arr.length;

    for (int i = 0; i < n - 1; i++) {
        // Assume the smallest value to be at position 'i'
        int minIndex = i;

        // Iterate over the array to find the actual minimum
value in the unsorted sublist
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

    }
}

```

After the second loop has finished running, swap the elements at indices `minIndex` and `i`.

```

public static void selectionSort(int[] arr) {
    int n = arr.length;

    for (int i = 0; i < n - 1; i++) {
        // Assume the smallest value to be at position 'i'
        int minIndex = i;

        // Iterate over the array to find the actual minimum
value in the unsorted sublist
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        // Swap the found minimum element with the first
element
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}

```

In the `main` method, your code will be provided with the unsorted array:

```
tex -hide-clipboard [64, 25, 12, 22, 11]
```

Your code should produce:

```
tex -hide-clipboard Sorted array: 11 12 22 25 64
```

The `selectionSort` function handles the sorting process. It starts with the leftmost element and goes through the list to find the minimum element. Once the minimum element in the unsorted sublist is found, it is swapped with the first unsorted element. This process continues until the list is entirely sorted.

The outer loop (`for (int i = 0; i < n - 1; i++)`) effectively places the smallest element from the unsorted sublist in its correct position in the sorted sublist. The inner loop (`for (int j = i + 1; j < n; j++)`) is responsible for finding the minimum element from the unsorted sublist.

---

# Selection Sort Analysis

## A Single Iteration of Selection Sort

Each iteration of the selection sort algorithm consists of the following steps:

1. **Initialization:** At the start of each iteration, the algorithm designates the first unsorted element as the minimum (or maximum, depending on the desired order).
2. **Searching for the Actual Minimum:** The algorithm then iteratively compares this designated minimum with each subsequent unsorted element. If it encounters an element smaller than the designated minimum, it updates its notion of the minimum to this new element.
3. **Swapping:** Once the true minimum of the unsorted segment is found, the algorithm swaps it with the first unsorted element. This effectively places the smallest unsorted element in its correct final position in the sorted segment of the list.
4. **Progression:** The boundary between the sorted and unsorted segments moves one element to the right.

By the end of each iteration, the smallest element from the unsorted segment is placed in its correct position in the sorted segment.

## Time Complexity Analysis

- **Best Case:** Even if the list is already sorted, the algorithm still goes through the motions of finding the minimum element in the unsorted segment for each iteration. This gives a best-case time complexity of  $n \times n$  or  $O(n^2)$ .
- **Worst Case:** In the worst case, the list is in reverse order. However, selection sort will still perform almost the same number of operations as in the best case since it does not exploit any existing order in the list. Thus, the worst-case time complexity is also  $O(n^2)$ .
- **Average Case:** On average, for any random arrangement of numbers, the algorithm will still need to compare elements and search for the minimum element in the unsorted segment for each iteration. This leads to an average-case time complexity of  $O(n^2)$ .

## Space Complexity Analysis

The algorithm sorts the list in-place, meaning it does not use any additional storage that grows with the size of the input list. Thus, the space complexity of selection sort is  $O(1)$ , meaning, it uses a constant amount of extra space.

Selection sort, while easy to understand and implement, is not efficient for large lists due to its quadratic time complexity. Its main advantage is its simplicity and the fact that it sorts in-place, requiring no additional space

# Bubble Sort

## Bubble Sort: Detailed Implementation

**Bubble sort**, named due to elements “bubbling up” to their correct positions, is one of the simplest sorting algorithms. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is entirely sorted.

Info

### Sorting Mechanics

The following gif demonstrates how a bubble sort works:



6 5 3 1 8 7 2 4

[CC BY-SA 3.0, Link](#)

## Steps

1. **Traversal & Comparison:** Begin from the first element of the list and compare it with the adjacent element.
2. **Swapping:** If the current element is larger than the next one (for ascending order), they are swapped.
3. **Repeat:** This continues for the entire length of the list, ensuring the largest element “bubbles up” to the end.
4. **Decreasing the Length:** With each outer iteration, the length of the inner loop decreases by one since the largest elements from the previous iterations are already correctly placed.

## Java Implementation

Create the method `bubbleSort` that takes an array of integers and returns nothing. Calculate the length of the array passed to the sorting algorithm.

```
public static void bubbleSort(int[] arr) {  
    int n = arr.length;  
  
}
```

Create a set of nested loops. The outer loop iterates over the array, stopping one element before the end of the array. We will see why in a bit. The inner loop starts at the beginning of the array but stops at  $n - i - 1$ . That means the inner loops runs one fewer time with each pass.

```
public static void bubbleSort(int[] arr) {  
    int n = arr.length;  
  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = 0; j < n - i - 1; j++) {  
  
    }  
}
```

Compare the elements at indices  $j$  and  $j + 1$ . This is why we stop the outer loop from running to the end of the loop; we would get an index out of bounds error otherwise. If `arr[j]` is greater than `arr[j + 1]`, swap the two values.

```
public static void bubbleSort(int[] arr) {  
    int n = arr.length;  
  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = 0; j < n - i - 1; j++) {  
            if (arr[j] > arr[j + 1]) {  
                // swap arr[j] and arr[j+1]  
                int temp = arr[j];  
                arr[j] = arr[j + 1];  
                arr[j + 1] = temp;  
            }  
        }  
    }  
}
```

In the `main` method, your code will be provided with the unsorted array:

```
tex -hide-clipboard [62, 88, 17, 95, 34]
```

Your code should produce:

```
tex -hide-clipboard Sorted array: 17 34 62 88 95
```

In the provided Java code, the function `bubbleSort` manages the sorting task. The outer loop runs  $n - 1$  times, where  $n$  is the number of elements in the list. The inner loop, which is responsible for comparing and potentially swapping elements, runs  $n - i - 1$  times during the  $i$ -th outer iteration.

With each complete traversal of the inner loop, the largest unsorted element has “bubbled up” to its correct position. Thus, with each iteration of the outer loop, one more element is correctly placed, and the inner loop can consider one less element.

# Bubble Sort Analysis

**Bubble sort**, while intuitive, is often less efficient for larger lists compared to more advanced sorting algorithms. Understanding its time and space complexities can shed light on why this is the case and in what scenarios it might be useful.

## Time Complexity Analysis

- **Best Case:** This occurs when the list is already sorted. For bubble sort, even in this scenario, the algorithm will still traverse the list to check if any swaps are needed. If an optimized version is used that checks if any swaps were made in a pass and stops if none were made, then the best-case time complexity is  $(O(n))$ . Without this optimization, it remains  $O(n^2)$ .
- **Worst Case:** The list is inversely sorted, i.e., in reverse order. The algorithm has to swap every adjacent pair for the first pass, then every pair except the last one for the next pass, and so on. This gives a worst-case time complexity of  $O(n^2)$ .
- **Average Case:** For a randomly ordered list of length  $n$ , the average number of comparisons and swaps in each pass is roughly half of the list length. As the outer loop also runs  $n$  times, the average-case time complexity is  $O(n^2)$ .

## Space Complexity Analysis

Bubble sort is an in-place sorting algorithm. It requires only a constant amount of extra memory (for swapping elements, etc.), irrespective of the size of the input list. Hence, the space complexity is  $O(1)$ .

Things to keep in mind when thinking of a bubble sort:

- **Adaptive Nature:** Bubble sort is adaptive, meaning its efficiency improves if given a partially sorted list. The optimized version (that checks for swaps) will run faster if there are fewer elements out of order.
- **Stable Sort:** Bubble sort is stable, implying that relative ordering of equal elements remains unchanged even after sorting. This can be crucial in scenarios where the order of duplicate elements carries significance.

- **Usage:** Due to its quadratic time complexity, bubble sort is not suitable for large datasets when compared to more efficient algorithms like merge sort or quicksort. However, it can be a suitable choice for small datasets or for teaching purposes due to its simplicity.

Understanding the intricacies of bubble sort gives one a foundational knowledge of sorting algorithms. While not the most efficient, its conceptual simplicity offers a stepping stone to grasp more complex algorithms. When dealing with real-world applications, it's essential to weigh the benefits of simplicity against the needs of efficiency and choose an appropriate sorting algorithm accordingly.

# Insertion Sort

## Insertion Sort: Detailed Implementation

**Insertion Sort** is a simple sorting algorithm. It operates by considering one element at a time, inserting it into its correct position within the already-sorted part of the array. Insertion sorts are really useful to compare it to more advance ones like quicksort, heapsort, or merge sort that we will cover in the next assignment.

Info

### Sorting Mechanics

The following gif demonstrates how an insertion sort works:



6 5 3 1 8 7 2 4

[CC BY-SA 3.0, Link](#)

## Steps

1. **Initialization:** Start from the second element (index 1) assuming the element at index 0 is sorted.
2. **Extraction:** Extract the current element to be compared. This element will be compared with the elements to its left.
3. **Comparison & Insertion:** Compare the current element with the previous elements:
  - If the current element is smaller than the previous element, we continue comparing with the elements before until we reach an element smaller or reach the beginning of the array.
  - Insert the current element in its correct position so that the left part of the array remains sorted.
4. **Iteration:** Repeat the process for each of the elements in the array.

## Java Implementation

Start by creating the `insertionSort` method that takes an array of integers and does not return anything. Iterate over the array starting at index 1.

Create the variable `key` which is assigned `arr[i]` and variable `j` which is assigned to `i - 1`. If we started iterating at index 0 then creating `j` would result in an index out of bounds error.

```
public static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int key = arr[i];
        int j = i - 1;

    }
}
```

Create a while loop that runs as long as `j >= 0` and `arr[j] > key`. As long as this loop is running, assign `arr[j + 1]` the value of `arr[j]`. Remember, `j = i - 1` so `arr[j + 1]` is actually `arr[i]`. However, we are not writing over any data since `arr[i]` has been stored in the variable `key`. Then decrement `j` by 1.

```
public static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements of arr[0..i-1] that are greater
        // than the key
        // to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }

    }
}
```

Finally, after the while loop concludes, assign `arr[j + 1]` the value of `key`. Think of an insertion sort as dividing the array into the left half and the right half. The dividing line between the two halves is the index `i`. The left half is sorted, while the right half is unsorted. Assigning `key` to `arr[j + 1]` means that you are placing `key` into its proper position on the left half of the array.

```
public static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements of arr[0..i-1] that are greater
        // than the key
        // to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

In the `main` method, your code will be provided with the unsorted array:

```
tex -hide-clipboard [12, 11, 13, 5, 6]
```

Your code should produce:

```
tex -hide-clipboard Sorted array: 5 6 11 12 13
```

Insertion Sort can be visualized as the method most people use to arrange playing cards in their hands. While not the most efficient for large datasets, its simplicity offers advantages:

- **Adaptive:** It's efficient for nearly-sorted lists.
- **Stable:** Retains relative order of input data.
- **In-Place:** Requires a constant amount of extra memory.

# Insertion Sort Analysis

After understanding the mechanics and implementation of an insertion sort, the next vital step is to analyze its performance in terms of time and space complexity. This allows us to gauge its efficiency and identify appropriate scenarios for its use.

## Time Complexity Analysis

- **Best Case:** The best-case scenario for an insertion sort is when the input list is already sorted. In this case, the inner loop won't be executed, resulting in a time complexity of  $O(n)$ .
- **Worst Case:** The worst-case scenario is when the input list is sorted in reverse order. For every  $i$ th element, the inner loop does  $i$  operations. Summing this over all  $n$  elements results in a time complexity of  $O(n^2)$ .
- **Average Case:** On average, the inner loop iterates half of the time compared to the worst case, leading to a time complexity of  $O(n^2)$ .

## Space Complexity Analysis

Insertion sort is an in-place sorting algorithm. This means it uses a constant amount of extra space (for variables like `key` and `j` in our implementation). Hence, the space complexity is  $O(1)$ .

## Auxiliary Space Analysis

Recall that auxiliary space is the extra space or temporary space used by an algorithm. For an insertion sort, the auxiliary space is  $O(1)$  since it only uses a constant amount of additional space (the variable `key`).

Things to keep in mind when thinking of insertion sorts:

- **Adaptivity:** Insertion sorts are adaptive. If only a few elements are out of their correct positions (i.e., the list is partially sorted), an insertion sort can be much faster than other sorting methods. For almost sorted data, it becomes linear in time complexity, i.e.,  $O(n)$ .
- **Stability:** Insertion sorts are stable, meaning that the relative order of equal data elements remains unchanged. This property can be crucial in specific applications where data order matters.

While an insertion sort has a quadratic time complexity, making it less efficient for large datasets, its simplicity and efficiency on nearly sorted data make it a good choice in specific scenarios. It works particularly well for small datasets and can outperform more advanced algorithms in such cases. Insertion sorts often used as the base case for higher overhead divide-and-conquer sorting algorithms, like merge sort or quicksort, when the problem size is below a certain threshold.

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- **Identify the underlying mechanics of each of the merge and quick sort algorithms**
- **Implement each sorting method in Java**
- **Analyze the time and space complexities of each algorithm**
- **Identify practical scenarios best suited for each algorithm**

# Advanced Sorting Algorithms and Analysis

While basic sorting algorithms like bubble sort, insertion sort, and selection sort are simple and intuitive, they are not optimized for large datasets. As we venture into larger and more complex data structures, the need for advanced sorting algorithms becomes paramount. In this assignment, we will delve into two fundamental advanced sorting algorithms:

- **Merge Sort:** A classic divide-and-conquer algorithm that splits the array into halves, recursively sorts the halves, and then merges them in a sorted manner.
- **Quick Sort:** Another divide-and-conquer technique but with a twist – it selects a “pivot” element from the array and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

# Merge Sort

## Merge Sort: Overview, Steps, and Java Implementation

**Merge sorts** are a recursive, divide-and-conquer sorting algorithm known for its consistent performance and stable sorting characteristics. The core idea is to continually divide an array into two halves, sort each half, and then merge the sorted halves together.

6 5 3 1 8 7 2 4

[CC BY-SA 3.0, Link](#)

## Steps

1. **Divide:** If the array has one or zero elements, it is considered sorted. Otherwise, divide the array into two halves.
2. **Conquer:** Recursively sort both halves of the array.
3. **Merge:** Combine (merge) the sorted halves to produce a single sorted array. This step is crucial and is where the two halves are merged in sorted order.

## Java Implementation

Create the method `mergeSort` that takes an array of integers, an integer that represents the left index, and an integer that represents the right index. The method should not return anything. This is a recursive algorithm, so we need to set up the base case. As long as `left` is less than `right`, calculate the midpoint, sort the left half, sort the right half, and then merge the two halves together.

```

public static void mergeSort(int[] arr, int left, int right)
{
    if (left < right) {
        int mid = (left + right) / 2;

        // Sort the halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

```

Next, create the `merge` method. It takes an array of integers and three integers representing the left index, the midpoint, and the right index. Calculate the lengths of the left array (`n1`) and the right array (`n2`). Then create temporary arrays for the left half and the right half.

```

private static void merge(int[] arr, int left, int mid, int
right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Temp arrays
    int L[] = new int[n1];
    int R[] = new int[n2];

}

```

Copy data from the left half of the original array (`arr`) into the temporary left array (`L`). Do the same thing for the right half.

```

private static void merge(int[] arr, int left, int mid, int
    right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Temp arrays
    int L[] = new int[n1];
    int R[] = new int[n2];

    // Copy data
    for (int i = 0; i < n1; ++i)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; ++j)
        R[j] = arr[mid + 1 + j];

}

```

Now that the original array has been divided into two temporary arrays, we can merge the elements back into the original array in order. Create the index variables *i*, *j*, and *k*. They will point to the position in the left array (*i*), the right array (*j*), and the original array (*k*). As long as *i* and *j* have both not reached the end of their arrays, ask if the element in the temporary left array is less than the element in the temporary right array. If so, put the element from the temporary left array into the original array. Then increment *i*. Otherwise, put the element from the temporary right array into the original array and increment *j*. Before the loop iterates again, increment *k*.

```

private static void merge(int[] arr, int left, int mid, int
    right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Temp arrays
    int L[] = new int[n1];
    int R[] = new int[n2];

    // Copy data
    for (int i = 0; i < n1; ++i)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; ++j)
        R[j] = arr[mid + 1 + j];

    // Merge
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

}

```

There may be some remaining data in the left and right arrays that have not yet been merged. This could happen if either *i* or *j* reach the end of their respective array before the other has. Copy over any outstanding data from the temporary arrays back to the original array.

```

private static void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Temp arrays
    int L[] = new int[n1];
    int R[] = new int[n2];

    // Copy data
    for (int i = 0; i < n1; ++i)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; ++j)
        R[j] = arr[mid + 1 + j];

    // Merge
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy any remaining elements
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

```

## ▼ Code

Your code should look like this:

```

public static void mergeSort(int[] arr, int left, int right)
{
    if (left < right) {
        int mid = (left + right) / 2;

        // Sort the halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

private static void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Temp arrays
    int L[] = new int[n1];
    int R[] = new int[n2];

    // Copy data
    for (int i = 0; i < n1; ++i)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; ++j)
        R[j] = arr[mid + 1 + j];

    // Merge
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy any remaining elements
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
}

```

```
while (j < n2) {  
    arr[k] = R[j];  
    j++;  
    k++;  
}  
}
```

In the `main` method, your code will be provided with the unsorted array:

```
tex -hide-clipboard [12, 11, 13, 5, 6, 7]
```

Your code should produce:

```
tex -hide-clipboard Sorted Array: 5 6 7 11 12 13
```

Merge sorts stand out due to their predictable and consistent performance. This makes merge sorts a common choice in many applications.  $O(n \log n)$  time complexity in all cases (best, average, and worst). Though it has a space overhead because of the temporary arrays used in the merge step, its stable and consistent performance makes it a reliable choice in many applications.

# Merge Sort Analysis

## Merge Sort: Detailed Analysis

Merge sorts are popular due to their predictable performance. That does not mean, however, that this is the best sorting algorithm to use. There are a few cases where the merge sort may not be the optimal solution. Let's take a look at the sort in detail.

## Time Complexity

- **Best Case:** The best-case scenario for a merge sort is when the array is already sorted or nearly sorted. However, due to the recursive nature of the algorithm and the way it divides and merges the array, the time complexity remains  $O(n \log n)$ .
- **Average Case:** On average, irrespective of the initial order of elements, merge sorts take  $O(n \log n)$  time to sort an array.
- **Worst Case:** Even in the worst scenario, the time complexity of a merge sort is  $O(n \log n)$ , making it one of the more consistent sorting algorithms in terms of performance.

## Space Complexity

Merge sorts use additional space for the temporary left ( $L[]$ ) and right ( $R[]$ ) arrays during the merge step. Thus, its space complexity is  $O(n)$ , making it less space-efficient compared to some in-place sorting algorithms.

## Stability

A merge sort is a **stable** sorting algorithm. This means that the relative order of equal elements remains unchanged after sorting. Stability is essential in scenarios where multiple sorting passes are needed, like when sorting by multiple criteria.

## Advantages

A merge sort guarantees  $O(n \log n)$  performance irrespective of the initial order of the data. Many developers desire the consistency this sort offers. Merge sorts are also stable which is advantageous, especially in database management and other areas where data integrity and order are crucial.

They are also well-suited for external sorting, where data to be sorted does not fit into memory and resides in slower external memory (like disk). It efficiently manages chunks of data, making it a preferred choice in such scenarios.

## Disadvantages

The major drawback is its space complexity of  $O(n)$ . This additional space requirement can be a concern for large arrays. However, that does not mean merge sorts are well-suited for small datasets either. For smaller lists, other algorithms, like insertion sort, might be faster due to the overhead of recursive function calls and memory allocations in a merge sort.

Merge sorts offer a blend of stable and consistent performance, making it a standard choice for many sorting tasks. While its predictable time complexity is a strength, potential users should be wary of its additional space requirements, especially in memory-constrained environments. When choosing a sorting algorithm, it's vital to consider the dataset's size, the available memory, and the importance of stability in sorting.

---

# Quick Sort

## Quick Sort: Overview, Steps, and Java Implementation

**Quick sort** is a divide-and-conquer sorting algorithm that operates by selecting a “pivot” element from the array and partitioning the other elements into two sub-arrays, based on whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

Info

### Sorting Mechanics

The following gif demonstrates how a quick sort works:



6 5 3 1 8 7 2 4

[CC BY-SA 3.0, Link](#)

## Steps

1. **Select Pivot:** Choose a pivot element from the array. The choice of pivot can be the first element, the last element, the middle one, or any random element.
2. **Partition:** Reorder the array by comparing with the pivot. All elements smaller than the pivot come before the pivot and all greater elements come after. After this step, the pivot is in its sorted position.
3. **Recursive Sort:** Recursively apply the above steps to the two sub-arrays (elements less than and greater than the pivot).

## Java Implementation

Create the recursive method `quickSort`. It takes an array of integers, as well as two integers representing the low and high indices. The method should not return anything. As long as `low` is less than `high`, find the partition index by calling the `partition` method. Then call `quickSort` on the left half and then again on the right half.

```
public static void quickSort(int arr[], int low, int high) {  
    if (low < high) {  
        int pivotIndex = partition(arr, low, high);  
        quickSort(arr, low, pivotIndex - 1);  
        quickSort(arr, pivotIndex + 1, high);  
    }  
}
```

Next, we need to define the `partition` method. It takes an array of integers as well as two integers representing the low and high indices. The method should return an integer. Set the pivot to the last element in the array. Create the variable `i` and set its value to `low - 1`. Create a for loop that has the loop variable `j` that starts at `low` and iterates as long as `j` is less than `high`.

```
private static int partition(int arr[], int low, int high) {  
    int pivot = arr[high];  
    int i = low - 1;  
    for (int j = low; j < high; j++) {  
  
    }  
}
```

Before we continue with the `partition` method, we are going to create the `swap` method that swaps two values in an array. We will be swapping a few different times, so it makes sense to put this logic into its own method. The method takes an array of integers as well as two other integers (the values to be swapped). It does not return anything.

```
private static void swap(int[] arr, int i, int j) {  
    int temp = arr[i];  
    arr[i] = arr[j];  
    arr[j] = temp;  
}
```

Return to the `partition` method. Inside the for loop, ask if the element at index `j` is less than the pivot. If so, increment `i` and swap the values `arr[i]` and `arr[j]` with the newly created `swap` method.

```
private static int partition(int arr[], int low, int high) {  
    int pivot = arr[high];  
    int i = low - 1;  
    for (int j = low; j < high; j++) {  
        if (arr[j] < pivot) {  
            i++;  
            swap(arr, i, j);  
        }  
    }  
}
```

After the for loop finishes running, swap the values at `arr[i + 1]` and `arr[high]`. Then return `i + 1`.

```
private static int partition(int arr[], int low, int high) {  
    int pivot = arr[high];  
    int i = low - 1;  
    for (int j = low; j < high; j++) {  
        if (arr[j] < pivot) {  
            i++;  
            swap(arr, i, j);  
        }  
    }  
    swap(arr, i + 1, high);  
    return i + 1;  
}
```

## ▼ Code

Your code should look like this:

```

public static void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

private static int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, high);
    return i + 1;
}

private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

```

In the `main` method, your code will be provided with the unsorted array:

`tex -hide-clipboard [10, 7, 8, 9, 1, 5]`

Your code should produce:

`tex -hide-clipboard Sorted array: 1 5 7 8 9 10`

As its name implies, the quick sort is a faster sort. In practice, it can even outperform other sorting algorithms with the same time complexity. Developers have developed common strategies like using a random pivot or the “median-of-three” approach to help compensate for some of the algorithm’s shortcomings.

# The Pivot

## The Importance of the Pivot

The choice of the pivot element is crucial as it directly influences the efficiency and performance of the sorting process. The pivot element is used to partition the array into two subarrays: one containing elements less than or equal to the pivot, and the other containing elements greater than the pivot. The partitioning step is a fundamental operation in the quick sort algorithm, and the choice of pivot significantly affects the algorithm's behavior. Choosing a bad pivot leads to imbalanced partitioning. For example, if you consistently choose a pivot that is either the smallest or largest element in the array, the partitioning process will result in highly imbalanced subarrays. This means one subarray will have significantly more elements than the other, which leads to inefficient sorting. Here are some common techniques used to select a pivot for a quick sort.

## Random Pivot Selection

In this strategy, we randomly select a pivot element from the array. This helps reduce the chances of consistently encountering a “bad” pivot. Start by importing the `Random` library. This should come at the beginning of the program.

```
import java.util.Random;
```

Then update the `QuickSort` class so that it instantiates a `Random` object. This should happen before the `quickSort` method.

```
public class QuickSort {
    private static final Random rand = new Random();

    // code below remains unchanged
```

Next, update the `partition` method. Generate a random number between the indices `low` and `high`. Then swap the pivot value to the end of the subarray. It will be swapped back to its correct position before returning `arr[i + 1]`.

```

private static int partition(int arr[], int low, int high) {
    int pivotIndex = rand.nextInt(high - low + 1) + low;
    int pivotValue = arr[pivotIndex];

    swap(arr, pivotIndex, high);
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivotValue) {
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, high);
    return i + 1;
}

```

## First Element as Pivot

In this strategy, we always select the first element of the subarray as the pivot. Update the partition method to use the first element in the subarray as the pivot value. Notice that `i` is set to `low` and the for loop sets `j` to `low + 1`. We are also swapping `arr[i]` and `arr[low]` after the for loop runs.

```

private static int partition(int[] arr, int low, int high) {
    int pivotValue = arr[low];
    int i = low;
    for (int j = low + 1; j <= high; j++) {
        if (arr[j] < pivotValue) {
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i, low);
    return i;
}

```

## Median-of-Three Pivot Selection

In this strategy, we choose the median of the first, middle, and last elements of the subarray as the pivot. Update the `quickSort` method so that it creates the `median` variable and sets its value as the result of the `medianOfThree` method. Then pass `median` to the `partition` method as a fourth argument.

```

public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int median = medianOfThree(arr, low, high);
        int pivotIndex = partition(arr, low, high, median);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

```

Next, update the `partition` method so that it has a fourth parameter, an integer representing the pivot index. After setting `pivotValue` to the element at `pivotIndex`, swap `arr[pivotIndex]` and `arr[high]`. The variable `i` is now `low - 1`, and the loop variable `j` is initialized to `low`. Finally, the swap after the for loop switches `arr[i + 1]` and `arr[high]`.

```

private static int partition(int[] arr, int low, int high,
    int pivotIndex) {
    int pivotValue = arr[pivotIndex];
    swap(arr, pivotIndex, high);
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivotValue) {
            i++;
            swap(arr, i, j);
        }
    }

    swap(arr, i + 1, high);
    return i + 1;
}

```

Create the `medianOfThree` method that takes an array of integers, and integer representing the first element in a subarray, and another integer representing the last element in a subarray. The method should return an integer. This method calculates the midpoint of the subarray and returns the middle element if you order the elements for `arr[low]`, `arr[mid]`, and `arr[high]`.

```
private static int medianOfThree(int[] arr, int low, int
    high) {
    int mid = (low + high) / 2;

    if (arr[low] > arr[mid]) {
        swap(arr, low, mid);
    }
    if (arr[low] > arr[high]) {
        swap(arr, low, high);
    }
    if (arr[mid] > arr[high]) {
        swap(arr, mid, high);
    }

    // At this point, arr[low] <= arr[mid] <= arr[high]
    // The median is now at arr[mid]

    return mid;
}
```

# Quick Sort Analysis

## Quick Sort: Detailed Analysis

The quick sort offers some advantages over other sorting algorithms like the merge sort. However, as we have seen so far, algorithm design choices are nothing but a series of tradeoffs. Let's take a look at the quick sort and see how it compares to other sorting algorithms.

## Time Complexity

- **Best Case:** If the pivot splits the array into roughly equal halves at every step, then the algorithm performs at its best, leading to a time complexity of  $O(n \log n)$ . This is because we're dividing the array into halves at each step (which gives the  $(\log n)$  factor) and then taking linear time to process each level.
- **Average Case:** On average, a quick sort runs in  $O(n \log n)$  time, assuming that the pivots chosen divide the array into reasonably balanced partitions.
- **Worst Case:** The worst-case scenario for a quick sort is when the smallest or largest element is always chosen as the pivot (like a sorted or reverse sorted array). This results in a time complexity of  $O(n^2)$  since we get  $(n)$  partitions of size  $(n - 1), (n - 2)$ , etc.

## Space Complexity

Quick sorts are an in-place sorting algorithm, meaning it sorts the array using a constant amount of extra space. The recursive stack might require  $(\log n)$  layers in the best case, leading to a space complexity of  $O(\log n)$ . In the worst case, if each recursive call processes a list of size one less than the previous list, it results in  $O(n)$  space complexity.

## Stability

Quick sorts are **not stable** by default. Stability in sorting algorithms means that when two elements have equal keys, the original order is preserved in the sorted output. However, it can be made stable with appropriate modifications.

## Advantages

Quick sorts are an in-place sort (it does not require extra storage space), making it memory efficient. They often have better cache performance than other sorting algorithms, like a merge sort, because they access array elements in a sequential manner. Quick sorts also have a tail-recursive structure, which for some architectures means a smaller call stack overhead.

## Disadvantages

If not implemented correctly, quick sorts can have a time complexity of  $O(n^2)$  in its worst-case scenario, which can be a severe inefficiency for large datasets. This algorithm is not stable, which means it is not a good choice for when you want to maintain the order of equal values. Finally, the choice in pivot that you make can have an effect on efficiency. A bad pivot choice can lead to reduced performance.

Quick Sorts are often the sorting algorithm of choice in practice for in-memory sorting due to its average-case performance and memory efficiency. However, care should be taken in scenarios where worst-case performance might be unacceptable. To avoid worst-case performance, many practical implementations use a hybrid approach that switches to another algorithm like insertion sort for small sub-arrays.

---

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- **Define Search Algorithm**
- **Understand the algorithm behind Linear Search, Binary Search, and Jump Search**
- **Implement Common search algorithm**
- **Analyze the complexities of Common Search Algorithms**

# Searching

## Linear Search, Binary Search, and Jump Search

In the field of computer science, searching algorithms play a pivotal role in locating specified items among a collection of items. Efficient and effective search algorithms are at the core of many operations — be it in databases, data retrieval, or simply within software applications. We mentioned some of these search algorithm before, but here we are giving them individualized attention. This assignment aims to delve into three common searching algorithms: **linear search**, **binary search**, and **jump search**.

Each of these algorithms has its unique characteristics, advantages, and disadvantages. Throughout this assignment, we will explore how each algorithm works, analyze their time and space complexities, and understand their best and worst-case scenarios.

## Goals of the Assignment

1. To understand the fundamental principles behind each searching algorithm.
2. To implement linear search, binary search, and jump search in Java.
3. To analyze the time and space complexities for each algorithm.
4. To compare and contrast the efficiency and practicality of each algorithm in various scenarios.
5. To draw conclusions regarding which algorithm is best suited for particular types of problems.

By the end of this assignment, you will have a comprehensive understanding of how each of these algorithms functions and where they are best applied. This will equip you with the skills necessary to make informed decisions when faced with problems that require searching operations.

# Linear Search

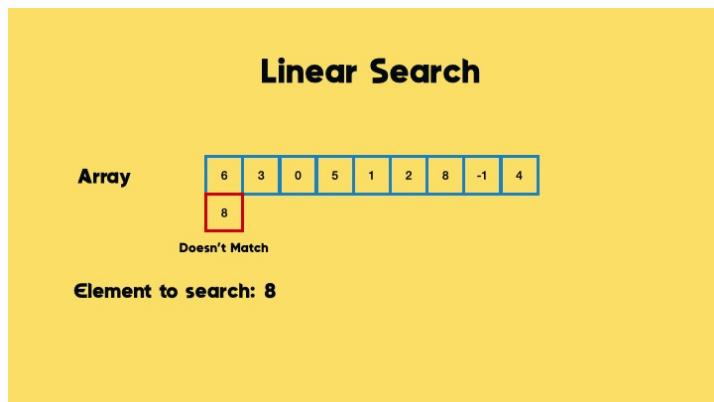
## Linear Search: Overview, Steps, and Implementation

**Linear search** is the simplest form of searching algorithm. It is often the first searching algorithm taught in computer science courses due to its straightforward logic. The algorithm scans through each element of the collection sequentially, stopping when it finds the element that matches the target. While not the most efficient for large datasets, it's simple to understand and implement.

info

### Searching Mechanics

The following gif demonstrates how a linear search works:



### Steps

1. **Start at the Beginning:** Begin from the first element of the array.
2. **Sequential Scan:** Move from one element to the next in sequence,

- comparing each with the target value.
3. **Check for Match:** If an element matches the target, return its index.
  4. **End of Array:** If the end of the array is reached without finding the target, return -1 to indicate that the item was not found.

## Implementation

We have seen linear search several times now, so the implementation of its algorithm should not come as a surprise. Create the `linearSearch` method. It takes an array of integers and another integer representing the search target. The method should return an integer.

```
public static int linearSearch(int[] arr, int target) {  
    }  
}
```

Starting at the first element, iterate over the array. Check each element in the array against the search target. If they match, return the index of the matching element. If the loop finishes without a match, that means the search target is not in the array. Return -1.

```
public static int linearSearch(int[] arr, int target) {  
    for (int i = 0; i < arr.length; i++) {  
        if (arr[i] == target) {  
            return i; // Target found, return index  
        }  
    }  
    return -1; // Target not found  
}
```

The code in the IDE is sending the information below to the `linearSearch` method. Your code should return 6.

```
int[] arr = {3, 9, 6, 4, 7, 1, 5, 8, 2};  
int target = 5;
```

Linear search is an excellent algorithm for smaller datasets or for unsorted arrays where more advanced algorithms cannot be applied. It is a brute-force method, checking each element one-by-one, which makes it less suitable for larger, sorted datasets where other methods like Binary Search or Jump Search may offer better performance.

# Linear Search Analysis

## Time Complexity

- **Worst-Case:**  $O(n)$  - In the worst-case scenario, the target element could be at the end of the array, necessitating a scan through every element.
- **Average-Case:**  $O(n)$  - On average, we expect to search through half of the elements, but the time complexity still scales linearly with the size of the data set.
- **Best-Case:**  $O(1)$  - In the best-case scenario, the target element is the first one in the array, and the search concludes immediately.

## Space Complexity

The space complexity of linear search is  $O(1)$ , meaning it requires a constant amount of additional memory to perform the search. This is because the algorithm only needs to store a single element (the target) for comparison, regardless of the size of the array.

## Practical Use-Cases

Linear search is practical when dealing with small arrays or lists, where its simplicity outweighs the speed benefits of more complex algorithms. Unlike algorithms such as binary search, linear search can be applied to unsorted datasets. In cases where the data is streaming or not all available at once, linear search can be useful as it processes elements sequentially.

## Comparison with Other Algorithms

1. **Ease of Implementation:** Linear search is easier to implement compared to other algorithms like binary or jump search.
2. **Universality:** It works on sorted and unsorted data, unlike binary and jump search which require sorted arrays.
3. **Efficiency:** It is less efficient than other search algorithms like binary and jump search for larger datasets.

Linear search, though straightforward, is often inefficient for larger datasets, especially when compared to more advanced algorithms like binary search or jump search. However, its simplicity and universality

make it a suitable choice for small or unsorted datasets. It also has the advantage of not requiring any additional memory, making it a “greedy” algorithm in terms of space.

Understanding the strengths and weaknesses of linear search is crucial for selecting the appropriate searching algorithm based on the problem requirements. In subsequent sections, we will delve deeper into more advanced searching algorithms to compare their efficiencies and practical applications.

# Binary Search

## Binary Search: Overview, Steps, and Implementation

**Binary search** is a highly efficient searching algorithm that works by repeatedly dividing the sorted array into halves until the target element is found. By taking advantage of the sorted nature of the array, binary search minimizes the number of comparisons needed, offering a substantial performance gain over algorithms like linear search.

info

### Searching Mechanics

The following gif demonstrates how a binary search works:

[CC BY-SA 4.0, Link](#)

## Steps

1. **Initialize Pointers:** Set the low pointer to the first index and the high pointer to the last index of the array.
2. **Calculate Mid:** Calculate the middle index as  $(\{low\} + \{high\}) / 2$ .
3. **Check for Match:** Compare the middle element with the target. If it matches, return the middle index.
4. **Adjust Pointers:**
  - If the target is less than the middle element, set high to  $\{mid\} - 1$

- ).
- If the target is greater than the middle element, set low to (`{mid} + 1` ).

5. **Iterate:** Repeat steps 2-4 until the low pointer is less than or equal to the high pointer. If not found, return -1.

## Implementation

Binary search can be implemented in a variety of ways. We are going to use an iterative approach. Start by creating the `binarySearch` method. It should take an array of integers and an integer representing the search target. The method should return an integer. Create the variables `low` and `high`, and set them to the first and last elements respectively.

```
public static int binarySearch(int[] arr, int target) {
    int low = 0, high = arr.length - 1;

}
```

As long as `low` is less than or equal to `high`, find the midpoint between the two pointers. If the element at the midpoint is equal to the target, return the midpoint. If the element at the midpoint is less than the target, set `low` to the midpoint plus 1. Otherwise set `high` to the midpoint minus 1. If the loop completes without returning a value, then the search target is not in the array. Return -1.

```
public static int binarySearch(int[] arr, int target) {
    int low = 0, high = arr.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;

        if (arr[mid] == target) {
            return mid; // Target found, return index
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1; // Target not found
}
```

The code in the IDE is sending the information below to the `binarySearch` method. Your code should return 5.

```
int[] arr = {1, 3, 5, 7, 9, 11, 13, 15};  
int target = 11;
```

Binary search offers a significant performance improvement over Linear Search for sorted arrays, particularly as the size of the dataset increases. However, it requires the array to be sorted beforehand, which could be a limitation depending on the application.

# Binary Search Analysis

## Time Complexity

- **Worst-Case:**  $O(\log n)$  - Even in the worst-case scenario, binary search minimizes the number of comparisons through successive divisions of the array.
- **Average-Case:**  $O(\log n)$  - The average case also benefits from the logarithmic nature of the algorithm.
- **Best-Case:**  $O(1)$  - In the best-case scenario, the target element is right at the center of the array, allowing the algorithm to find it in just one comparison.

## Space Complexity

The space complexity for Binary Search is  $O(1)$ , meaning it requires a constant amount of additional memory for its operation. The algorithm only requires variables to store the high, low, and mid indices, which does not scale with the size of the input array.

### ▼ Recursive Space Complexity

You can also have a recursive solution to binary search. In this case, the space complexity would not be constant. Instead it would be  $O(\log n)$ .

## Practical Use-Cases

Binary search is ideal for search problems involving large datasets, provided that the data is sorted. It is frequently used in applications that require fast data retrieval times, such as databases. Binary search also provides version control, meaning you can quickly find specific versions in a sorted version history.

## Comparison with Other Algorithms

- **Efficiency:** Binary search is significantly more efficient than linear search for large datasets, but less efficient than specialized algorithms like Fibonacci search or interpolation search for uniformly distributed datasets.
- **Preconditions:** Unlike linear search, binary search requires the input array to be sorted.
- **Simplicity:** It is slightly more complicated to understand and

implement than linear search, but simpler than many other advanced searching algorithms.

Binary Search offers an excellent balance of efficiency and simplicity. Its logarithmic time complexity makes it a preferred choice for large, sorted datasets. However, the prerequisite of a sorted array can sometimes be a limitation.

In terms of efficiency, Binary Search performs exceptionally well, but it is crucial to remember its limitation that it can only operate on sorted arrays. Its performance and operational characteristics make it a popular choice in many applications that require efficient search capabilities.

# Jump Search

## Jump Search: Overview, Steps, and Implementation

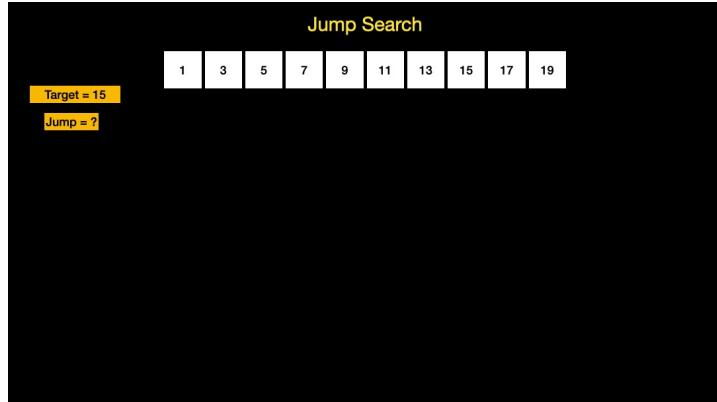
### Overview:

**Jump Search** is an intermediate searching algorithm that falls between linear and binary search in terms of efficiency. It's designed to find a target element within a sorted array by skipping a predefined number of steps. If Jump Search goes too far (i.e., the current element is greater than the target), it takes a step back and performs a linear search to locate the target. This strategy capitalizes on a balance between skipping a large number of elements and refining the search space.

info

### Searching Mechanics

The following gif demonstrates how a jump search works:



The image shows the steps of a jump search. The array has the elements 1, 3, 5, 7, 9, 11, 13, 15, 17 and 19. 15 is the search target, and the jump size is 3. Arrows show the jumps through the array. When the third jump points to an element with a value greater than 15, the algorithm starts the linear portion. It does a linear search for the elements 13, 15, 17, and 19. When the linear search gets to 15, the gif repeats.

## Steps

1. **Determine Jump Size:** Typically, the jump size is calculated as  $\sqrt{n}$ , where  $n$  is the length of the array.
2. **Initialize Pointers:** Set the initial position to 0.
3. **Jump Ahead:** Jump forward by the step size.
4. **Check Position:**
  - If the element at the current position is equal to the target, return the position.
  - If the element at the current position is greater than the target, move to the linear search phase.
  - If the element at the current position is less than the target, repeat the jump.
5. **Linear Search Phase:** If you have jumped too far (current element > target), backtrack to the previous jump and search linearly up to the current jump position.
6. **Iterate:** Continue this process until the target is found or you have searched the entire array. If you did not find the target, return -1.

Since this is our first time encountering this specific search algorithm, let's walk through a jump search together. Assume we are starting with the following information.

```
tex -hide-clipboard Array: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
Target: 15
```

First, find the step size by calculating  $\sqrt{10}$ , since the length of the array is 10. The jumps are going to be an index, so we need to remove any decimals from the step size. In this example, we will be jumping by 3 elements.

1. **First Jump:** Start at index 0. Jump 3 steps to index 3,  $\text{Array}[3] = 7$  and  $7 < 15$ .
2. **Second Jump:** Jump another 3 steps to index 6,  $\text{Array}[6] = 13$  and  $13 < 15$ .
3. **Third Jump:** Jump another 3 steps to index 9,  $\text{Array}[9] = 19$  and  $9 > 15$ .
4. **Linear Search Phase:** At this point, we've jumped too far. We backtrack to index 6 and perform a linear search between index 6 and index 9.
  - **Linear Search 1:** Check index 6,  $\text{Array}[6] = 13$  and  $13 < 15$ .
  - **Linear Search 2:** Check index 7,  $\text{Array}[7] = 15$  and  $15 = 15$ .
5. **Found:** We find that the element at index 7 is the target value. The search stops and returns the index 7.

## Implementation

Before we begin programming, do note that a jump search commonly employs using a square root. This means you must import the `java.lang.Math` library. Start by creating the `jumpSearch` method. It takes an array of integers, and integer that represents the search target, and the

method returns an integer. Calculate the length of the array, and use the length to calculate the step size for each jump. Since taking the square root of a number can return a decimal, use the `Math.floor` method to truncate any decimals. We also need the variable `prev` which we will use to represent the index prior to the jump.

```
public static int jumpSearch(int[] arr, int target) {  
    int n = arr.length;  
    int step = (int) Math.floor(Math.sqrt(n));  
    int prev = 0;  
  
}
```

We need to be careful to not “jump out of” the array. We are going to do this with a while loop. By saying `arr[Math.min(step, n) - 1]`, we are telling Java to use the most recent jump as the index, unless it were to exceed the length of the array. In that case, use the last element in the array. This ensures that we do not have an index out of bound error.

Keep iterating as long as the index at the `step` value is less than the target. When this happens, set `prev` to `step` and increase the value of `step`. If `prev` is greater than the size of the array, that means the search target is not in the array. Return -1.

```
public static int jumpSearch(int[] arr, int target) {  
    int n = arr.length;  
    int step = (int) Math.floor(Math.sqrt(n));  
    int prev = 0;  
  
    while (arr[Math.min(step, n) - 1] < target) {  
        prev = step;  
        step += (int) Math.floor(Math.sqrt(n));  
        if (prev >= n) {  
            return -1; // Target not found  
        }  
    }  
}
```

If the value of `arr[step]` and `arr[n]` is greater than the search target, then we are going to switch to the linear search portion of the algorithm. Start the for loop at the value of `prev`. Continue iterating as long as the loop variable is less than the smallest value between `step` and `n`. If the element at the index of the loop variable is the search target, return the loop variable. If the for loop finishes without returning a value, that means the search target is not in the array. Return -1.

```

public static int jumpSearch(int[] arr, int target) {
    int n = arr.length;
    int step = (int) Math.floor(Math.sqrt(n));
    int prev = 0;

    while (arr[Math.min(step, n) - 1] < target) {
        prev = step;
        step += (int) Math.floor(Math.sqrt(n));
        if (prev >= n) {
            return -1; // Target not found
        }
    }

    // Linear search phase
    for (int i = prev; i < Math.min(step, n); i++) {
        if (arr[i] == target) {
            return i; // Target found, return index
        }
    }
    return -1; // Target not found after linear search
}

```

The code in the IDE is sending the information below to the `jumpSearch` method. Your code should return 3.

```

int[] arr = {1, 3, 5, 7, 9, 11, 13, 15};
int target = 7;

```

Jump Search strikes a balance between skipping large portions of the data and ensuring no data is missed. By taking ‘jumps’ and then refining the search with linear methods, it offers a balance of speed and precision. However, similar to Binary Search, it requires a sorted array.

# Jump Search Analysis

## Time Complexity

- **Best-Case:** The best-case scenario occurs when the target element is the first element of the array. In this case, the time complexity is  $O(1)$ .
- **Worst-Case:** The worst-case scenario occurs when the target is either the last element in the array or not present at all. In this case, you'll have to do approximately  $(\sqrt{n})$  jumps, and then up to  $(\sqrt{n})$  linear searches, giving a worst-case time complexity of  $O(\sqrt{n})$ .
- **Average-Case:** On average, you can expect the algorithm to perform in  $O(\sqrt{n})$  time.

## Space Complexity

The space complexity for Jump Search is  $O(1)$ , as it only uses a few extra variables and does not require any additional data structures like stacks or queues.

## Practical Use-Cases

Jump searches work well with large arrays, database indexing, and approximate searching. Approximate searching is when an exact match is not necessary. It is also a good choice for when you want a search that is faster than linear search, but has an algorithm that is a bit easier to understand than some more advanced searching solutions.

## Comparison with Other Algorithms

- **Balance:** Jump search sit between linear and binary search in terms of performance and complexity.
- **Preconditions:** Like binary search, a jump search requires the input array to be sorted.
- **Distribution:** Jump searches perform better when elements are uniformly distributed, as each ‘jump’ can eliminate a consistent number of elements from consideration.
- **Jumping Too Far:** Switching to linear search after jumping too far ensures that the algorithm will not miss the target element. However, it does mean that the actual search range is not always cut in half or reduced by  $(\sqrt{n})$ , which contributes to its worst-case time complexity.

Jump Search offers a balanced compromise between linear and binary search in both implementation complexity and performance. It's particularly useful when you have a read-only memory, as it does not require the array to be divided like in Binary Search. Jump searches allow you to find a balance between speed and simplicity.

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- **Define Abstract Data Types**
  - **Compare abstract data types and Data structures**
  - **Implement List ADT using an ArrayList**
  - **Apply different operations to list adt.**
-

# Abstract Data Types

## Understanding ADTs

In Computer Science, an **abstract data type (ADT)** is a theoretical concept of how to represent data. Most importantly, ADTs focus on the behavior of the data type. They are not concerned with how the data type is implemented. This allows you to abstract away the implementation of the data type. Instead, you can talk about how to use a graph, tree, or list. It does not matter what programming language you are using or how you implement the ADT.

As its name implies, ADTs offer a level of abstraction when coding. They encapsulate the internal structure of the data, exposing only those operations that are necessary. Because of this, the coder can focus on the interface (what the ADT can do) as opposed to the implementation (how it is done).

Think back to when you first learned about strings. The reference material introduced them as a sequence of characters. You also learned about *what* you could and could not do with strings.

```
String str = "Far out in the uncharted backwaters of the
unfashionable end of the western spiral arm of the
Galaxy lies a small, unregarded yellow sun.";

System.out.println(str.contains("ded y"));
String modifiedStr = str.replaceAll("a", "@");
System.out.println(modifiedStr);
```

However, you did not learn about *how* all of this was done. Do strings in Java use an array, an ArrayList, or a linked list to represent the sequence of characters? In the code example above, which algorithm did Java use to find the substring or replace all instances of the a character? From the point of view of the user, it does not matter.

What matters is the interface – those actions you can perform with strings. If, in a future version of Java, the implementation of the String class changes, the interface will not. You can still instantiate string variables and call string methods just as you always have.

## ADT Importance

ADTs are crucial for several reasons:

- **Modularity:** They allow for greater modularity, as you can change the implementation without affecting the rest of the program.
- **Maintainability:** Code is easier to understand and maintain when you separate the interface from the implementation.
- **Reusability:** Because ADTs are not tied to specific implementations, they can be reused in different contexts.

## ADT vs Data Structure

The table below summarizes the contrast between ADTs and data structures based on features like focus, implementation, user perspective, and advantages.

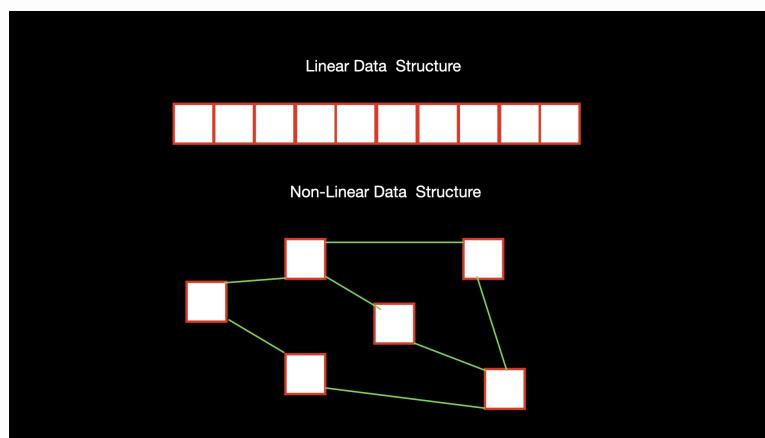
| Feature          | Abstract Data Type<br>(ADT)                       | Data Structure                       |
|------------------|---------------------------------------------------|--------------------------------------|
| Focus            | Behavior of the data                              | Concrete representation of the data  |
| Implementation   | Not specified                                     | Specified                            |
| User perspective | What operations can be performed on the data      | How the data is stored and organized |
| Advantage        | More modular, easier to understand, more portable | Better performance                   |

ADTs focus on the bigger picture (the what), while data structures focus on the details (the how). While ADTs offer several advantages over a concrete data structure, they often come at the cost of performance.

# Lists

## Linear Data Structures

**Linear data structures** are the most straightforward form of data structures, where elements are arranged sequentially. Each element has a unique predecessor and successor except for the first and last elements, respectively.



The image shows two data structures. On top is a linear data structure. On the bottom is a nonlinear data structure.

In Java, the most common examples of a linear data structure are arrays and ArrayLists. There may be distinct differences between the two structures, but they are similar at a fundamental level. Each structure represents a unified block of data. The elements in the block are ordered, which allows us to traverse the structure and access data stored in each element.

Arrays and ArrayLists are not the only data structures that share the characteristics mentioned above. Linked lists are another variation of a linear data structure. They too store data in an ordered manner, they can be traversed, and their data can be accessed. Instead of being a unified block of data, linked lists are made up of separate nodes that have links connecting them. There are even variations within the linked list family. With so many different variations of a linear data structure, how do you talk about these similar yet different structures?

## List Abstract Data Type

Computer scientists use the term **list abstract data Type (ADT)** to refer to this particular group of data structures. We do not have to worry about the data structure being an ArrayList or a linked list, because the List ADT has a set of shared principles:

- **Order Preservation:** The list maintains the order of elements as they were inserted.
- **Sequential Access:** Elements in a list are accessed one after another, either from the beginning to the end or vice versa.
- **Variable Size:** Unlike arrays (in some languages), the size of a list can be dynamically changed, allowing for more flexible storage options.

The list ADT also has a shared set of basic operations:

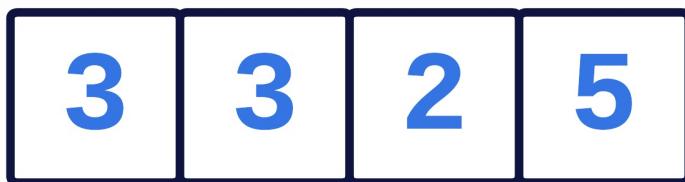
- **Insertion:** Adding a new element at a given position.
- **Deletion:** Removing an element from a specific position.
- **Traversal:** Moving through each element in the list.
- **Searching:** Finding the index or position of an element.

Lists offer a dynamic and flexible way to store data. They are excellent for scenarios where you need quick insertions and deletions, and where the size of the data set might change. Lists are a cornerstone in computer science and programming, serving as the foundational data structure for more complex types like stacks, queues, and even some graph representations.

# Implementing the List ADT

Because the list ADT can be represented by several different concrete data structures, it is important to understand the consequences for selecting a particular implementation. Each data structure has its own advantages and disadvantages. The most common implementations are arrays and linked lists.

## Array-Based Implementation



The image shows an array of four elements. They are “3”, “3”, “2”, and “5”.

An array-based list uses an array to store its elements. The array may have a fixed size or can be dynamically resized.

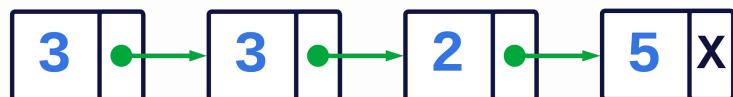
### Strengths:

- \* Fast random access  $O(1)$  to elements.
- \* Less memory overhead compared to linked lists.

### Weaknesses:

- \* Insertions and deletions are generally slower  $O(n)$  since elements may need to be shifted.
- \* Fixed-size arrays limit the list’s size, while dynamic arrays incur occasional resizing costs.

## Linked List-Based Implementation



The image shows a linked list with the elements “3”, “3”, “2”, and “5”. The first three nodes have a pointer pointing to the next node in the list.

A linked list-based list uses nodes, each containing a data element and a reference to the next node.

**Strengths:**

- \* Insertions and deletions are faster  $O(1)$  if a reference to the insertion/deletion position is already available.
- \* No wasted space as the list grows or shrinks dynamically with the number of elements.

**Weaknesses:**

- \* Slower random access  $O(n)$  to elements.
- \* Each node has additional memory overhead for storing the ‘next’ reference.

## Choosing the Right Implementation

Using the information above, we can select the best implementation for some common scenarios:

- **Fixed vs. Dynamic:** If you know the maximum size of your list in advance, an array might be more efficient. For dynamic sizes, a linked list is generally better.
- **Access Patterns:** If your primary use-case is random access, go for array-based lists. If you are frequently inserting and deleting elements, consider linked lists.
- **Memory Concerns:** If memory is a constraint, the less overhead of an array-based list may be beneficial.

Some advanced implementations combine features of both arrays and linked lists to offer a balanced set of advantages. For example, a dynamic array of linked lists or a linked list of sub-arrays.

Both array-based and linked list-based implementations have their pros and cons. The best choice depends on the specific requirements of your application. Understanding these trade-offs can help you make informed decisions and use the list ADT more effectively.

# Arrays

Arrays are fundamental data structures in Java, offering fast access to elements through an index. They have been the dominate form of data structure that we have used so far. One of the defining characteristics of an array is its fixed size.

```
int[] integerArray = new int[5];
String[] stringArray = {"John", "Jane", "Doe"};
```

This allows for predictable memory usage and fast speeds when accessing a stored value. However, arrays have some limitations. We are going to explore why we will not be using an array when implementing the List ADT.

## Limitations

**Arrays** in Java are homogeneous data structures, meaning they store elements of a single data type, whether it's primitive or reference types. Arrays have been the dominant form of data structures that we have seen up until this point.

### Declaration and Initialization:

```
public static void main(String[] args) {
    String[] pets = {"dog", "cat", "fish", "bird"};
    System.out.println(pets);
}
```

Instead, you see something like this:

```
tex -hide-clipboard [Ljava.lang.String;@1dbd16a6
```

The [Ljava.lang.String; indicates that it is an array of strings, while @1dbd16a6 represents the memory reference for the data structure.

You can print the contents of an array, but you must first import the `Arrays` library (already done). Use the `toString` method, and Java will print a string representation of the array and its contents.

```
public static void main(String[] args) {
    String[] pets = {"dog", "cat", "fish", "bird"};
    System.out.println(Arrays.toString(pets));
}
```

Bigger issues revolve around the fixed size of an array. Let's assume we want to add "hamster" to the array `pets`. There are no built-in methods to do this for us; everything must be done manually. Create the method `add` that takes an array of strings and a string representing the new value to add to the array. The method should return an array of strings.

```
public static String[] add(String[] arr, String val) {
    int newSize = arr.length + 1;
    String[] newArr = new String[newSize];

    for (int i = 0; i < arr.length; i++) {
        newArr[i] = arr[i];
    }
    newArr[newSize - 1] = val;
    return newArr;
}

public static void main(String[] args) {
    String[] pets = {"dog", "cat", "fish", "bird"};
    String[] newArray = add(pets, "hamster");
    System.out.println(Arrays.toString(newArray));
}
```

Similarly, if we want to remove an element from the array, we need to write the logic ourselves. The `remove` method takes an array of strings and an integer representing the index of the element we want to remove.

```
public static String[] remove(String[] arr, int index) {
    int newSize = arr.length - 1;
    String[] newArr = new String[newSize];

    for (int i = 0, j = 0; i < arr.length; i++) {
        if (i != index) {
            newArr[j] = arr[i];
            j++;
        }
    }

    return newArr;
}

public static void main(String[] args) {
    String[] pets = {"dog", "cat", "fish", "bird"};
    String[] newArray = remove(pets, 2);
    System.out.println(Arrays.toString(newArray));
}
```

Adding and removing elements from a list is a common task. With arrays (in Java at least), basic operations feel a bit cumbersome. As such, we will not be using them when implementing the List ADT.

# ArrayLists

## ArrayList in Java

The `ArrayList` class is a part of the Java Collection Framework, and is present in `java.util` package. It provides dynamic arrays, and generally makes working with lists more enjoyable.

Let's recreate the operations from the previous page. Use the `add` method to add an element to a list. You can even specify the index at which you like to add the element to the list.

```
public static void main(String[] args) {  
    List<String> pets = new ArrayList<>(Arrays.asList("dog",  
        "cat", "fish", "bird"));  
    pets.add("hamster");  
    pets.add(0, "rabbit");  
    System.out.println(pets);  
}
```

Removing elements is just as easy. Note, the `remove` method only removes the first instance of the value.

```
public static void main(String[] args) {  
    List<String> pets = new ArrayList<>(Arrays.asList("dog",  
        "cat", "fish", "bird"));  
    pets.remove("bird");  
    System.out.println(pets);  
}
```

We tend to think of `ArrayLists` as a dynamic array, but that does not mean the syntax that works with arrays also works with `ArrayLists`. Here are some common operations that are different with `ArrayLists`.

The length of a list is done with the `size` method; do not forget the parentheses.

```
public static void main(String[] args) {  
    List<String> pets = new ArrayList<>(Arrays.asList("dog",  
        "cat", "fish", "bird"));  
    System.out.println("The length of pets is: " +  
        pets.size());  
}
```

You do not use square brackets when working with elements in a list. To access a value, use the `get` method. To update the value of an element, use the `set` method.

```
public static void main(String[] args) {
    List<String> pets = new ArrayList<>(Arrays.asList("dog",
    "cat", "fish", "bird"));
    System.out.println("The first element is: " +
    pets.get(0));
    pets.set(0, "hamster");
    System.out.println("New array:");
    System.out.println(pets);
}
```

challenge

## Try this variation:

Use the `toArray` method to turn the `pets` list into an array. Be sure to pass the method a new array of strings the same size as `pets`.

### ▼ Solution

Here is one possible solution. We are using `.getClass().getName()` to prove that the result is actually an array instead of an `ArrayList`.

```
public static void main(String[] args) {
    List<String> pets = new ArrayList<>(Arrays.asList("dog", "cat", "fish", "bird"));
    String[] newPets = pets.toArray(new
    String[pets.size()]);
    System.out.println(newPets.getClass().getName());
}
```

Implementing the List ADT with an `ArrayList` offers many benefits like dynamic resizing. Most importantly, the built-in methods mean less functionality needs to be done manually. However, `ArrayLists` are not without their own disadvantages. They are not synchronized, which means multiple threads can concurrently access the `ArrayList`. This can lead to inconsistencies in the data. `ArrayLists` are generally slower than traditional arrays.

## The List Interface

You may have noticed that we declared the ArrayLists in a slightly different way. Traditionally, courses teach ArrayList declarations like this:

```
ArrayList<String> myList = new ArrayList<>();
```

Instead, we are using the List interface and implementing it with an ArrayList.

```
List<String> myList = new ArrayList<>();
```

This a common and preferred practice as it offers flexibility to your code base. The List interface is similar to the List ADT in that both can be implemented by different concrete data structures.

If, in the future, we decide that we need a linked list in our code base, we can replace new ArrayList with new LinkedList and our code will not break.

Going forward, we will use the List interface when declaring ArrayLists. We will also use the term “list” instead of ArrayList. The *what* is more important than the *how*.

# ArrayList Operations

Now that you're acquainted with arrays and ArrayLists in Java, let's delve deeper into some more operations you can perform with Java's List interface. We'll cover sorting, reversing, and sub-listing, along with their implications. We will tackle methods that define what the List ADT can do, but not how it does it.

## Sorting a List

Sorting is an essential operation in programming, and Java's Collections framework makes it extremely straightforward. Here's how to sort a list of integers:

```
public static void main(String[] args) {
    List<Integer> nums = new ArrayList<>(Arrays.asList(3, 1,
        4, 1, 5, 9));
    System.out.println("Before: " + nums);
    Collections.sort(nums);
    System.out.println("After: " + nums);
}
```

## Reversing a List

You can easily reverse a list using the `reversed()` method. Note, this method returns a new list object.

```
public static void main(String[] args) {
    List<String> cities = new ArrayList<>
        (Arrays.asList("Paris", "Tokyo", "Berlin"));
    System.out.println("Before: " + cities);
    System.out.println("After: " + cities.reversed());
}
```

## Sublists

Java allows you to create a sublist from an existing list using the `subList()` method. Note, the starting index is inclusive, while the ending index is not. Sublists are a shallow copy, so changes to them will affect the original list and vice versa.

```
public static void main(String[] args) {
    List<Integer> nums = new ArrayList<>(Arrays.asList(3, 1,
        4, 1, 5, 9));
    System.out.println("Before: " + nums);
    List<Integer> numsSubList = nums.subList(1,4);
    System.out.println("After: " + numsSubList);
}
```

## Contains a Value

The `contains` method takes a value returns a boolean indicating if the value was found in the list or not.

```
public static void main(String[] args) {
    List<Integer> nums = new ArrayList<>(Arrays.asList(3, 1,
        4, 1, 5, 9));
    Integer value = 5;
    boolean hasValue = nums.contains(value);
    System.out.println("Value " + value + " in list: " +
        hasValue);
}
```

## Find Index

Use the `indexOf` method to find the index of a value in the list. Note, the method will return the index for only the first occurrence of the value. If the value is not in the list, the method will return a -1.

```
public static void main(String[] args) {
    List<Integer> nums = new ArrayList<>(Arrays.asList(3, 1,
        4, 1, 5, 9));
    System.out.println("Location of '1' is at index: " +
        nums.indexOf(1));
}
```

## List Iterator

A `listIterator` is an object from the `java.utils` package. You can use it to iterate over a list.

```

public static void main(String[] args) {
    List<Integer> nums = new ArrayList<>(Arrays.asList(3, 1,
        4, 1, 5, 9));
    ListIterator<Integer> numIterator = nums.listIterator();

    while(numIterator.hasNext()) {
        System.out.println("Value: " + numIterator.next());
    }
}

```

If you pass the method a value, it will create the iterator object starting at that index.

```

public static void main(String[] args) {
    List<Integer> nums = new ArrayList<>(Arrays.asList(3, 1,
        4, 1, 5, 9));
    ListIterator<Integer> numIterator =
        nums.listIterator(2);

    while(numIterator.hasNext()) {
        System.out.println("Value: " + numIterator.next());
    }
}

```

You can even traverse over the list in reverse order using `hasPrevious` and `previous`.

```

public static void main(String[] args) {
    List<Integer> nums = new ArrayList<>(Arrays.asList(3, 1,
        4, 1, 5, 9));
    ListIterator<Integer> numIterator =
        nums.listIterator(nums.size());

    while(numIterator.hasPrevious()) {
        System.out.println("Value: " +
            numIterator.previous());
    }
}

```

## Remove on Condition

```

public static void main(String[] args) {
    List<Integer> nums = new ArrayList<>(Arrays.asList(3, 1,
        4, 1, 5, 9));
    nums.removeIf(n -> (n % 2 != 0));
    System.out.println("Even numbers: " + nums);
}

```

challenge

## Try this variation:

Assume the following list:

```
List<String> nums = new ArrayList<>()
((Arrays.asList("Juan", "Douglas", "Juliette",
"Clara"));
```

Implement a simple program that performs these operations:

- \* Sort a list of names.
- \* Reverse the sorted list.
- \* Create a sublist containing the first three names.
- \* Remove names if they start with the letter “J”.

### ▼ Code

Here is one possible solution:

```
public static void main(String[] args) {
    List<String> names = new ArrayList<>()
    (Arrays.asList("Juan", "Douglas", "Juliette",
    "Clara"));

    // Initial List
    System.out.println("Initial list: " + names);

    // Sort
    Collections.sort(names);
    System.out.println("After sorting: " + names);

    // Reverse
    Collections.reverse(names);
    System.out.println("After reversing: " + names);

    // Sublist
    List<String> sublist = names.subList(0, 3);
    System.out.println("Sublist (first 3 elements): " +
    sublist);

    // Remove names starting with 'J'
    names.removeIf(name -> name.startsWith("J"));
    System.out.println("After removing names starting
    with 'J': " + names);
}
```

The `List` interface in Java offers a rich set of operations beyond the basic add, remove, and search. Understanding these operations and their implementations can greatly enhance your Java programming skills, offering you a versatile toolset for data manipulation.

# **Formative Assessment 1**

## **Formative Assessment 2**

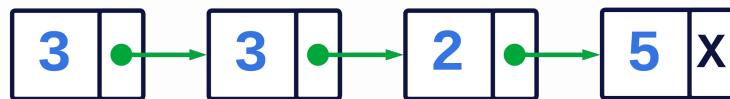
# **Learning Objectives**

**Learners will be able to...**

- **Define a linked list and its components**
- **Implement a linked list in java**
- **Apply different operations to a linked list, such as insertion, deletion, searching, and traversal**
- **Analyze the complexities of different linked list operations**

# Linked List

After exploring arrays and array-based lists, it's time to venture into another type of list that offers a different approach to storing and managing data: **linked lists**. Unlike arrays, which use a contiguous block of memory, linked lists are composed of individual elements known as **nodes**, each pointing to the next node in the list.



The image shows a linked list with the elements “3”, “3”, “2”, and “5”. The first three nodes have a pointer pointing to the next node in the list.

Our goal aims to demystify the concept of linked lists, focusing on their structure, types, basic operations, and use-cases. We will also compare linked lists to arrays, shedding light on their respective strengths and weaknesses.

## What is a Linked List?

A **Linked list** is a linear data structure made up of nodes, where each node contains data and a reference (or link) to the next node in the list.

```
class Node {  
    int data; // the data part  
    Node next; // the reference to the next node  
}
```

## Types of Linked Lists

There are several types of linked lists, including:

1. **Singly Linked List**: Each node points to the next node.
2. **Doubly Linked List**: Each node points to both the next and the previous nodes.
3. **Circular Linked List**: The last node points back to the first node.

This assignment will only deal with singly linked lists. The other variations will be covered in subsequent assignments.

## **Basic Operations**

Basic operations associated with linked lists include:

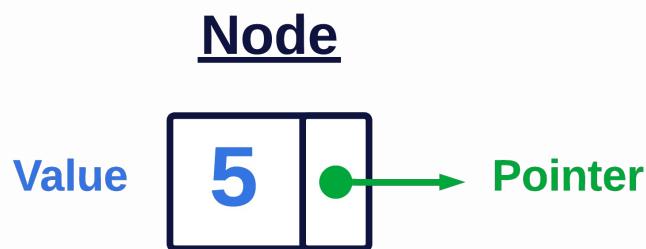
- **Insertion:** To add a new node.
- **Deletion:** To remove a node.
- **Traversal:** To go through the list.
- **Searching:** To find a node in the list.

# Components of a Linked List

## Breaking Down Linked Lists

When interacting with linked list you will often hear the following terms: **Value, pointers, head, tail.** It is important to understand these terms in order to grasp how linked lists function and how they perform operations like insertion, deletion, and traversal efficiently. The examples that follow use user-defined objects to define nodes and a linked list.

### Nodes



The image shows the components of a node. They are the value and the pointer.

For our example, nodes will be represented by the `Node` class. Just as a list is made up of elements, a linked list is made up with nodes. In a list, the job of an element is to store data. In a linked list, node stores information and points to the next node in the list.

When we talk about **value**, we are talking about the information stored in the node. In our example, the value is represented by the `data` field.

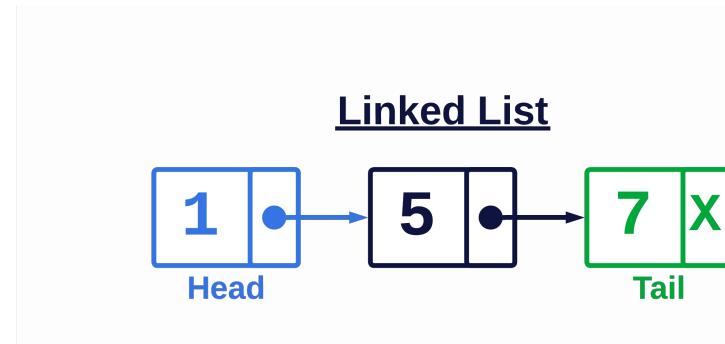
Nodes, like lists, have a data type. The example on the opposite page is a node that works with integers. When you instantiate a `Node` object, the value passed to the constructor will be stored in the `data` field.

In the context of a linked list, a **pointer** is simply a reference to another node. Unlike languages such as C or C++, Java does not allow for direct manipulation of memory addresses, but the concept is similar. The `next` field in our `Node` class acts like a pointer pointing to the next node in the list. As such, the `next` field is another `Node` object.

Every time that you add a new node to a linked list, it goes at the end. One of the defining characteristics of the last node is that it does not point to another node. Be sure that you instantiate a `Node` object with `null` as the

value for next.

## Linked Lists



The image shows the components of a linked list. They are the head and the tail. The head is the first node in the list and points to another node. The tail is the last node in a linked list. It does not point to another node.

Now that we have seen a node, we can talk about a linked list as a whole. Linked lists are dynamic, and its size is often unknown. However, we can say with certainty that every linked list has a first and last node.

The **head** is the first node in the linked list. When a linked list is empty, the head is null. When a new element is added to an empty list, the head points to this new element. The head is crucial for the traversal of the linked list; starting at the head, you can reach any node by following the next pointers.

### ▼ Default constructor

You may have noticed that the `LinkedList` class does not have a constructor. In these cases, Java uses something called a default constructor which is called behind the scenes. Default constructors give each attribute their default value. The only attribute in the class is `Node head`. User-defined objects have the default value of `null`.

The **tail** is the last node in the linked list. Unlike arrays, the tail in a singly linked list is not directly accessible; you have to traverse the list from the head to find it. However, in a doubly linked list or a circular linked list, you can keep a reference to the tail for quick access. The tail's next pointer is always null in a singly linked list, signifying the end of the list.

### ▼ Tails in other linked lists

In a singly linked list, there is no reference to the tail node. We can distinguish it because the `next` field has the value `null`. As mentioned above, you may see a reference to the tail node in other variations of a linked list. It would look something like this:

```
class LinkedList {  
    Node head;  
    Node tail;  
  
    // class code  
}
```

# Implementation

## Coding a Linked List

Taking what we know, let's build a linked list. We will start with the `Node` class, and then create the `LinkedList` class. For now, we are going to keep things simple. Our linked list list will work with integers. We will be able to add a node to the list, and then print the values stored in the list. We will discuss other operations later.

The `Node` class is used to represent each element in a linked list. It has two attributes, `data` and `next`. In the constructor, set the value of `data` to the constructor's parameter. By default, the `next` attribute should be `null`.

```
class Node {  
    int data; // Value stored in the node  
    Node next; // Reference to the next node in the linked list  
  
    // Constructor for initializing a Node with a given data  
    // value  
    public Node(int data) {  
        this.data = data;  
        this.next = null; // Initially, the node doesn't point  
        // to any other node  
    }  
}
```

Next, we need the `LinkedList` class to encapsulate the behavior of our linked list. We need only one attribute, `head`, which is a `Node` object. This will represent the first node in our linked list. We are going to use Java's default constructor, which sets the value of `head` to `null`. The value for `data` will be `0`. Creating a `LinkedList` object will create a node, but it does not have a value or a pointer. That job is left to the `add` method.

```

class LinkedList {
    Node head; // The head node of the linked list

    // Method to add a new node to the end of the linked list
    public void add(int data) {
        Node newNode = new Node(data); // Create a new node
        with the given data

    }

}

```

We need to be able to add a new node to the list. Create the add method, which takes an integer as a parameter. Create a new Node object with the value from the parameter. We then need to place the new node at the end of the list. There are two possible cases, there are no nodes in the list (the head node has the value null) or there are one or more nodes in the list. First check if the first node is null. If so, set head to the new node.

```

// Method to add a new node to the end of the linked list
public void add(int data) {
    Node newNode = new Node(data); // Create a new node
    with the given data

    // If the linked list is empty, make the new node the
    // head
    if (head == null) {
        head = newNode;
    }
}

```

If there are nodes in the list, then we need to traverse it to find the end. Create a temporary node and assign its value to head. Using a while loop, check to see if the pointer for the temporary node is not null. In this case, set the value of the temporary node to the next node in the list. Eventually, temp.next will be equal to null. That means we have reached the end of the list. In this case, set temp.next to the new node.

```

// Method to add a new node to the end of the linked list
public void add(int data) {
    Node newNode = new Node(data); // Create a new node with the given data

    // If the linked list is empty, make the new node the head
    if (head == null) {
        head = newNode;
    } else {
        // Otherwise, traverse to the last node and add the new node there
        Node temp = head;
        while (temp.next != null) {
            temp = temp.next; // Go to the next node
        }
        temp.next = newNode; // Make the last node point to the new node
    }
}

```

The last thing we want to do for this example is to print the linked list so we can see that our values are properly stored in the list. Create the `printList` method. Similar to the `add` method, we are going to traverse the list. Create a temporary node whose initial value is the first node in the linked list. Print the value for the `data` attribute, and then move the temporary node to the next node in the list. Once we reach the end, print a newline character.

```

// Method to print all the elements in the linked list
public void printList() {
    Node temp = head; // Start from the head node
    while (temp != null) {
        System.out.print(temp.data + " "); // Print the data of each node
        temp = temp.next; // Go to the next node
    }
    System.out.println(); // Print a newline after traversing all nodes
}

```

### ▼ Code

Your code should look like this:

```

class Node {
    int data; // Value stored in the node
    Node next; // Reference to the next node in the linked list

    // Constructor for initializing a Node with a given data
    // value
    public Node(int data) {
        this.data = data;
        this.next = null; // Initially, the node doesn't point
        // to any other node
    }
}

class LinkedList {
    Node head;

    // Method to add a new node to the end of the linked list
    public void add(int data) {
        Node newNode = new Node(data); // Create a new node
        // with the given data

        // If the linked list is empty, make the new node the
        // head
        if (head == null) {
            head = newNode;
        } else {
            // Otherwise, traverse to the last node and add the
            // new node there
            Node temp = head;
            while (temp.next != null) {
                temp = temp.next; // Go to the next node
            }
            temp.next = newNode; // Make the last node point to
            // the new node
        }
    }

    // Method to print all the elements in the linked list
    public void printList() {
        Node temp = head; // Start from the head node
        while (temp != null) {
            System.out.print(temp.data + " ");
            // Print the
            // data of each node
            temp = temp.next; // Go to the next node
        }
        System.out.println(); // Print a newline after
        // traversing all nodes
    }
}

```

Create the `main` method so that you instantiate a `LinkedList` object and add 1, 3, 5, 7, and 9 to the end of the list. Then print the linked list.

```
public static void main(String[] args) {  
    LinkedList list = new LinkedList();  
  
    // Add elements  
    list.add(1);  
    list.add(3);  
    list.add(5);  
    list.add(7);  
    list.add(9);  
  
    list.printList();  
}
```

challenge

## Try this variation:

Update the `main` method so that you instantiate a `LinkedList` object but do not add any nodes. Then print it.

```
public static void main(String[] args) {  
    LinkedList list = new LinkedList();  
  
    // Print list  
    System.out.println("Linked List after adding  
    elements:");  
    list.printList();  
}
```

### ▼ What is happening?

Java's default constructor should create the `head` and give it the value of `null`. Yet, the program does not print anything from the linked list. Why? If you at the `printList` method, you will notice that the while loop only runs if the temporary node is not `null`. The default constructor will set the value of `head` to `null`, so the loop never runs.

# Linked List Operations

## Basic Operations

Having established a fundamental understanding of pointers, the head, and the tail of a linked list, let's now look at some basic operations like traversal, insertion, deletion, and searching. We will use our previously defined `Node` and `LinkedList` classes for this purpose.

### Traversal

Traversing a linked list involves walking through the list from the head to the last node. We have seen examples of this with the `add` and `printList` methods. Linked lists do not have the `size` method or `length` attribute. We cannot know the length of any given linked list ahead of time, so we need to utilize a while loop. Create a temporary node. This will act as a pointer to each node in the list. Start with the head node, and as long as the temporary node is not null, advance to the next node by setting the value of the temporary node to the pointer of the current node.

```
public int countNodes() {  
    Node temp = head;  
    int counter = 0;  
    while (temp != null) {  
        counter += 1;  
        temp = temp.next;  
    }  
    return counter;  
}
```

After you add the `countNodes` method to the `LinkedList` class, update the `main` method to display the number of nodes in the list.

```

public static void main(String[] args) {
    LinkedList list = new LinkedList();

    // Add elements
    list.add(1);
    list.add(2);
    list.add(3);

    // Count nodes
    System.out.println("There are " + list.countNodes() + " nodes.");
}

```

## Insertion at the Beginning

To insert a node at the beginning of the list, you have to change the head to the new node and make the new node point to the original head. Add the following method to the `LinkedList` class.

```

public void insertAtBeginning(int data) {
    Node newNode = new Node(data);
    newNode.next = head;
    head = newNode;
}

```

Update the `main` method to insert a new node with the value of 4 at the beginning of the list. Use the `printList` method to verify that it works as expected.

```

public static void main(String[] args) {
    LinkedList list = new LinkedList();

    // Add elements
    list.add(1);
    list.add(2);
    list.add(3);
    list.insertAtBeginning(4);

    // Print nodes
    System.out.println("Linked List after adding elements:");
    list.printList();
}

```

## Insertion at the End

To add a node at the end of the list, you have to traverse the list to the last node and then link the new node. The logic for this already exists in the `add` method, however `insertAtEnd` does a better job at describing what the method does. Replace the `add` method with the one below.

```
public void insertAtEnd(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        return;
    }
    Node last = head;
    while (last.next != null) {
        last = last.next;
    }
    last.next = newNode;
}
```

Update the `main` method so that it calls `insertAtEnd` instead of `add`.

```
public static void main(String[] args) {
    LinkedList list = new LinkedList();

    // Add elements at the end
    list.insertAtEnd(1);
    list.insertAtEnd(2);
    list.insertAtEnd(3);

    // Print list
    System.out.println("Linked List after adding
elements:");
    list.printList();
}
```

## Deletion

To delete a node, you first need to find it and then rearrange the pointers to exclude it from the list. There are three different cases for this operation. The node to delete is the head, the node to delete is not in the list, or the node to delete is somewhere in the list. Let's take a look how to write code for each of the cases. We are going to need two additional `Node` objects when performing a deletion – `temp` and `prev`.

First, check to see if the node to delete is the head. We know this to be true if value of the `head` is not `null` and the value in the node matches the value we want to delete. If so, set the `head` attribute to `temp.next`. Now, the `head` attribute points to the second node in the list.

```
public void deleteNode(int data) {
    Node temp = head, prev = null;
    if (temp != null && temp.data == data) {
        head = temp.next;
        return;
    }
}
```

If the node to delete is not the head, then we need to traverse the list. In this case, we want keep progressing through the list as long as the `temp` is not `null` and the value of the node is not the value we want to delete. When this loop stops, it could be for one of two reasons: you have reached the end of the list or you have found the value to delete. Notice how we iterate. The `temp` node moves to the next node in the list, while `prev` is one node behind `temp`.

```
public void deleteNode(int data) {
    Node temp = head, prev = null;
    if (temp != null && temp.data == data) {
        head = temp.next;
        return;
    }
    while (temp != null && temp.data != data) {
        prev = temp;
        temp = temp.next;
    }
}
```

If we are at the end of the list (`temp == null`), that means the value to delete was not found. Use a `return` statement to exit the method. If the loop ended because you found the value to delete, set the value of `next` attribute of `prev` to the `next` attribute of the `temp` node. In effect, `prev` now points to the node after `temp`, which “deletes” the node since we will never see it again on a subsequent traversal.

```

public void deleteNode(int data) {
    Node temp = head, prev = null;
    if (temp != null && temp.data == data) {
        head = temp.next;
        return;
    }
    while (temp != null && temp.data != data) {
        prev = temp;
        temp = temp.next;
    }
    if (temp == null) return;
    prev.next = temp.next;
}

```

Update the `main` method to print the linked before and after the deletion.

```

public static void main(String[] args) {
    LinkedList list = new LinkedList();

    // Add elements
    list.insertAtEnd(1);
    list.insertAtEnd(3);
    list.insertAtEnd(5);
    list.insertAtEnd(7);
    list.insertAtEnd(9);

    // delete node
    System.out.println("Linked List before deletion:");
    list.printList();

    System.out.println("Linked List after deletion:");
    list.deleteNode(7);
    list.printList();
}

```

## Searching

To search for a node in a linked list, you are going to use a linear search. Start at the head and traverse until you find the element or reach the end of the list. Return `true` if you find the value, or `false` if you do not. Unlike searching an array, you do not return an index, as direct access to a value is not done in a linked list.

```
public boolean search(int data) {
    Node current = head;
    while (current != null) {
        if (current.data == data)
            return true;
        current = current.next;
    }
    return false;
}
```

Update the `main` method to search for two different values – one that is in the list and another that is not.

```
public static void main(String[] args) {
    LinkedList list = new LinkedList();

    // Add elements
    list.insertAtEnd(1);
    list.insertAtEnd(3);
    list.insertAtEnd(5);
    list.insertAtEnd(7);
    list.insertAtEnd(9);

    // search list
    System.out.println(list.search(3));
    System.out.println(list.search(2));
}
```

These basic operations form the core of linked list manipulation. Understanding how they work is essential for both utilizing linked lists effectively and for building more complex data structures based on linked lists.

# Advantages and Disadvantages of Linked Lists

After covering the basics of linked list operations, it's essential to understand where and when to use linked lists. In this section, we will talk about the advantages and disadvantages of using linked lists compared to arrays, and highlight scenarios where one may be preferable over the other.

## Advantages

Linked lists may not have the built-in methods and attributes of arrays and ArrayLists, but their unique structure does provide the following benefits:

- **Dynamic Size:** Linked lists dynamically allocate memory, which is a significant advantage over arrays with fixed size.
- **Ease of Insertion/Deletion:** Inserting a new node or deleting a node is easier and faster compared to arrays as you just need to update some pointers. There's no need to shift elements.
- **No Wasted Space:** Because linked lists allocate memory as needed, they do not waste space like arrays can.

## Disadvantages

Linked lists, however, have their own set of tradeoffs. Here are some reasons you may not want to implement a linked list:

- **Memory Usage:** Each node in a linked list requires extra memory for the next pointer, which can be a disadvantage for large lists.
- **Traversal:** Elements are stored in non-contiguous memory locations. Therefore, traversal can be slower than that of arrays.
- **Random Access:** Direct/random access is not allowed. One must traverse the list from the head node to find the desired element.

## When to Use Linked Lists

Given the above information, here are some scenarios in which a linked list may be a good choice:

- **Unknown Size:** When you don't know the size of the list in advance.
- **Frequent Additions/Deletions:** When you have frequent insertions and deletions, and these operations need to be fast.

- **Stack/Queue Implementations:** They can be used to implement other data structures like stacks and queues.

You should avoid using a linked list when you need fast, random access to the values stored in the list. This operation relies too much on traversal to offer a fast solution. You should also consider not using a linked list when memory is a concern. While linked lists allocate memory as needed, storing the pointer to the next node does require some additional memory.

## Code Comparison

Just for a recap, let's compare how one might implement insertion in an `ArrayList` and a linked list.

While both `ArrayList` and linked lists offer  $O(1)$  insertion at the end, linked lists can provide  $O(1)$  insertion at the beginning and in the middle, given that you have a reference to the node after which you want to insert.

# Complexities of Linked List Operations

## Analyzing Complexities of Linked List Operations

Having discussed the general advantages and disadvantages of linked lists, let's delve into the performance aspect, specifically the time complexity for different operations.

### Time Complexities

- **Insertion at the Beginning:**  $O(1)$  - Constant time as you only need to change a few pointers.
- **Insertion at the End:**  $O(n)$  - You may need to traverse the list to insert at the end, unless you maintain a tail pointer.
- **Deletion:**  $O(n)$  - In the worst case, you may have to traverse the list to find the node to be deleted.
- **Traversal:**  $O(n)$  - You have to walk through the list from the head to the tail.
- **Searching:**  $O(n)$  - Similar to traversal, you may need to go through each element in the list.

There are some best case scenarios for linked lists. However, these scenarios require advance knowledge of the node being acted upon. The only node that is known is the head, which is why insertion at the beginning is so quick. All other operations make use of traversal, which explains the  $O(n)$  time complexity.

#### ▼ Keeping track of the tail

Some implementations of a linked list will keep track of the tail. In these instances, you can quickly insert at the end of the list because no traversal is required. Not all linked lists, however, keep track of the tail node.

### Space Complexity

The space complexity for a linked list is  $O(n)$ . In addition, each node takes up some additional space for storing the 'next' pointer, aside from the data.

## Comparing with Array Operations

Let's take a look at how some common operations and compare them when using linked lists and arrays.

## Insertion

In an ArrayList, you can have fast insertions by adding an element to the end of the list with the add method.

```
// insertion in an ArrayList

List<Integer> list = new ArrayList<>();
list.add(1); // O(1) time complexity
```

You can also add elements to other locations in the list. For example, `list.add(0, 2)` adds the value 2 to the beginning of the list. However, this requires a “shifting” of the list – all of the existing elements need a new index. In this case, insertion goes from  $O(1)$  to  $O(n)$ .

In a linked list, the fast insertion happens at the beginning of the list. This is because the head is always known.

```
// insertion in a linked list

public void insertAtBeginning(int data) {
    Node newNode = new Node(data);
    newNode.next = head;
    head = newNode; // O(1) time complexity
}
```

The lack of indices means that appending after any node with given value requires traversal. Again, insertion goes from  $O(1)$  to  $O(n)$ .

## Deletion

We see a similar relationship between linked lists and arrays when it comes to deletion. The performance could be  $O(1)$  under the right circumstance, or it can be  $O(n)$ . In the case of arrays, deletions are fast when removing the last element as there is no shifting that needs to be done.

Likewise, deletions are fast in a linked list when the node to remove is known (typically the head). However, if you do not know where the node is in the list, you have to rely on traversal to find it.

## Searching

Arrays offer much more flexibility when it comes to searching. If you know the index, then searching is  $O(1)$ . If the target element is unknown, there are several searching algorithms that provide better performance than a simple linear search.

Linked lists, however, have one known node (head). There may be a few implementations keeping track of the tail, but these are not common. Known nodes have a time complexity of  $O(1)$ . Unknown nodes require traversal, which is  $O(n)$ .

In summary, linked lists offer more flexibility than arrays but come at the cost of additional memory and slower random access times. Your choice between the two should depend on the specific needs and constraints of your project.

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

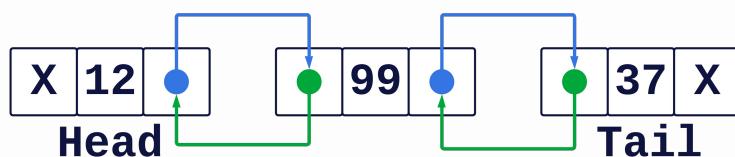
- **Define doubly linked list**
- **Differentiate a doubly linked list from a singly linked list**
- **Implement a doubly linked**
- **Analyze the complexities of different linked list operations**

# Doubly Linked Lists

## What is a Doubly Linked List?

Now that we have a good understanding of a singly linked list, we can extend this concept and introduce a doubly linked list. A **doubly linked list (DLL)** is a list of nodes, where each node has two pointers. One references the next node in the list, while the other pointer references the previous node. This allows for bidirectional traversal, making some algorithms more efficient and easier to implement.

The image below is a representation of a doubly linked list. In the case of our head node, we can see the pointer for the previous node has the value `null`, while the pointer for the next node references the next Node object. The tail node has a pointer that references the previous Node object, but the next pointer has the value of `null`. Nodes between the head and tail have pointers that reference both the previous and next nodes.



[.guides/img/DLL](#)

## Implementing a Doubly Linked List

To create a doubly linked list, we are going to make some slight changes to the code for a singly linked list. Starting with the `Node` class, add the attribute `prev` that is of type `Node`. In the constructor, set the initial value of `prev` to `null`.

```
class Node {
    int data;
    Node next;
    Node prev;

    Node(int data) {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}
```

Next, create the `DoublyLinkedList` class. Just as before, this class will encapsulate all of the operations we want to perform on the doubly linked list. However, we are going to keep track of the tail node. This is not a requirement of a doubly linked list, but you should see how doing this adds some efficiency.

```
class DoublyLinkedList {
    Node head;
    Node tail;

}
```

We also need to create the `printList` method so we can see the changes we make to our doubly linked list. The logic of this method does not change from a singly linked list to a doubly linked list. Create a `Node` object set to the `head`. As long as this object is not `null` print the value stored in the node. Update the node to the `next` attribute.

```
public void printList() {
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.data + " ");
        temp = temp.next;
    }
    System.out.println();
}
```

# Insertion

## Adding Nodes

We will delve into the nitty-gritty details of each operation in a doubly linked list. The inclusion of a `prev` pointer in each node in a doubly linked list allows us to navigate the list in both directions, which can lead to some optimizations and ease of manipulation. Understanding these operations in detail provides you with the ability to manipulate doubly linked lists more efficiently. Let's start with insertion.

### At the Beginning

Inserting a node at the beginning is straightforward. You allocate a new node, set its `next` pointer to the current head. If the doubly linked list has already has a node in it, then you update the `prev` pointer of the old head to point to the new node. Update the `head` attribute to point to the new node. If our list only has one node (`head.next` is `null`) then that node is both the head and the tail. In this case, update `tail` to point to the newly added node.

```
public void insertAtBeginning(int data) {
    Node newNode = new Node(data);
    newNode.next = head;

    if (head != null) {
        head.prev = newNode;
    }

    head = newNode;

    if (head.next == null) {
        tail = newNode;
    }
}
```

In the `DLL` class toward the bottom of the IDE, create the `main` method that instantiates a `DoublyLinkedList` object and adds a node to the beginning. Print the list.

```
public static void main(String[] args) {
    DoublyLinkedList list = new DoublyLinkedList();
    list.insertAtBeginning(42);
    list.printList();
}
```

## At the End

Because we are keeping track of the tail node, inserting at the end of the list is much more efficient. Start by creating a new node. If the list is empty (head is equal to null) then set head and tail to the new node. Otherwise set the next attribute of the old tail to the new node. Then set the prev attribute of the new node to the old tail. Finally, set the tail attribute to the new node.

```
public void insertAtEnd(int data) {
    Node newNode = new Node(data);

    if(head == null) {
        head = newNode;
        tail = newNode;
    } else {
        tail.next = newNode;
        newNode.prev = tail;
        tail = newNode;
    }
}
```

Update the `main` method to add a node at the end of the list. Just to verify that our logic is correct, call the `insertAtEnd` method before calling `insertAtBeginning`.

```
public static void main(String[] args) {
    DoublyLinkedList list = new DoublyLinkedList();
    list.insertAtEnd(3325);
    list.insertAtBeginning(42);
    list.printList();
}
```

## At a Specific Position

To insert at a specific position, traverse the list to that point, update the next and prev pointers for the node before and the node after, and then insert the new node in between.

The `insertAtPos` method should take data for the new node as well as the position of the insertion. First check to see if the position is not valid; i.e. is it less than 1. Print a message to the user for an invalid position. Note, linked lists do not have indices so we are going to assume the first position is represented by the number 1. If the position is 1, then call the `insertAtBeginning` method and pass it the data variable.

```
public void insertAtPos(int data, int pos) {  
    if (pos < 1) {  
        System.out.println("Invalid position.");  
        return;  
    }  
  
    if (pos == 1) {  
        insertAtBeginning(data);  
    } else {  
  
    }  
}
```

Next, create the `current` variable which is a pointer to the current node in the list. We also need a new node that will be inserted. The variable `currentPos` is a counter variable to help us end the traversal. Use a while loop to traverse the list. Stop when `currentPos` is greater than or equal to the specified position minus 1 or when the value of `current` is null (gone past the end of the list).

```

public void insertAtPos(int data, int pos) {
    if (pos < 1) {
        System.out.println("Invalid position.");
        return;
    }

    if (pos == 1) {
        insertAtBeginning(data);
    } else {
        Node current = head;
        Node newNode = new Node(data);
        int currentPos = 1;

        // Traverse the list to find the node before the
desired position.
        while (currentPos < pos - 1 && current != null) {
            current = current.next;
            currentPos++;
        }

    }
}

```

If we have traversed past the end of the list (current is equal to null), print a message that the specified position is not valid.

```

public void insertAtPos(int data, int pos) {
    if (pos < 1) {
        System.out.println("Invalid position.");
        return;
    }

    if (pos == 1) {
        insertAtBeginning(data);
    } else {
        Node current = head;
        Node newNode = new Node(data);
        int currentPos = 1;

        // Traverse the list to find the node before the
        // desired position.
        while (currentPos < pos - 1 && current != null) {
            current = current.next;
            currentPos++;
        }

        if (current == null) {
            System.out.println("Invalid position.");
            return;
        }

    }
}

```

Finally, we are ready to insert the new node. Set the prev attribute of the new node to the current node. Set the next attribute of the new node to the next attribute of the current node. If the current node is the tail, set the tail to the new node. Otherwise, set the prev attribute of the node past the current node to the new node. Lastly, set the next attribute of the current node to the new node.

```

public void insertAtPos(int data, int pos) {
    if (pos < 1) {
        System.out.println("Invalid position.");
        return;
    }

    if (pos == 1) {
        insertAtBeginning(data);
    } else {
        Node current = head;
        Node newNode = new Node(data);
        int currentPos = 1;

        // Traverse the list to find the node before the
        // desired position.
        while (currentPos < pos - 1 && current != null) {
            current = current.next;
            currentPos++;
        }

        if (current == null) {
            System.out.println("Invalid position.");
            return;
        }

        // Adjust links to insert the new node at the
        // specified position.
        newNode.prev = current;
        newNode.next = current.next;
        if (current.next == null) {
            tail = newNode;
        } else {
            current.next.prev = newNode;
        }
        current.next = newNode;
    }
}

```

Update the `main` method so that our doubly linked list has four nodes in it. Then add a node at the third position.

```
public static void main(String[] args) {
    DoublyLinkedList list = new DoublyLinkedList();
    list.insertAtBeginning(1);
    list.insertAtEnd(2);
    list.insertAtEnd(3);
    list.insertAtEnd(4);
    list.insertAtPos(5, 3);
    list.printList();
}
```

challenge

### Try these variations:

- Test the `insertAtPos` method with the valid positions: 1, 2, 3, 4, and 5.
- Test the `insertAtPos` method with the invalid positions: `0` and `6`.

# Deletion

## Removing Nodes

Just like the insertion operations, we are going to cover how to remove nodes from the beginning, end, and at a specific position in a list.

### From the Beginning

Deleting from the beginning is straightforward. Make the head point to the second node (`head.next`) and update the `prev` pointer of the new head to `null`.

```
public void deleteAtBeginning() {
    head = head.next;
    head.prev = null;
}
```

Update the `main` method so that the list contains the values 1, 2, 3, and 4. Then remove the first node from the list with the `deleteAtBeginning` method.

```
public static void main(String[] args) {
    DoublyLinkedList list = new DoublyLinkedList();
    list.insertAtBeginning(1);
    list.insertAtEnd(2);
    list.insertAtEnd(3);
    list.insertAtEnd(4);
    list.deleteAtBeginning();
    list.printList();
}
```

### From the End

Because we are keeping track of the tail node, you can quickly delete the last node. Update the `tail` to point to the second last node and update the `next` pointer of the new tail to `null`.

```
public void deleteAtEnd() {
    tail = tail.prev;
    tail.next = null;
}
```

Update the `main` method so that the list contains the values 1, 2, 3, and 4. Then remove the tail node from the list with the `deleteAtEnd` method.

```
public static void main(String[] args) {
    DoublyLinkedList list = new DoublyLinkedList();
    list.insertAtBeginning(1);
    list.insertAtEnd(2);
    list.insertAtEnd(3);
    list.insertAtEnd(4);
    list.deleteAtEnd();
    list.printList();
}
```

## From a Specific Position

Deleting a node from a specific position is similar to insertion, however, there are a few important differences. Start by validating that the position to remove is greater than 1. If the node to remove is at position 1, call the `deleteAtBeginning` method. This traversal will stop at the node to be removed (`currentPos < pos`). Again, if you traverse past the end of the list, print a message to the user.

```

public void deleteAtPos(int pos) {
    if (pos < 1) {
        System.out.println("Invalid position.");
        return;
    }

    if (pos == 1) {
        deleteAtBeginning();
    } else {
        Node current = head;
        int currentPos = 1;

        // Traverse the list to find the node at the desired
        position.
        while (currentPos < pos && current != null) {
            current = current.next;
            currentPos++;
        }

        if (current == null) {
            System.out.println("Invalid position.");
            return;
        }

    }
}

```

When it comes to actually removing the node, set the `next` attribute of the node before the current node to `next` attribute of the current node. If the current node is the tail, set the tail to the `prev` attribute of the current node. Otherwise, set the `prev` attribute of the node after the current node to the node that comes before the current node.

```

public void deleteAtPos(int pos) {
    if (pos < 1) {
        System.out.println("Invalid position.");
        return;
    }

    if (pos == 1) {
        deleteAtBeginning();
    } else {
        Node current = head;
        int currentPos = 1;

        // Traverse the list to find the node at the desired
        // position.
        while (currentPos < pos && current != null) {
            current = current.next;
            currentPos++;
        }

        if (current == null) {
            System.out.println("Invalid position.");
            return;
        }

        // Adjust links to delete the node at the specified
        // position.
        current.prev.next = current.next;
        if (current.next == null) {
            tail = current.prev;
        } else {
            current.next.prev = current.prev;
        }
    }
}

```

Update the `main` method so that the list contains the values 1, 2, 3, and 4. Then remove the second node from the list with the `deleteAtPos` method.

```
public static void main(String[] args) {
    DoublyLinkedList list = new DoublyLinkedList();
    list.insertAtBeginning(1);
    list.insertAtEnd(2);
    list.insertAtEnd(3);
    list.insertAtEnd(4);
    list.deleteAtPos(2);
    list.printList();
}
```

challenge

### Try these variations:

- Test the `deleteAtPos` method with the valid positions: 1, 2, 3, and 4.
- Test the `deleteAtPos` method with the invalid positions: 0 and 5.

# Traversal & Searching

## Navigating a List

Having pointers for both the previous and next nodes allows you to move around a doubly linked list with a bit more freedom.

### Traversing

The pointer for the previous node (as well as knowing the tail) means you can traverse a list in reverse. Create the method `reversePrint` which will print out the values of the nodes in reverse order. Instantiate the variable `current` and set it to the tail. As long as the current node is not `null`, print the value stored in the node and set `current` to the previous node.

```
public void reversePrint() {
    Node current = tail;

    while (current != null) {
        System.out.print(current.data + " ");
        current = current.prev;
    }
    System.out.println();
}
```

Update the `main` method to add four nodes to the list, and then call the `reversePrint` method.

```
public static void main(String[] args) {
    DoublyLinkedList list = new DoublyLinkedList();
    list.insertAtBeginning(1);
    list.insertAtEnd(2);
    list.insertAtEnd(3);
    list.insertAtEnd(4);
    list.reversePrint();
}
```

### Searching

Searching a doubly linked list is still done in a linear manner. Let's take a look at how we can simplify some of our other operations using a search method.

Start by creating the `findNode` method. This method takes a value stored in a node as a parameter. Create the `current` variable and set its value to the head. Traverse through the list. If the data in the node matches the value passed to the method, return the node. If you finish traversing the list and have not found the value, return `null` which signifies the node is not in the list.

```
public Node findNode(int value) {
    Node current = head;

    while (current != null) {
        if (current.data == value) {
            return current;
        }
        current = current.next;
    }

    return null; // node not found
}
```

We can now create the `deleteNode` method that removes a given node from the list. Pass the method the node you want to delete. If either the list is empty (`head` is `null`) or the node is not in the list (`del` is `null`) then use the `return` statement to exit the method. If the list only has one node (`head == tail`), then set both the `head` and `tail` to `null`. If you are deleting the `head`, then call `deleteAtBeginning`. If you are deleting the `tail`, call `deleteAtEnd`. Otherwise set the `next` attribute of the node before `del` to the node after `del`. Then set the `prev` attribute of the node after `del` to the node before `del`.

```

public void deleteNode(Node del) {
    if (head == null || del == null) return;

    if (head == tail) {
        head = null;
        tail = null;
        return;
    }

    if (head == del) {
        deleteAtBeginning();
        return;
    }

    if (tail == del) {
        deleteAtEnd();
        return;
    }

    del.prev.next = del.next;
    del.next.prev = del.prev;
}

```

Update the `main` method so there are four nodes in the list. Use the `findNode` method to locate the node with the value of 3. Pass this node to the `deleteNode` method.

```

public static void main(String[] args) {
    DoublyLinkedList list = new DoublyLinkedList();
    list.insertAtBeginning(1);
    list.insertAtEnd(2);
    list.insertAtEnd(3);
    list.insertAtEnd(4);
    Node target = list.findNode(3);
    list.deleteNode(target);
    list.printList();
}

```

challenge

**Try these variations:**

- Test the `deleteNode` method with the valid positions: 1, 2, 3, and 4.
- Test the `deleteNode` method with the invalid positions: 0 and 5.

# Doubly Linked List Analysis

## Time Complexity Analysis of Doubly Linked Lists Operations

Understanding the time complexity of various operations in data structures is crucial for writing efficient code. In this section, we will analyze the time complexity for the fundamental operations in a doubly linked list : insertion, deletion, traversal, and searching.

### Insertion

Time complexities for insertion operations depend on where the node is being inserted and if the tail node is known.

- **At the Beginning** - Inserting a node at the beginning of a DLL takes constant time  $O(1)$ , as you're merely changing a few pointers and updating the head of the list.
- **At the End** - Inserting a node at the end can be  $O(n)$  if you start from the head and traverse the list to find the last element. However, this operation can be optimized to  $O(1)$  if a tail pointer is maintained.
- **At a Specific Position** - Inserting a node at a specific position would require traversing the list up to that point, making the time complexity  $O(n)$ .

### Deletion

Just like insertion, the time complexity can vary based on the location of the node to be deleted and knowing the tail node.

- **At the Beginning** - Deleting at the beginning takes  $O(1)$  time.
- **At the End** - Deleting at the end also takes  $O(1)$  time if a tail pointer is maintained; otherwise, it takes  $O(n)$ .
- **At a Specific Position** - If the node to be deleted is known and given as a pointer, the operation can be done in  $O(1)$ . Otherwise, it's  $O(n)$ .

### Traversal & Searching

- **Traversal** - Traversing the entire list to perhaps print out elements or perform any operation on them will take  $O(n)$  time, as you have to visit each node once.

- **Searching** - Searching for an element would require traversing the list until you find the element. In the worst case, you'd need to go through the entire list, yielding a time complexity of  $O(n)$ .

## Table of Operations

The table below summarizes the time complexities for the various operations covered. This includes optimized and non-optimized doubly linked lists.

| Operation                    | Time Complexity  | Optimized Time Complexity  |
|------------------------------|------------------|----------------------------|
| <b>Insertion (Beginning)</b> | $O(1)$           | $O(1)$                     |
| <b>Insertion (End)</b>       | $O(n)$           | $O(1)$ (with tail pointer) |
| <b>Insertion (Position)</b>  | $O(n)$           | $O(n)$                     |
| <b>Deletion (Beginning)</b>  | $O(1)$           | $O(1)$                     |
| <b>Deletion (End)</b>        | $O(n)$           | $O(1)$ (with tail pointer) |
| <b>Deletion (Position)</b>   | $O(n)$ or $O(1)$ | $O(1)$ (if node is known)  |
| <b>Traversal</b>             | $O(n)$           | $O(n)$                     |
| <b>Searching</b>             | $O(n)$           | $O(n)$                     |

# **Formative Assessment 1**

## **Formative Assessment 2**

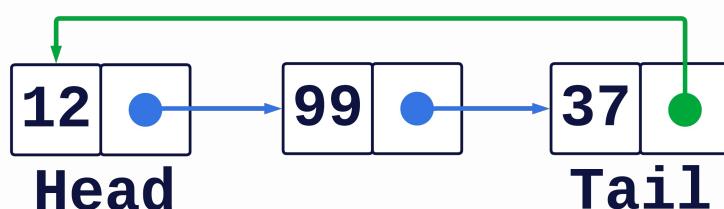
# **Learning Objectives**

**Learners will be able to...**

- **Define a circular linked list**
- **Differentiate circular linked lists from singly linked and doubly linked lists**
- **Implement a circular linked lists**
- **Analyze the complexities of different linked list operations**

# Circular Linked Lists

In contrast to singly and doubly linked lists where the last node points to null, a circular linked list is slightly different. In a **circular linked list (CLL)**, the last node points back to the first node, thereby forming a loop. This unique characteristic has its applications and complexities, and we will delve into the specific operations that can be performed on a CLL. Circular linked lists offer unique advantages such as the ease of rotation, but they also come with their challenges, especially in terms of traversal and maintenance of the circular structure.



The image depicts a circular linked list. There are three nodes in the list. The first node is labeled “Head” and has the value 12. Its pointer points to the next node with the value 99. The second node points to the third node which is labeled “Tail” and has the value 37. The tail node points all the way back to the head node.

## Implementation

Circular linked lists can be either singly or doubly linked. However, it more common to see a circular singly linked list due to its lower memory overhead (one pointer vs two pointers). The big risk you run with a circular linked list is getting stuck in an infinite loop when traversing. To optimize performance, we are only going to manage the head and tail nodes.

```
class Node {  
    Node next;  
    int data;  
  
    public Node(int data) {  
        this.data = data;  
        next = null;  
    }  
}
```

Create the `CircularList` class that has the `head` and `tail` attributes. We are also going to create the `printList` method so we can see how operations alter the list. We are going to start the traversal the same way as singly and doubly linked lists. However, the defining characteristic of a circular list is that the tail points back to the head. That means there are no nodes with a pointer whose value is `null`. Basically, we have initiated an infinite loop.

The logic behind traversing a circular list is to keep traversing until you get back to the head node. Because of this, the order of our code is important. The first thing you should do inside the while loop is the desired action. In this case, we are printing a value (you could just as easily search for a value, modify a value, etc.). After the action is performed, then advance to the next node. Finally, ask if the new node is the head. If so, break out of the loop.

```
class CircularList {
    Node head;
    Node tail;

    public void printList() {
        Node current = head;

        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;

            if (current == head) {
                System.out.println();
                break;
            }
        }
    }
}
```

If we are keeping track of the tail, why not traverse until we reach the tail? Assume that we were to change the `printList` method to the code below. **Important**, this code is faulty, do not add it to your program.

```
public void printList() {
    Node current = head;

    while (current != tail) {
        System.out.print(current.data + " ");
        current = current.next;

        if (current == head) {
            System.out.println();
            break;
        }
    }
}
```

The method would traverse up until the tail node. Once it reaches the tail node, however, the while loop would end and therefore not print the value of the tail node. We could update the method to check if the current node is equal to the tail, then break out of the loop. This way, the traversal would touch all of the nodes.

```
public void printList() {
    Node current = head;

    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;

        if (current == tail) {
            System.out.println();
            break;
        }
    }
}
```

However, the code directly above only works for circular lists with two pointers per node. If we go back to the original version of `printList`, this traversal will work with singly linked and doubly linked circular lists.

# Insertion

## Adding Nodes

### At the Beginning

When adding a node to the beginning of a list, start by creating a new node. We also need a temporary node, which we set to the head. Then make the new head point to the new node. Now that we have a new head, its `next` attribute should point to the temporary node, which represents the rest of the list.

```
void insertAtBeginning(int data) {
    Node newNode = new Node(data);
    Node temp = head;
    head = newNode;
    newNode.next = temp;

}
```

The last step is to have the `next` field of the tail node point to the new head. However, we need to take into account the fact that the node we just added may be the only node in the list. Check to see if the temporary node is `null` (only one node in the list). If so, set the tail to the new node and have the `next` attribute point back to itself. Otherwise set the `next` attribute of the tail point back to the new head.

```
void insertAtBeginning(int data) {
    Node newNode = new Node(data);
    Node temp = head;
    head = newNode;
    newNode.next = temp;

    if(temp == null) {
        // List has one node, make the new node the tail.
        tail = newNode;
        newNode.next = newNode;
    } else {
        tail.next = newNode;
    }
}
```

In the `CLL` class, create the `main` method. Use the `insertAtBeginning` method to add the values 1, 2, and 3 to the list. Then print the list.

```
public static void main(String[] args) {
    CircularList list = new CircularList();
    list.insertAtBeginning(3);
    list.insertAtBeginning(2);
    list.insertAtBeginning(1);
    list.printList();
}
```

## At the End

Start by creating a new node. Then check to see if the list is empty (`head == null`). If so, set both the head and the tail to the new node. Set the `next` attribute of the new node to point back to itself. Use the `return` keyword to exit the method.

```
void insertAtEnd(int data) {
    Node newNode = new Node(data);

    if(head == null) {
        // List is empty, make the new node the head and tail.
        head = newNode;
        tail = newNode;
        newNode.next = newNode;
        return;
    }

}
```

If the list is not empty, then set the `next` attribute to loop back to the head. Then have the `next` attribute of the old tail point to the new tail. Finally, set the `tail` attribute to the new node.

```

void insertAtEnd(int data) {
    Node newNode = new Node(data);

    if(head == null) {
        // List is empty, make the new node the head and tail.
        head = newNode;
        tail = newNode;
        newNode.next = newNode;
        return;
    }

    newNode.next = head;
    tail.next = newNode; // Point to itself for circularity.
    tail = newNode;
}

```

Update the `main` method so that it uses the `insertAtEnd` method to add the values 1, 2, and 3 to the list. Then print the list.

```

public static void main(String[] args) {
    CircularList list = new CircularList();
    list.insertAtEnd(1);
    list.insertAtEnd(2);
    list.insertAtEnd(3);
    list.printList();
}

```

## At a Specific Position

The `insertAtPos` method takes values for the data of the new node and the position for the insertion. Check to see if the position for insertion is less than 1. If so, print a message to the user about the invalid position, and exit the method with a `return` statement. Next, check to see if the position is 1. If so, call the `insertAtBeginning` method.

```
public void insertAtPos(int data, int pos) {  
    if (pos < 1) {  
        System.out.println("Invalid position.");  
        return;  
    }  
  
    if (pos == 1) {  
        insertAtBeginning(data);  
    }  
}
```

Add an else statement to the conditional and create the current node. We also need a new node and currentPos, which is a counter for the current positon. As long as currentPos is less than the desired insertion position minus 1, set current to the next node and increment currentPos by 1. Check to see if we have traversed the list (current == head). If so, print a message to the user that the position they provided was not valid (i.e. the position goes beyond the length of the list). Exit out of the method with a return statement.

```

public void insertAtPos(int data, int pos) {
    if (pos < 1) {
        System.out.println("Invalid position.");
        return;
    }

    if (pos == 1) {
        insertAtBeginning(data);
    } else {
        Node current = head;
        Node newNode = new Node(data);
        int currentPos = 1;

        // Traverse the list to find the node before the
        // desired position.
        while (currentPos < pos - 1) {
            current = current.next;
            currentPos++;
        }

        if (current == head) {
            System.out.println("Invalid position.");
            return;
        }
    }
}

```

The last part is to insert the node into the position. Set the next attribute of the new node to the next attribute of the current node. Then set the next attribute of the current node to the new node. If the new node points to the head, set the tail to the new node.

```

public void insertAtPos(int data, int pos) {
    if (pos < 1) {
        System.out.println("Invalid position.");
        return;
    }

    if (pos == 1) {
        insertAtBeginning(data);
    } else {
        Node current = head;
        Node newNode = new Node(data);
        int currentPos = 1;

        // Traverse the list to find the node before the
        // desired position.
        while (currentPos < pos - 1) {
            current = current.next;
            currentPos++;
        }

        if (current == head) {
            System.out.println("Invalid position.");
            return;
        }
    }

    // Adjust links to insert the new node at the
    // specified position.
    newNode.next = current.next;
    current.next = newNode;
    if (newNode.next == head) {
        tail = newNode;
    }
}

```

Update the `main` method so that it uses the `insertAtEnd` method to add the values 1, 2, and 3 to the list. Then print the list.

```
public static void main(String[] args) {  
    CircularList list = new CircularList();  
    list.insertAtEnd(1);  
    list.insertAtEnd(2);  
    list.insertAtEnd(3);  
    list.insertAtEnd(4);  
    list.insertAtPos(5, 3);  
    list.printList();  
}
```

challenge

### Try these variations:

- Test the `insertAtPos` method with the valid positions: 1, 2, 3, 4, and 5.
- Test the `insertAtPos` method with the invalid positions: 0 and 6.

# Deletion

## Removing Nodes

### From the Beginning

Deleting from the beginning has three different cases. If the list is empty (`head == null`), do nothing and use a `return` to exit the method. If the list only has one node, set the head and tail to `null` and exit the method with `return`. In all other cases, make the head point to the second node (`head.next`). Then have the tail point to the new head, and update the `prev` pointer of the new head to the tail.

```
public void deleteAtBeginning() {
    if (head == null) return;

    if (head == tail) {
        head = null;
        tail = null;
        return;
    }

    head = head.next;
    tail.next = head;
}
```

Update the `main` method so that the list contains the values 1, 2, 3, and 4. Then remove the first node from the list with the `deleteAtBeginning` method.

```
public static void main(String[] args) {
    CircularList list = new CircularList();
    list.insertAtBeginning(1);
    list.insertAtEnd(2);
    list.insertAtEnd(3);
    list.insertAtEnd(4);
    list.deleteAtBeginning();
    list.printList();
}
```

### From the End

Just as before, we are going to do nothing if the list is empty. If the list only has one node, then set both the head and tail to null and exit the method. Things get a little more complicated in all other cases. We may know the tail of the list, but we cannot easily access it since our list is singly linked. Create a new node called secondToLast and set it to the head. Traverse the list until secondToLast points to the tail. Then have secondToLast point to the head and make secondToLast the tail.

```
public void deleteAtEnd() {
    if (head == null) return;

    if (head == tail) {
        head = null;
        tail = null;
        return;
    }

    Node secondToLast = head;
    while (secondToLast.next != tail) {
        secondToLast = secondToLast.next;
    }

    secondToLast.next = head;
    tail = secondToLast;
}
```

Update the main method so that the list contains the values 1, 2, 3, and 4. Then remove the tail node from the list with the deleteAtEnd method.

```
public static void main(String[] args) {
    CircularList list = new CircularList();
    list.insertAtBeginning(1);
    list.insertAtEnd(2);
    list.insertAtEnd(3);
    list.insertAtEnd(4);
    list.deleteAtEnd();
    list.printList();
}
```

## From a Specific Position

Start by checking for three special cases. If the list is empty, print a message that you cannot remove a node from an empty list and exit the method. If the position is less than 1, print a message that the position is invalid and exit the method. If the position is 1, call the deleteAtBeginning method.

```

void deleteAtPos(int pos) {
    if (head == null) {
        System.out.println("Cannot remove node from an empty
list.");
        return;
    }

    if(pos < 1) {
        System.out.println("Invalid position.");
        return;
    }

    if(pos == 1) {
        deleteAtBeginning();
    }

}

```

Add and else statement to the last conditional. Create the node `current` and set it to the head. Create `currentPos` which is a counter for the current position. As long as `currentPos` is less than the position for deletion minus 1, have `current` move to the next node in the list. Then increment `currentPos` by 1. If `current` is equal to the head, the the position to delete exceeds the length of the list. Print a message to the user. After the loop ends, set the value of `current.next` to `current.next.next`. This skips over the node that is to be removed from the list. Finally, check to see if `current.next` is pointing to the head. If so, set the tail to `current`.

```

void deleteAtPos(int pos) {
    if (head == null) {
        System.out.println("Cannot remove node from an empty
list.");
        return;
    }

    if(pos < 1) {
        System.out.println("Invalid position.");
        return;
    }

    if(pos == 1) {
        deleteAtBeginning();
    } else {
        Node current = head;
        int currentPos = 1;

        // Traverse the list to find the node before the
desired position.
        while (currentPos < pos - 1) {
            current = current.next;
            currentPos++;
        }

        if (current == head) {
            System.out.println("Invalid position.");
            return;
        }
    }

    // Adjust links to insert the new node at the
specified position.
    current.next = current.next.next;
    if (current.next == head) {
        tail = current;
    }
}

```

challenge

### Try these variations:

- Test the `deleteAtPos` method with the valid positions: 1, 2, 3, and 4.
- Test the `deleteAtPos` method with the invalid positions: 0 and 5.



# Traversal & Searching

## Navigating a Circular Linked List

### Traversing

We have already seen how to traverse a list with the `printList` method. However, I would like to return the one of the biggest challenges when working with circular lists, an infinite loop. Find the `printList` method in the IDE and comment out the `break` statement. Running this code will produce a lot (I mean a lot) of output before finally timing out. Pay particular attention when traversing a circular list. You have to manually stop the loop yourself.

```
public void printList() {
    Node current = head;

    while(current != null) {
        System.out.print(current.data + " ");
        current = current.next;

        if(current == head) {
            System.out.println();
            // break;
        }
    }
}
```

info

### Important

Be sure to uncomment the `break` statement in the `printList` method. We do not want any more infinite loops when we run our code.

#### ▼ Traversing a circular list in reverse

If you have a circular list that is doubly linked, you can traverse it in reverse. Instead of setting the `current` node to the head, set it to the tail. You will create an infinite loop just as before. Set `current` to `current.prev` to

move backwards through the list. When `current` is equal to the tail, break out of the loop.

**Important**, the code below is an example of how to traverse a circular list in reverse. However, do not copy and paste it into the IDE. The code snippet assumes both `prev` and `next` pointers. Our code only has the `next` pointer.

```
public void reversePrint() {  
    Node current = tail;  
  
    while(current != null) {  
        System.out.print(current.data + " ");  
        current = current.prev;  
  
        if(current == tail) {  
            System.out.println();  
            break;  
        }  
    }  
}
```

## Searching

To search for a specific node, create the `findNode` method that takes an integer value. The node with this value will be returned. Create the node `current`, and traverse the list. If the value of the current node is equal to the target value, return the current node. Advance to the next node in the list. If you get back to the head, break out of the loop. Breaking out of the loop means the node is not in the list, so return `null`.

```

public Node findNode(int targetValue) {
    Node current = head;

    while(current != null) {
        if (current.data == targetValue) {
            return current;
        }
        current = current.next;

        if(current == head) {
            break;
        }
    }

    return null;
}

```

Now that we can find a specific node, we need to be able to delete it. Create the `deleteNode` method that takes a `Node` object. Start by checking for special cases. If the list is empty (`head` is `null`) or the node does not exist in the list (`del` is `null`), do nothing and exit the method. If you are removing the first node or there only one node in the list (`head` equals `tail`) call the `deleteAtBeginning` method. This method handles both of those cases. If you are deleting the tail node, call the `deleteAtEnd` method.

```

public void deleteNode(Node del) {
    if (head == null || del == null) return;

    if (head == del || head == tail) {
        deleteAtBeginning();
        return;
    }

    if (tail == del) {
        deleteAtEnd();
        return;
    }
}

```

If the node to delete is elsewhere in the list, we are going to have to traverse the list, stopping at the node before the one to be deleted (`del`). Create the node `prevDel` and set it to the `head`. Traverse the list with this node. If the node after `prevDel` is the node to be deleted, break out of the loop. Otherwise advance to the next node. After leaving the loop, have the node before `del` point to the node after `del`.

```

public void deleteNode(Node del) {
    if (head == null || del == null) return;

    if (head == del || head == tail) {
        deleteAtBeginning();
        return;
    }

    if (tail == del) {
        deleteAtEnd();
        return;
    }

    Node prevDel = head;

    while (prevDel != null) {
        if (prevDel.next == del) {
            break;
        }
        prevDel = prevDel.next;
    }

    prevDel.next = del.next;
}

```

### ▼ Checking for the head

You may have noticed that this traversal did not check to see if `prevDel` has gone back to the head of the list. Normally, you would do this to avoid an infinite loop. If the node is not in the list, it will continue to traverse without stopping. However, we know in advance if the node exists or not – if it has the value `null` it is not in the list. We handled this case in the first conditional of the method.

Update the `main` method so the list has four nodes. Find and delete the node with the value of 3. Then print the list.

```
public static void main(String[] args) {
    CircularList list = new CircularList();
    list.insertAtEnd(1);
    list.insertAtEnd(2);
    list.insertAtEnd(3);
    list.insertAtEnd(4);
    Node target = list.findNode(3);
    list.deleteNode(target);
    list.printList();
}
```

challenge

### Try these variations:

- Test the `deleteNode` method with the valid positions: 1, 2, 3, and 4.
- Test the `deleteNode` method with the invalid positions: 0 and 5.

# Analyzing Circular Linked Lists

## Time Complexity Analysis of Operations in Circular Linked Lists

### Insertion

- **At the Beginning** - The time complexity for this operation is  $O(1)$  because we only have to change a few pointers, assuming we maintain both a head and tail pointer.
- **At the End** - This operation has a time complexity of  $O(n)$ . Even if you know the tail pointer, a singly linked list means you have to traverse up to the node to delete. A doubly linked circular list can remove nodes at the end with a time complexity of  $O(1)$ .
- **At a Specific Position** - In the worst-case scenario, this could take  $O(n)$  time, as we may have to traverse the entire list to reach the specific position.

### Deletion

- **From the Beginning** - Deletion from the beginning has a time complexity of  $O(1)$  because it involves just a few pointer updates.
- **From the End** - The time complexity is  $O(1)$  for deletion from the end if a tail pointer is maintained.
- **Specific Position** - Just like insertion, deletion at a specific position could go up to  $O(n)$  in the worst-case scenario.

### Traversal & Searching

- **Traversal** - Traversing the entire list would take  $O(n)$  time. Here, one should be careful to stop the traversal once we reach back to the head to avoid an infinite loop.
- **Searching** - In a circular linked list, searching involves traversing through the list until the item is found or the loop comes back to the head. Therefore, the time complexity is  $O(n)$ .

### Table of Operations

The table below summarizes the time complexities for the various operations covered. Any reductions to time complexity are due to knowing the tail node.

| Operation                    | Singly Linked    | Doubly Linked    |
|------------------------------|------------------|------------------|
| <b>Insertion (Beginning)</b> | $O(1)$           | $O(1)$           |
| <b>Insertion (End)</b>       | $O(n)$ or $O(1)$ | $O(1)$ or $O(1)$ |
| <b>Insertion (Position)</b>  | $O(n)$           | $O(n)$           |
| <b>Deletion (Beginning)</b>  | $O(1)$           | $O(1)$           |
| <b>Deletion (End)</b>        | $O(n)$           | $O(n)$ or $O(1)$ |
| <b>Deletion (Position)</b>   | $O(n)$           | $O(n)$           |
| <b>Traversal</b>             | $O(n)$           | $O(n)$           |
| <b>Searching</b>             | $O(n)$           | $O(n)$           |

While circular linked lists offer some advantages like the ease of performing rotation operations and efficient insertions and deletions when they use the tail pointer and double pointers. However, searching and specific insertions or deletions can require traversing the list, making those operations  $O(n)$  in the worst case.

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

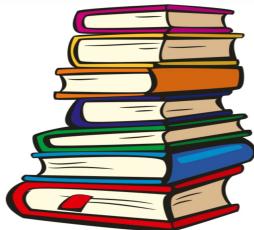
- **Define a stack**
- **Cover Basic operations of stacks**
- **Perform Basic Operations with stacks**

# Stacks

## What are Stacks?

A **stack** is an abstract data type (ADT) that allows all data operations at one end only. At any given time, we can only access the top element of a stack. You cannot directly access elements elsewhere in the stack.

Stacks are also as a linear data structure, but it may be more helpful to think of the stack data structure as a physical stack of books. If you want to add a book to the stack, it goes on top. If you remove a book, you take it from the top. Moreover, the most recently added book will be the first to be removed.



## Basic Operations

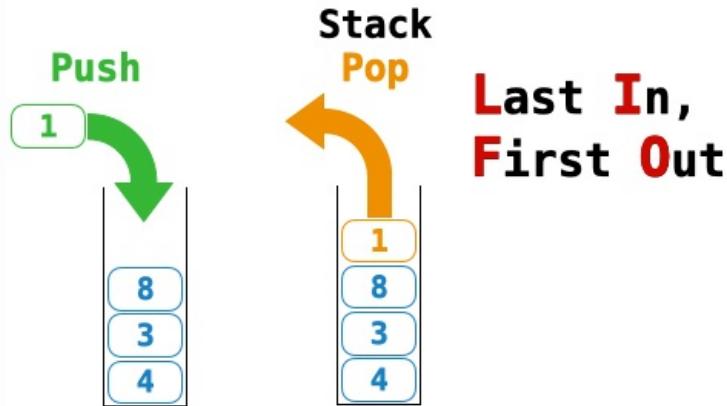
A stack has two fundamental operations: **push** and **pop**. Pushing adds an element (insertion) to the top of the stack, while popping removes (deletion) the top element from the stack. Again, both of these operations only work on the top of the stack. Insertions and deletions cannot happen elsewhere in the stack.

Another common operation used with stacks is **peek**. Peeking involves looking at the top element in the collection while keeping the element on the stack. Unlike pushing and popping, peeking is not a fundamental operation associated with the stack data structure. Stacks must be able to push and pop; peeking is optional.

You may have noticed that traversing is not an operation associated with a stack. Stacks cannot be directly traversed. If you want to interact with elements in the stack, you first must pop them off the data structure.

## LIFO

The order in which elements come off and on a stack is always the same. The last element added is the first one to be deleted. This consistent behavior is referred to as last in, first out (LIFO). LIFO is pretty efficient since we always know which element can be removed and where the next element will be added.



The gif depicts the last in, first out behavior of stacks. A push places an element on the top of the stack. Popping removes the top element from the stack. The text “Last In First Out” appears to the right. The first letter of each word is red. These letters form “LIFO”.

Now that we know what a stack is, we are going to see how to implement one in a couple of different ways.

# Array-Based Stack Implementation

## Representing a Stack with an Array

The stack abstract data type can be implemented using various underlying data structures, with arrays and linked lists being the most commonly used. The choice between these implementations often depends on the specific requirements of the application.

In an array-based stack, a variable, often named `top`, is used to keep track of the index of the most recently added element. This is essential for adhering to the last-in, first-out (LIFO) behavior intrinsic to stacks. The `top` variable allows for quick and efficient push, pop, and peek operations, which are the core functions of a stack.

Arrays are often the go-to data structure for implementing stacks due to their ability to access elements in constant time. However, a limitation of using arrays is their fixed size. One must either define the maximum size of the stack beforehand or use dynamic resizing techniques.

## Key Operations and Their Time Complexities

Here are the essential operations for an array-based stack along with their time complexities:

- **Push:**  $O(1)$
- **Pop:**  $O(1)$
- **Peek:**  $O(1)$
- **IsEmpty:**  $O(1)$
- **IsFull:**  $O(1)$

## Implementation

Create the `ArrayStack` class. It should have three attributes: `maxSize`, `top`, and `stackArray`. Since we are using an array, we need to know the size of the array to be used. This is done with `maxSize`. The `top` attribute represents the top element in the stack. Finally we have `stackArray`, which is the actual array used to store data in the stack.

```

class ArrayStack {
    private int maxSize; // Initialize the maximum size of the stack
    private int top; // Initialize a variable to track the top element
    private int[] stackArray; // Create an array to hold the stack elements

}

```

Create a constructor, which takes the expected size of the stack. Set `maxSize` to the size of the stack. Then use this size to instantiate an array. Since the stack is empty upon instantiation, set `top` to -1.

```

// Constructor to set up the stack size
public ArrayStack(int size) {
    maxSize = size;
    stackArray = new int[maxSize];
    top = -1; // Set the top to -1 as the stack is initially empty
}

```

Arrays are not dynamic, so if we want to push an element to the stack, we first have to see if there is room in the array for the new element. We will cover the `isFull` helper method a bit later. If the array is full, throw an exception that the stack is full. Otherwise, increment `top` and add the value to the array.

```

// Push operation to add a value to the top of the stack
public void push(int value) {
    // Check if the stack is full
    if (isFull()) {
        throw new IllegalStateException("Stack is full");
    }
    // Increment the top and add the value
    stackArray[++top] = value;
}

```

Similarly, we need to check to see if the array is empty before popping an element from the stack. We will cover the `isEmpty` helper method in a bit. Throw an exception if the stack is empty. Otherwise, decrement `top` and return the value in the array at index `top`.

```

// Pop operation to remove the top value from the stack
public int pop() {
    // Check if the stack is empty
    if (isEmpty()) {
        throw new IllegalStateException("Stack is empty");
    }
    // Remove the top value and decrement the top
    return stackArray[top--];
}

```

Peeking at the top element in a stack is very similar to popping an element off the stack. Start by checking to see if the stack is empty. If yes, throw an exception that the stack is empty. Otherwise return the element at index `top` but *do not* decrement `top`.

```

// Peek operation to view the top value without removing it
public int peek() {
    // Check if the stack is empty
    if (isEmpty()) {
        throw new IllegalStateException("Stack is empty");
    }
    // Return the top value without removing it
    return stackArray[top];
}

```

Lastly, we need to create the `isEmpty` and `isFull` helper methods. If `top` is equal to `-1`, then the stack is empty. When the stack is full, then `top` is equal to `maxSize` minus 1. Add the following methods to the class:

```

// Check if the stack is empty
public boolean isEmpty() {
    return (top == -1);
}

// Check if the stack is full
public boolean isFull() {
    return (top == maxSize - 1);
}

```

## ▼ Code

Your code should look like this:

```

class ArrayStack {
    private int maxSize; // Initialize the maximum size of the stack
}

```

```
private int top; // Initialize a variable to track the top
                  element
private int[] stackArray; // Create an array to hold the
                           stack elements

// Constructor to set up the stack size
public ArrayStack(int size) {
    maxSize = size;
    stackArray = new int[maxSize];
    top = -1; // Set the top to -1 as the stack is initially
              empty
}

// Push operation to add a value to the top of the stack
public void push(int value) {
    // Check if the stack is full
    if (isFull()) {
        throw new IllegalStateException("Stack is full");
    }
    // Increment the top and add the value
    stackArray[++top] = value;
}

// Pop operation to remove the top value from the stack
public int pop() {
    // Check if the stack is empty
    if (isEmpty()) {
        throw new IllegalStateException("Stack is empty");
    }
    // Remove the top value and decrement the top
    return stackArray[top--];
}

// Peek operation to view the top value without removing it
public int peek() {
    // Check if the stack is empty
    if (isEmpty()) {
        throw new IllegalStateException("Stack is empty");
    }
    // Return the top value without removing it
    return stackArray[top];
}

// Check if the stack is empty
public boolean isEmpty() {
    return (top == -1);
}

// Check if the stack is full
```

```
public boolean isFull() {
    return (top == maxSize - 1);
}
```

In the `main` method, instantiate an array-based stack with a size of five elements. Add the integers 1, 2, and 3 to the stack. Peek at the top value and print it. Then pop the top element off the stack and print it. Finally, peek at the new top element and print it.

```
public static void main(String[] args) {
    ArrayStack myStack = new ArrayStack(5);

    myStack.push(1);
    myStack.push(2);
    myStack.push(3);

    System.out.println("Peek: " + myStack.peek());
    System.out.println("Pop: " + myStack.pop());
    System.out.println("Peek: " + myStack.peek());
}
```

You should see the following output:

```
tex -hide-clipboard Peek: 3 Pop: 3 Peek: 2
```

Creating a stack with array involves having the `top` variable that represents the index of the element considered to be at the top of the stack. Pushing and popping require you to increment or decrement `top`, which peeking does not.

# List-Based Stack Implementation

## Representing a Stack with a Linked List

After delving into array-based stacks, let's shift our focus to linked list-based implementations. Linked lists offer more flexibility than arrays as they can dynamically grow and shrink, avoiding the size limitation issue often found in array-based stacks. The head node of the linked list will serve as the top of the stack.

## Key Operations and Their Time Complexities

Here are the essential operations for an list-based stack along with their time complexities:

- **Push:**  $O(1)$
- **Pop:**  $O(1)$
- **Peek:**  $O(1)$
- **IsEmpty:**  $O(1)$

## Implementation

Since this implementation is using a singly linked list, we first need to create the `Node` class. Each node stores data and contains a pointer. The pointer should be `null` when a creating a `Node` object.

```
class Node {  
    int data;  
    Node next;  
  
    public Node(int data) {  
        this.data = data;  
        this.next = null;  
    }  
}
```

Next, create the `LinkedStack` class. The `head` attribute keeps track of the head node in the list. This attribute represents the top of the stack. It should be `null` to indicate that the list is empty.

```

class LinkedStack {
    // Initialize the head (top) of the linked list
    private Node head;

    // Constructor to initialize an empty stack
    public LinkedStack() {
        head = null;
    }

}

```

Linked lists are dynamic, so we do not need to ask if the list is full. We can add as many nodes as we want. When pushing, create a new Node object and insert it at the beginning of the list. Be sure to update the head attribute.

```

// Push operation
public void push(int value) {
    Node newNode = new Node(value);
    newNode.next = head;
    head = newNode;
}

```

We do, however, need to check to see if the list is empty before popping an element from the stack. Ask if the stack is empty and throw an exception if it is. We also need to return an integer value, so -1 will represent an empty list. Otherwise, return the data stored in the head node. Then update the head to point to the next node in the list.

```

// Pop operation
public int pop() {
    if (isEmpty()) {
        throw new IllegalStateException("Stack is empty");
    }
    int poppedValue = head.data;
    head = head.next;
    return poppedValue;
}

```

Again, peeking is similar to popping. First check to see if the list is empty and throw an exception if so. Otherwise, return the value stored in the head node. *Do not* set the value of head to the next node in the list.

```
// Peek operation
public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Stack is empty");
    }
    return head.data;
}
```

Finally, we need to create the `isEmpty` helper method. The stack is empty if the head has the value `null`. Return a boolean value representing if the list is empty or not.

```
// Check if stack is empty
public boolean isEmpty() {
    return head == null;
}
```

## ▼ Code

Your code should look like this:

```
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class LinkedStack {
    // Initialize the head (top) of the linked list
    private Node head;

    // Constructor to initialize an empty stack
    public LinkedStack() {
        head = null;
    }

    // Push operation
    public void push(int value) {
        Node newNode = new Node(value);
        newNode.next = head;
        head = newNode;
    }
}
```

```

// Pop operation
public int pop() {
    if (isEmpty()) {
        throw new IllegalStateException("Stack is empty");
    }
    int poppedValue = head.data;
    head = head.next;
    return poppedValue;
}

// Peek operation
public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Stack is empty");
    }
    return head.data;
}

// Check if stack is empty
public boolean isEmpty() {
    return head == null;
}

```

In the `main` method, instantiate an list-based stack. We do not need to indicate its size since linked lists are dynamic. Add the integers 1, 2, and 3 to the stack. Peek at the top value and print it. Then pop the top element off the stack and print it. Finally, peek at the new top element and print it.

```

public static void main(String[] args) {
    LinkedStack myStack = new LinkedStack();

    myStack.push(1);
    myStack.push(2);
    myStack.push(3);

    System.out.println("Peek: " + myStack.peek());
    System.out.println("Pop: " + myStack.pop());
    System.out.println("Peek: " + myStack.peek());
}

```

You should see the following output:

```
tex -hide-clipboard Peek: 3 Pop: 3 Peek: 2
```

In the linked list-based stack, each node contains the data and a reference to the next node. The push, pop, and peek operations are carried out at the head to maintain a time complexity of  $O(1)$ .

# Array-Based vs Linked List-Based Stacks

## Comparing Stack Implementations

Having explored both array-based and linked list-based stacks, it's essential to compare them to understand their strengths and weaknesses better. This will help you make a more informed choice depending on your specific needs.

We will compare the two implementations with regards to memory allocation, time complexity, ease of implementation, and use cases.

### Memory Allocation

Array-based stacks require that memory be allocated beforehand. It is possible to create a new array of a different size and move over all of the elements if need be. However, these additional operations can be costly.

Linked list-based stacks do not need to worry about memory allocation beforehand as linked lists are dynamic. However, nodes in a linked list require a bit of extra memory for pointers.

### Time Complexity

Array-based stacks have a set of three core operations: push, pop, and peek. Due to the direct access nature of arrays, these operations have a time complexity of  $O(1)$ . This performance assumes that array does not need to be resized. Dynamically altering the array will cause additional overhead.

Linked list-based stacks have the same set of core operations. Because stacks only allow operations at the top of the stack, that means all operations happen at the head of the list. Because this position is always known, push, pop, and peek also have a time complexity of  $O(1)$ .

### Ease of Implementation

Array-based stacks are a bit easier to implement and use as they use a built-in data structure. However, the fixed-size nature of arrays means you need to check if the stack is full before pushing.

Linked list-based stacks are slightly more complex due to creating the classes for the nodes and the linked list. In addition, you have to update the head attribute so that it always points to the top of the stack. Linked lists are

dynamic, so you can push to the stack without checking to see if it is full.

## Use Cases

Array-based stacks make the most sense when the size of the stack is known and does not change. This implementation also lends itself nicely when you want a quick and simple solution.

Linked list-based stacks make the most sense if the stack needs to frequently change size. Using a linked list for a stack offers more flexibility if you do not mind managing pointers.

In summary, array-based stacks are simpler but can have limitations with dynamic sizing. Linked list-based stacks are more flexible but a bit more complex to implement. Neither implementation is more performant than the other. The choice between the two will largely depend on your application's specific requirements for memory use and computational efficiency.

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- **Define a queue and compare it to a stack**
  - **Implement a queue with an array, a linked list, and a dynamic array**
  - **Compare and contrast the different queue implementations**
-

# Queue

## What is a Queue?

A **queue** is a linear data structure that follows a particular order in which the operations are performed. Unlike stacks, where the order is last in, first out (LIFO), the order for queues is **first in, first out (FIFO)**. This means that the item that is inserted first is the first one to be removed. Imagine a real-life queue of people waiting in line for a bus; the first person to arrive is the first one to board.



The image depicts 20 cartoon people standing in a line.

Like stacks, queues are also an abstract data type. The ADT specifies which operations must be included (see below), but it does not state how the underlying data structure should be implemented. Whether you use an array, a linked list, or any other data structure to implement a queue, is entirely up to you and the requirements of your application.

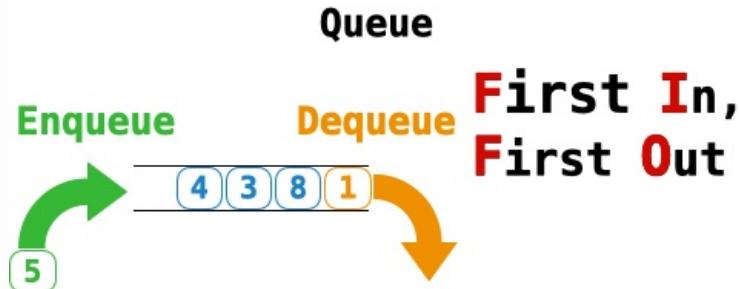
## Queue Operations

A Queue supports a basic set of operations that make it unique:

- **Enqueue:** This operation adds an item to the back of the queue.
- **Dequeue:** This operation removes an item from the front of the queue.
- **Peek/Front:** This operation returns the item at the front of the queue without removing it.
- **IsEmpty:** Checks if the queue is empty.
- **IsFull:** Checks if the queue is full (applicable mainly to array-based implementations).

## FIFO?

The FIFO paradigm is particularly useful in scenarios where fairness or order of processing is critical. Examples include job scheduling, order processing systems, and even data packet management in computer networks.



The gif depicts the first in, first out behavior of queues.  
Enqueueing places an element on the end of the queue.  
Dequeuing removes the element from the front of the queue. The text “First In, First Out” appears to the right. The first letter of each word is red. These letters form “FIFO”.

The rest of this assignment covers the different implementations of a queue, and compares and contrasts these implementations.

# Array-Based Implementation of Queue in Java

## Representing a Queue with an Array

Now that we understand what a queue is and the operations it supports, let's implement one using an array. The choice of an array for this exercise offers simplicity and performs well for the operations we need.

In particular, we are going to create a circular queue. This means that once you fill up the array, you can "loop back around" and write over the elements that have already been dequeued.

## Key Operations and their Time Complexities

Here's a breakdown of the key operations and their time complexities for an array-based queue:

- **Enqueue:**  $O(1)$
- **Dequeue:**  $O(1)$
- **Peek/Front:**  $O(1)$
- **IsEmpty:**  $O(1)$
- **IsFull:**  $O(1)$

## Implementation

For our array-based Queue, we'll use a simple array with two pointers to keep track of the front and rear of the Queue. Since arrays are static, we also need to set the length of the array.

Create the `ArrayQueue` class with the attributes, `capacity`, `front`, `rear`, `size`, and `array`. `capacity` represents how many elements can be stored in the queue, `size` represents how many elements are currently in the queue, `front` points to the element at the front of the queue, `rear` points to the element at the rear of the queue, and `array` is the array used to store elements. In the constructor, set `capacity` to the size passed as a parameter, create an array with the specified size, set `front` to `0`, and set `rear` to `-1` since the array is empty.

```

class ArrayQueue {
    private int capacity; // Maximum capacity of the queue
    private int front; // Front index for dequeue operation
    private int rear; // Rear index for enqueue operation
    private int size; // Current size of the queue
    private int[] array;

    public ArrayQueue(int c) {
        capacity = c;
        array = new int[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }
}

```

To add elements to a queue, we are going to create the `enqueue` method, which takes an integer as a parameter. Because arrays are static, we need to check if the array is full. If so, throw an exception with a message that the queue is full. Then increment `rear` by 1 and find the remainder when dividing by `capacity`. This is the “wrap around” feature mentioned above. When `rear + 1` is equal to `capacity`, using the modulo operator will return `0`, and we are back at the beginning of the array. We also need to add the new data to the array with `rear` being the index. Lastly, increment `size` by one.

```

// Enqueue: Add an element to the rear of the queue
public void enqueue(int data) {
    if(isFull()) {
        throw new IllegalStateException("Queue is full");
    }

    rear = (rear + 1) % capacity;
    array[rear] = data;
    size++;
}

```

Create the `dequeue` method to remove an element from the queue. Start by checking if the queue is empty. If so, throw an exception with the message that the queue is empty. Otherwise, use a temporary variable to store the value in the front element. Increment `front` by 1 and find the remainder when dividing by `capacity`. This will wrap the `front` pointer back around to index `0`. Then decrease the size of the queue, and return the temporary variable.

```

// Dequeue: Remove an element from the front of the queue
public int dequeue() {
    if(isEmpty()) {
        throw new IllegalStateException("Queue is empty");
    }

    int data = array[front];
    front = (front + 1) % capacity;
    size--;
    return data;
}

```

To peek at the first element, first check that it is full. If so, throw an exception with a message that the queue is empty. Otherwise, return the value stored in the `front` element of the queue. *Do not* change the `front` or `size` attributes.

```

// Peek: View the front element without removing it
public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Queue is empty");
    }

    return array[front];
}

```

Lastly, we need to create the `isEmpty` and `isFull` helper methods. A queue is empty when the `size` attribute is `0`, and a queue is full when `size` is equal to the length of the array.

```

// Check if queue is empty
public boolean isEmpty() {
    return size == 0;
}

// Check if queue is full
public boolean isFull() {
    return size == array.length; // Circular queue condition
                                for full
}

```

## ▼ Code

Your code should look like this:

```
class ArrayQueue {
```

```

private int capacity; // Maximum capacity of the queue
private int front; // Front index for dequeue operation
private int rear; // Rear index for enqueue operation
private int size; // Current size of the queue
private int[] array;

public ArrayQueue(int c) {
    capacity = c;
    array = new int[capacity];
    front = 0;
    rear = -1;
    size = 0;
}

// Enqueue: Add an element to the rear of the queue
public void enqueue(int data) {
    if(isFull()) {
        throw new IllegalStateException("Queue is full");
    }

    rear = (rear + 1) % capacity;
    array[rear] = data;
    size++;
}

// Dequeue: Remove an element from the front of the queue
public int dequeue() {
    if(isEmpty()) {
        throw new IllegalStateException("Queue is empty");
    }

    int data = array[front];
    front = (front + 1) % capacity;
    size--;
    return data;
}

// Peek: View the front element without removing it
public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Queue is empty");
    }

    return array[front];
}

// Check if queue is empty
public boolean isEmpty() {
}

```

```

    return size == 0;
}

// Check if queue is full
public boolean isFull() {
    return size == array.length; // Circular queue condition
                                for full
}
}

```

In the `main` method, instantiate an array-based queue with a size of five elements. Add the integers 1, 2, and 3 to the stack. Peek at the first value and print it. Then dequeue the first element from the queue and print it. Finally, peek at the new first element and print it.

```

public static void main(String[] args) {
    ArrayQueue myQueue = new ArrayQueue(5);
    myQueue.enqueue(1);
    myQueue.enqueue(2);
    myQueue.enqueue(3);
    System.out.println("Peek: " + myQueue.peek());
    System.out.println("Dequeue: " + myQueue.dequeue());
    System.out.println("Peek: " + myQueue.peek());
}

```

You should see the following output:

```
tex -hide-clipboard Peek: 1 Dequeue: 1 Peek: 2
```

# Linked List-Based Implementation of Queue

## Representing a Queue with a Linked List

After discussing the array-based implementation of a queue, let's move on to using a linked list for the same purpose. This approach allows for dynamic resizing, offering a bit more flexibility compared to an array-based structure.

## Key Operations and their Time Complexities

Here's a breakdown of the key operations and their time complexities for a linked list-based Queue:

- **Enqueue:**  $O(1)$
- **Dequeue:**  $O(1)$
- **Peek/Front:**  $O(1)$
- **IsEmpty:**  $O(1)$

## Implementation

In a linked list-based queue, we'll use a singly linked list with two pointers, `front` and `rear`, to keep track of the beginning and the end of the queue, respectively.

Start by creating the `Node` class. It has an attribute for the data stored and a pointer to the next node. In the constructor, set the `data` attribute to the value passed as a parameter, and set the pointer to `null`.

```
// Define Node class for linked list elements
class Node {
    int data;
    Node next;

    Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

Then create the `LinkedListQueue` class. It has two attributes, `front` and `rear` which are of type `Node`. In the constructor, set both of these attributes to `null`.

```
class LinkedListQueue {  
    private Node front; // Front of the queue  
    private Node rear; // Rear of the queue  
  
    // Initialize an empty queue  
    public LinkedListQueue() {  
        front = null;  
        rear = null;  
    }  
}
```

To enqueue an element, first check to see if the queue is empty. If yes, set both `front` and `rear` to the new node. If not, have the `rear` node point to the new node, and then update `rear` to the new node.

```
// Enqueue: Add an element to the end of the queue  
public void enqueue(int value) {  
    Node newNode = new Node(value);  
    if (isEmpty()) {  
        front = newNode;  
        rear = newNode;  
    } else {  
        rear.next = newNode;  
        rear = newNode;  
    }  
}
```

To dequeue an element, first check to see if the queue is empty. If yes, throw an exception that the queue is empty. If the queue is not empty, create a temporary variable and store the data from the front element in it. Then have the new front become the next node in the list. It is possible that dequeuing a node will create an empty list. Check to see if the list is empty one more time. If yes, set the `rear` attribute to `null`. Lastly, return the value stored in the `temp` variable.

```

// Dequeue: Remove an element from the front of the queue
public int dequeue() {
    if (isEmpty()) {
        throw new IllegalStateException("Queue is empty");
    }
    int temp = front.data;
    front = front.next;
    if (isEmpty()) {
        rear = null; // If queue becomes empty
    }
    return temp;
}

```

To peek at the first element, first check to see if the queue is empty. If yes, throw an exception that the queue is empty. Otherwise return the value stored in `front.data`. *Do not* change the `front` or `rear` attributes.

```

// Peek: View the front element without removing it
public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Queue is empty");
    }
    return front.data;
}

```

The last thing to do is to define the `isEmpty` helper method. A queue is empty when the `front` node is equal to `null`.

```

// Check if the queue is empty
public boolean isEmpty() {
    return front == null;
}

```

## ▼ Code

Your code should look like this:

```

// Define Node class for linked list elements
class Node {
    int data;
    Node next;

    Node(int data) {
        this.data = data;
        this.next = null;
    }
}

```

```

        }
    }

class LinkedListQueue {
    private Node front; // Front of the queue
    private Node rear; // Rear of the queue

    // Initialize an empty queue
    public LinkedListQueue() {
        front = null;
        rear = null;
    }

    // Enqueue: Add an element to the end of the queue
    public void enqueue(int value) {
        Node newNode = new Node(value);
        if (isEmpty()) {
            front = newNode;
            rear = newNode;
        } else {
            rear.next = newNode;
            rear = newNode;
        }
    }

    // Dequeue: Remove an element from the front of the queue
    public int dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        int temp = front.data;
        front = front.next;
        if (isEmpty()) {
            rear = null; // If queue becomes empty
        }
        return temp;
    }

    // Peek: View the front element without removing it
    public int peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        return front.data;
    }

    // Check if the queue is empty
    public boolean isEmpty() {

```

```
    return front == null;
}
}
```

In the `main` method, instantiate an array-based queue with a size of five elements. Add the integers 1, 2, and 3 to the stack. Peek at the first value and print it. Then dequeue the first element from the queue and print it. Finally, peek at the new first element and print it.

```
public static void main(String[] args) {
    LinkedListQueue myQueue = new LinkedListQueue();
    myQueue.enqueue(1);
    myQueue.enqueue(2);
    myQueue.enqueue(3);
    System.out.println("Peek: " + myQueue.peek());
    System.out.println("Dequeue: " + myQueue.dequeue());
    System.out.println("Peek: " + myQueue.peek());
}
```

You should see the following output:

```
tex -hide-clipboard Peek: 1 Dequeue: 1 Peek: 2
```

In this example, we add three elements (1, 2, 3) to the Queue, then peek at the front element (1), and dequeue it. This confirms that our linked list-based Queue implementation works as expected.

# Dynamic Array-Based Implementation

## Representing a Queue with a Dynamic Array

On the previous pages, we looked at array and linked list-based implementations of a queue. Now, let's explore another approach that utilizes dynamic arrays (an `ArrayList` in Java) for implementation. This is particularly useful when the size of the queue can change dynamically and you don't want to manage pointers yourself.

## Key Operations and their Time Complexities

Using dynamic arrays like `ArrayList` can be convenient for queue implementation. Here are the time complexities for key operations in this case:

- **Enqueue:**  $O(1)$  (amortized)
- **Dequeue:**  $O(1)$
- **Peek/Front:**  $O(1)$
- **IsEmpty:**  $O(1)$

### ▼ What does “amortized” mean?

Amortized time complexity is an averaged time per operation over a sequence of operations, giving a more comprehensive understanding of performance. For example, in a dynamic array-based queue, most enqueue operations will be  $O(1)$ , but occasionally the operation will be more expensive due to the resizing of the array. When you average out the cost of these occasional expensive operations over a large number of total operations, you get a constant time complexity on average, often noted as  $O(1)$  “amortized”.

## Implementation

Start by importing the `ArrayList` class. Then create the class `DynamicArrayQueue` which has two attributes, an `ArrayList` and an integer that keeps track of the front of the queue. In the constructor, instantiate an `ArrayList` of integers and set the front index pointer to `0`.

```

import java.util.ArrayList;

class DynamicArrayQueue {
    private ArrayList<Integer> queue;
    private int frontIndex; // Keeps track of the front index

    // Initialize an empty queue
    public DynamicArrayQueue() {
        queue = new ArrayList<>();
        frontIndex = 0;
    }
}

```

To enqueue an element, call the add method on the ArrayList. Unlike a linked list, you do not have to manage any pointers while still retaining the flexibility of a dynamic list.

```

// Enqueue: Add an element to the end of the queue
public void enqueue(int value) {
    queue.add(value); // ArrayList handles resizing
}

```

Create the dequeue method to remove an element from the queue. First, ask if the queue is empty. If so, throw an exception and print a message to the user. Otherwise, store the value from the first element in a temporary variable. Then increment the front index by 1. Finally, return the temporary variable.

```

// Dequeue: Remove an element from the front of the queue
public int dequeue() {
    if (isEmpty()) {
        throw new IllegalStateException("Queue is empty");
    }
    int temp = queue.get(frontIndex);
    frontIndex++;
    return temp;
}

```

Again, peeking at an element is very similar to dequeuing an element. Start by checking if the queue is empty. If so, throw a message and print a message to the user. Otherwise, return the value of the element at the front index. *Do not* increment the index.

```
// Peek: View the front element without removing it
public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Queue is empty");
    }
    return queue.get(frontIndex);
}
```

Finally, create the `isEmpty` helper method. The queue is empty when the front index is greater than or equal to the size of the queue. This may seem a bit odd at first. Remember, every time you dequeue an element, you increment the front index. Because the front index will only get larger, that is why we check to see if it is larger than the size of the queue.

```
// Check if the queue is empty
public boolean isEmpty() {
    return frontIndex >= queue.size();
}
```

### ▼ Code

```

import java.util.ArrayList;

class DynamicArrayQueue {
    private ArrayList<Integer> queue;
    private int frontIndex; // Keeps track of the front index

    // Initialize an empty queue
    public DynamicArrayQueue() {
        queue = new ArrayList<>();
        frontIndex = 0;
    }

    // Enqueue: Add an element to the end of the queue
    public void enqueue(int value) {
        queue.add(value); // ArrayList handles resizing
    }

    // Dequeue: Remove an element from the front of the queue
    public int dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        int temp = queue.get(frontIndex);
        frontIndex++;
        return temp;
    }

    // Peek: View the front element without removing it
    public int peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        return queue.get(frontIndex);
    }

    // Check if the queue is empty
    public boolean isEmpty() {
        return frontIndex >= queue.size();
    }
}

```

In the `main` method, instantiate an array-based queue with a size of five elements. Add the integers 1, 2, and 3 to the stack. Peek at the first value and print it. Then dequeue the first element from the queue and print it. Finally, peek at the new first element and print it.

```
public static void main(String[] args) {
    DynamicArrayQueue myQueue = new DynamicArrayQueue();
    myQueue.enqueue(1);
    myQueue.enqueue(2);
    myQueue.enqueue(3);
    System.out.println("Peek: " + myQueue.peek());
    System.out.println("Dequeue: " + myQueue.dequeue());
    System.out.println("Peek: " + myQueue.peek());
}
```

You should see the following output:

```
tex -hide-clipboard Peek: 1 Dequeue: 1 Peek: 2
```

# Comparing Queue Implementations

Choosing the right data structure for implementing a queue is crucial for performance optimization. Below, we evaluate the three implementations discussed: array-based, linked list-based, and dynamic array-based queues, focusing on the pros and cons of each.

## Array-Based Queue

### Pros

- **Constant Time Operations:** Enqueue and Dequeue operations are  $O(1)$  if no resizing is needed.
- **Memory Efficiency:** Uses a contiguous block of memory, making better use of caches.
- **Simplicity:** Easier to implement and understand.

### Cons

- **Fixed Size:** The size needs to be defined upfront, which may lead to wasted space or the need for resizing.
- **Resizing Overhead:** If the array needs to be resized, it can be expensive.

## Linked List-Based Queue

### Pros

- **Dynamic Size:** The size can change dynamically, so there's no wasted space.
- **Constant Time Operations:** Both enqueue and dequeue operations are  $O(1)$ .

### Cons

- **Memory Overhead:** Each node requires extra memory for the next pointer.
- **Cache Inefficiency:** Nodes might be scattered across the memory, leading to cache inefficiency.

## Dynamic Array-Based Queue (ArrayList)

### Pros

- **Dynamic Resizing:** Automatically resizes, offering a balance between array and linked list implementations.
- **Amortized Constant Time:** Enqueue operation is  $O(1)$  on average.

### Cons

- **Amortized Time:** The time complexity is amortized, meaning occasionally operations may take longer.
- **Memory Overhead:** Internal resizing operations may temporarily require additional memory.

## Comparison Table

| Criteria          | Array-Based | Linked List-Based | Dynamic Array-Based |
|-------------------|-------------|-------------------|---------------------|
| Enqueue           | $O(1)$      | $O(1)$            | $O(1)$ (amortized)  |
| Dequeue           | $O(1)$      | $O(1)$            | $O(1)$              |
| Peek/Front        | $O(1)$      | $O(1)$            | $O(1)$              |
| Memory Efficiency | High        | Moderate          | Moderate            |
| Cache Efficiency  | High        | Low               | Moderate            |
| Flexibility       | Low         | High              | High                |

By understanding the strengths and weaknesses of each implementation, you can choose the one that best fits your specific needs.

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- **Define a priority queue**
- **Differentiate a priority queue from a traditional queue**
- **Implement a priority queue using an array, ArrayList, and linked list**

# Priority Queue

## What is a Priority Queue?

A **priority queue** is an abstract data type (ADT) similar to a regular queue or stack but with an added dimension: **priority**. Unlike standard queues where the first in, first out (FIFO) principle applies, elements in a priority queue are dequeued based on their priority; higher priority elements are removed first. This data structure is particularly useful in algorithms and systems that require a more flexible and nuanced approach to data processing, such as task scheduling, data compression algorithms, and network traffic management.

Priorities can be represented in a variety of ways. For the purposes of this assignment, we are going to keep our priority simple. Our priority queue is going to work with integers. The larger the integer, the greater the priority. Essentially, our examples will store values in descending order.

## Basic Operations

Here are the main operations associated with priority queues:

- **Enqueue:** Insert an element to the queue with an associated priority.
- **Dequeue:** Delete the highest-priority element from the queue. If two elements have the same priority, the queue may resort to FIFO.
- **Peek:** Views the highest-priority element without removing it from the queue.
- **IsEmpty:** Checks if the priority queue is empty.
- **Size:** Returns the number of elements in the priority queue.

## Implementations

Priority Queues can be implemented using various underlying data structures, the most common of which are:

- **Arrays or Linked Lists:** Simpler but generally less efficient,  $O(n)$  time complexity for insertion.
- **Binary Heaps:** More complex but provide better efficiency, offering  $O(\log n)$  time complexity for both insertion and removal.
- **Fibonacci Heaps:** Advanced data structures that offer better amortized time complexities.

In this assignment, we will explore different implementations of priority queues, starting with the most straightforward array-based approach, we will touch on heaps in a later assignment.

# Array-Based Implementation

## Representing a Priority Queue with an Array

In its most straightforward form, a priority queue can be implemented using a simple array. Although this approach is less efficient than using more advanced data structures like heaps, it serves as a good introduction to the concept. In this section, we'll explore how to build a priority queue using an array as the underlying data structure.

## Key Operations and their Time Complexities

Here are the key operations and their respective time complexities for an array-based Priority Queue:

- **Enqueue:**  $O(n)$
- **Dequeue:**  $O(n)$
- **Peek:**  $O(1)$
- **IsEmpty:**  $O(1)$
- **Size:**  $O(1)$

Note that both the Enqueue and Dequeue operations have a time complexity of  $O(n)$  because in the worst-case scenario we may need to traverse the entire array to find the correct position for insertion or deletion.

## Implementation

Create the `ArrayPriorityQueue` class which has three attributes: `capacity`, `array`, and `size`. `capacity` refers to the number of elements the queue can hold, `array` is the underlying data structure, and `size` refers to how many elements are currently in the priority queue. The constructor takes a value for the capacity and initializes the three attributes.

```

class ArrayPriorityQueue {
    private int capacity;
    private int[] array;
    private int size;

    // Constructor
    public ArrayPriorityQueue(int capacity) {
        this.capacity = capacity;
        array = new int[this.capacity];
        size = 0;
    }

}

```

When enqueueing an element, let's start by checking for two special cases. If the queue is full, throw an exception with a message to the user. If the queue is empty, set the element at index `size` to the given value. Then increment `size` by 1 and use `return` to exit the method.

```

// Enqueue operation with priority handling
public void enqueue(int value) {
    if (isFull()) {
        throw new IllegalStateException("Priority queue is full");
    }

    if (isEmpty()) {
        array[size++] = value;
        return;
    }

}

```

If we need to add an element and these two special cases do not apply, we need to place the element in its proper position. The larger the value, the greater the priority. Start by creating the variable `i`. Use `i` in a for loop that iterates backwards over the array. If the given value is less than the value at index `i`, shift the value at index `i` to the right. If the given value is greater than or equal the value at index `i`, break out of the loop. Then set the element at `i + 1` to the given value and increment `size` by 1.

```

// Enqueue operation with priority handling
public void enqueue(int value) {
    if (isFull()) {
        throw new IllegalStateException("Priority queue is full");
    }

    if (isEmpty()) {
        array[size++] = value;
        return;
    }

    int i;
    for (i = size - 1; i >= 0; i--) {
        if (value < array[i]) {
            array[i + 1] = array[i];
        } else {
            break;
        }
    }
    array[i + 1] = value;
    size++;
}

```

When dequeuing an element, first check to see if the queue is empty. If so, throw an exception with a message to the user. If there are elements in the queue, decrement the value `size` by 1 and then return the element at this new index.

```

// Dequeue operation
public int dequeue() {
    if (isEmpty()) {
        throw new IllegalStateException("Priority queue is empty");
    }

    return array[--size];
}

```

To peek at the highest priority element, return the value stored at `size - 1`. However, *do not* alter the value of `size`.

```

// Peek operation
public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Priority queue is empty");
    }

    return array[size - 1];
}

```

Checking the size of the priority queue is just a accessor method that returns the `size` attribute.

```

// Check the size of the Priority Queue
public int size() {
    return size;
}

```

Lastly, a priority queue is empty when `size` is equal to `0`. Have the `isEmpty` method return the result from this boolean expression.

```

// Check if the Priority Queue is empty
public boolean isEmpty() {
    return size == 0;
}

// Check if the priority queue is full
public boolean isFull() {
    return size == capacity;
}

```

## ▼ Code

Your code should look something like this:

```

class ArrayPriorityQueue {
    private int capacity;
    private int[] array;
    private int size;

    // Constructor
    public ArrayPriorityQueue(int capacity) {
        this.capacity = capacity;
        array = new int[this.capacity];
        size = 0;
    }
}

```

```

// Enqueue operation with priority handling
public void enqueue(int value) {
    if (isFull()) {
        throw new IllegalStateException("Priority queue is full");
    }

    if (isEmpty()) {
        array[size++] = value;
        return;
    }

    int i;
    for (i = size - 1; i >= 0; i--) {
        if (value < array[i]) {
            array[i + 1] = array[i];
        } else {
            break;
        }
    }
    array[i + 1] = value;
    size++;
}

// Dequeue operation
public int dequeue() {
    if (isEmpty()) {
        throw new IllegalStateException("Priority queue is empty");
    }

    return array[--size];
}

// Peek operation
public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Priority queue is empty");
    }

    return array[size - 1];
}

// Check the size of the Priority Queue
public int size() {
    return size;
}

```

```

// Check if the Priority Queue is empty
public boolean isEmpty() {
    return size == 0;
}

// Check if the priority queue is full
public boolean isFull() {
    return size == capacity;
}

```

Create the `main` method which instantiates an array-based priority queue with a capacity of 5. Add the values 3, 2, and 4. Use `peek` to display the value with the highest priority, print the size, dequeue the element with the highest priority, and print the size once again.

```

public static void main(String[] args) {
    ArrayPriorityQueue myPQ = new ArrayPriorityQueue(5);
    myPQ.enqueue(3);
    myPQ.enqueue(2);
    myPQ.enqueue(4);
    System.out.println("Highest priority element: " +
    myPQ.peek());
    System.out.println("Priority queue size: " +
    myPQ.size());
    System.out.println("Dequeue operation result: " +
    myPQ.dequeue());
    System.out.println("Priority queue size: " +
    myPQ.size());
}

```

You should see the following output:

```
tex -hide-clipboard Highest priority element: 4 Priority queue size:  
3 Dequeue operation result: 4 Priority queue size: 2
```

challenge

## Try this variation

Update the enqueue method so that the conditional checks to see if the given value is greater than element at index i. Then run the program again.

```
if (value > array[i]) {  
    array[i + 1] = array[i];  
} else {
```

### ▼ What is happening?

If we change the boolean operator, we are now shifting elements in the array when the given value is greater than the current element. That is, smaller values get moved to the front of the queue. The queues goes from being arranged from largest to smallest, to now being smallest to largest. Elements with a smaller value are given priority over element with a larger value.

In the next section, we'll see how to implement a priority queue with a linked list.

# Linked List-Based Implementation

## Representing a Priority Queue with a Linked List

Another straightforward way to implement a priority queue is by using a Linked List as the underlying data structure. This method may offer some advantages over the array-based approach, particularly in terms of dynamic sizing. In this section, we will delve into the specifics of a Linked List-based Priority Queue.

## Key Operations and their Time Complexity

Here are the key operations and their respective time complexities for a Linked List-based Priority Queue:

- **Enqueue:**  $O(n)$
- **Dequeue:**  $O(1)$
- **Peek:**  $O(1)$
- **IsEmpty:**  $O(1)$
- **Size:**  $O(1)$

As with the array-based approach, the enqueue operation may require traversing the entire list to find the correct position for the new item, leading to a time complexity of  $O(n)$ . However, dequeue and peek are more efficient with  $O(1)$  since we maintain a reference to the head of the list.

## Implementation

Since we are using a linked list, we need a Node object. This is going to be a singly linked list, so our node has a next pointer and a data attribute to store information.

```

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

```

Now we can create the `LinkedListPriorityQueue` class. It has two attributes, `head` (keeps track of the first node) and `size` (keeps track of the size of the list). In the constructor, set `head` to `null` and `size` to `0`.

```

class LinkedListPriorityQueue {
    private Node head;
    private int size;

    // Constructor
    public LinkedListPriorityQueue() {
        head = null;
        size = 0;
    }

}

```

When we enqueue an element, we either put the element directly at the head or iterate through the list to find the correct location. A node can go to the head if the either the list is empty or the given value is greater than the value already stored at the head.

```

// Enqueue operation with priority handling
public void enqueue(int value) {
    Node newNode = new Node(value);
    if (isEmpty() || value > head.data) {
        newNode.next = head;
        head = newNode;
    }

}

```

Otherwise we need to iterate over the linked list until either we reach the end or the value of the next node is less than or equal to the value of the new node. Be sure to properly manage the pointers. The current node

should point to the new node, and the new node should point to the node after the current node. Then increment `size` by 1.

```
// Enqueue operation with priority handling
public void enqueue(int value) {
    Node newNode = new Node(value);
    if (isEmpty() || value > head.data) {
        newNode.next = head;
        head = newNode;
    } else {
        Node current = head;
        while (current.next != null && current.next.data <=
value) {
            current = current.next;
        }
        newNode.next = current.next;
        current.next = newNode;
    }
    size++;
}
```

To dequeue an element, start by asking if the list is empty. If yes, throw an exception with a message for the user. Then create a temporary variable that stores the value in the head node. Change the new head to the next node in the list. After that, decrement `size` by 1. Finally, return the temporary variable.

```
// Dequeue operation
public int dequeue() {
    if (isEmpty()) {
        throw new IllegalStateException("Priority queue is
empty");
    }
    int value = head.data;
    head = head.next;
    size--;
    return value;
}
```

When peeking at the element with the highest priority, return the data found at the head of the list. *Do not* change any pointers.

```
// Peek operation
public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Priority queue is empty");
    }
    return head.data;
}
```

Checking the size of the priority queue is just a accessor method that returns the `size` attribute.

```
// Check the size of the Priority Queue
public int size() {
    return size;
}
```

Lastly, we need the `isEmpty` helper method. A linked list is empty when the head of the list is `null`.

```
// Check if the Priority Queue is empty
public boolean isEmpty() {
    return head == null;
}
```

## ▼ Code

Your code should look something like this:

```
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class LinkedListPriorityQueue {
    private Node head;
    private int size;

    // Constructor
    public LinkedListPriorityQueue() {
        head = null;
    }

    // Peek operation
    public int peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Priority queue is empty");
        }
        return head.data;
    }

    // Check if the Priority Queue is empty
    public boolean isEmpty() {
        return head == null;
    }

    // Size accessor
    public int size() {
        return size;
    }

    // Check the size of the Priority Queue
    public int size() {
        return size;
    }
}
```

```

        size = 0;
    }

// Enqueue operation with priority handling
public void enqueue(int value) {
    Node newNode = new Node(value);
    if (isEmpty() || value > head.data) {
        newNode.next = head;
        head = newNode;
    } else {
        Node current = head;
        while (current.next != null && current.next.data <=
value) {
            current = current.next;
        }
        newNode.next = current.next;
        current.next = newNode;
    }
    size++;
}

// Dequeue operation
public int dequeue() {
    if (isEmpty()) {
        throw new IllegalStateException("Priority queue is
empty");
    }
    int value = head.data;
    head = head.next;
    size--;
    return value;
}

// Peek operation
public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Priority queue is
empty");
    }
    return head.data;
}

// Check the size of the Priority Queue
public int size() {
    return size;
}

// Check if the Priority Queue is empty
public boolean isEmpty() {

```

```
    return head == null;
}
}
```

Create the `main` method which instantiates an linked list-based priority queue. Add the values 3, 2, and 4. Use `peek` to display the value with the highest priority, print the size, dequeue the element with the highest priority, and print the size once again.

```
public static void main(String[] args) {
    LinkedListPriorityQueue myPQ = new
        LinkedListPriorityQueue();
    myPQ.enqueue(3);
    myPQ.enqueue(2);
    myPQ.enqueue(4);
    System.out.println("Highest priority element: " +
        myPQ.peek());
    System.out.println("Priority queue size: " +
        myPQ.size());
    System.out.println("Dequeue operation result: " +
        myPQ.dequeue());
    System.out.println("Priority queue size: " +
        myPQ.size());
}
```

You should see the following output:

```
tex -hide-clipboard Highest priority element: 4 Priority queue size:
3 Dequeue operation result: 4 Priority queue size: 2
```

# ArrayList-Based Implementation

## Representing a Priority Queue with an ArrayList

ArrayLists in Java are dynamic arrays that resize themselves automatically when elements are added or removed. They provide the benefits of arrays while removing the need for manual size management. In this section, we explore implementing a Priority Queue using an ArrayList.

## Key Operations and their Time Complexities

Here are the key operations and their respective time complexities for an ArrayList-based Priority Queue:

- **Enqueue:**  $O(n)$
- **Dequeue:**  $O(1)$
- **Peek:**  $O(1)$
- **IsEmpty:**  $O(1)$
- **Size:**  $O(1)$

The time complexity of the enqueue operation is  $O(n)$  because, in the worst-case scenario, you may need to traverse the entire list to find the correct position for the new item. On the other hand, dequeue and peek operations are generally  $O(1)$  if we keep track of the front and rear of the queue.

## Implementation

Start by importing the `ArrayList` class. Then create the `ArrayListPriorityQueue` class, which only has an `ArrayList` as an attribute. This is where all of the data for the priority queue will be stored. Instantiate the `ArrayList` in the constructor.

```

import java.util.ArrayList;

class ArrayListPriorityQueue {
    private ArrayList<Integer> arrayList;

    // Constructor
    public ArrayListPriorityQueue() {
        arrayList = new ArrayList<>();
    }

}

```

Since we are working with an `ArrayList`, we are going to use methods associated with this class for our priority queue operations. Start by creating the `enqueue` method that takes an integer value. We also need the variable `indexToInsert` which will be the index to use when inserting the given value. Iterate over the `ArrayList` with a for loop. Create the `current` variable and set its value to the element in the loop. If the given value is greater than the current value, set `indexToInsert` to the loop variable and break out of the loop. If `value` is not greater than `current`, increment `indexToInsert`. After the loop ends, use the `add` method and `indexToInsert` to insert the given value at the correct position.

```

// Enqueue operation with priority handling
public void enqueue(int value) {
    int indexToInsert = 0;
    // Find the correct index to insert the item based on
    // priority
    for (int i = 0; i < arrayList.size(); i++) {
        int current = arrayList.get(i);
        // Compare the new item with the current item for
        // priority
        if (value > current) {
            indexToInsert = i;
            break;
        }
        // If the new item has higher priority, increment
        // the index
        indexToInsert++;
    }
    // Insert the item at the determined index
    arrayList.add(indexToInsert, value);
}

```

When dequeuing an element, first check to see if the priority queue is empty. If so, thrown an exception with a message to the user. Otherwise, use the `remove` method to remove the first element in the priority queue.

```
// Dequeue operation
public int dequeue() {
    if (isEmpty()) {
        throw new IllegalStateException("Priority queue is empty");
    }
    return arrayList.remove(0);
}
```

To peek at the highest priority element, first check to see if the priority queue is empty. If yes, thrown an exception with a message to the user. Otherwise, use the get method to return the first element in the priority queue.

```
// Peek operation
public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Priority queue is empty");
    }
    return arrayList.get(0);
}
```

Checking the size of the priority queue is done with the ArrayList method size. This will return the length of the arrayList attribute (the priority queue).

```
// Check the size of the Priority Queue
public int size() {
    return arrayList.size();
}
```

Lastly, create the isEmpty helper method. ArrayLists have a method named isEmpty which is what we will call on the priority queue. This method will return a boolean value.

```
// Check if the Priority Queue is empty
public boolean isEmpty() {
    return arrayList.isEmpty();
}
```

## ▼ Code

Your code should look something like this:

```
import java.util.ArrayList;

class ArrayListPriorityQueue {
    private ArrayList<Integer> arrayList;

    // Constructor
    public ArrayListPriorityQueue() {
        arrayList = new ArrayList<>();
    }

    // Enqueue operation with priority handling
    public void enqueue(int value) {
        int indexToInsert = 0;
        // Find the correct index to insert the item based on
        // priority
        for (int i = 0; i < arrayList.size(); i++) {
            int current = arrayList.get(i);
            // Compare the new item with the current item for
            // priority
            if (value > current) {
                indexToInsert = i;
                break;
            }
            // If the new item has higher priority, increment
            // the index
            indexToInsert++;
        }
        // Insert the item at the determined index
        arrayList.add(indexToInsert, value);
    }

    // Dequeue operation
    public int dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("Priority queue is
empty");
        }
        return arrayList.remove(0);
    }

    // Peek operation
    public int peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Priority queue is
empty");
        }
        return arrayList.get(0);
    }
}
```

```

// Check the size of the Priority Queue
public int size() {
    return arrayList.size();
}

// Check if the Priority Queue is empty
public boolean isEmpty() {
    return arrayList.isEmpty();
}
}

```

Create the `main` method which instantiates an `ArrayList`-based priority queue. Add the values 3, 2, and 4. Use `peek` to display the value with the highest priority, print the size, dequeue the element with the highest priority, and print the size once again.

```

public static void main(String[] args) {
    ArrayListPriorityQueue myPQ = new
        ArrayListPriorityQueue();
    myPQ.enqueue(3);
    myPQ.enqueue(2);
    myPQ.enqueue(4);
    System.out.println("Highest priority element: " +
        myPQ.peek());
    System.out.println("Priority queue size: " +
        myPQ.size());
    System.out.println("Dequeue operation result: " +
        myPQ.dequeue());
    System.out.println("Priority queue size: " +
        myPQ.size());
}

```

You should see the following output:

```

tex -hide-clipboard Highest priority element: 4 Priority queue size:
3 Dequeue operation result: 4 Priority queue size: 2

```

In the following section, we will compare the performance of different implementations of priority queues to help you understand the trade-offs involved.

# Analyzing Priority Queue Implementations

## Comparing Different Implementations of Priority Queue

We have so far looked at implementing a priority queue with array-based, linked list-based, and ArrayList-based implementations individually. Let's now compare the array-based priority queue with other common implementations: linked list and dynamic array.

| Implementation Type | Enqueue | Dequeue | Peek   | IsEmpty | Space Complexity      |
|---------------------|---------|---------|--------|---------|-----------------------|
| Array-Based         | $O(n)$  | $O(n)$  | $O(1)$ | $O(1)$  | $O(n)$                |
| Linked List         | $O(n)$  | $O(n)$  | $O(1)$ | $O(1)$  | $O(n)$                |
| Dynamic Array       | $O(n)$  | $O(n)$  | $O(1)$ | $O(1)$  | $O(1)$<br>(amortized) |

### ▼ What does “amortized” mean?

The term “amortized” is used to describe an average time per operation over a sequence. In the context of a dynamic array, although individual operations can be expensive due to resizing, when averaged over a large number of operations, the cost becomes constant.

## Array-Based Priority Queue

### Advantages:

- Simple to implement
- Constant time complexity for peek and isEmpty operations

### Disadvantages:

- Fixed size limits the number of elements that can be stored
- Both enqueue and dequeue operations are  $O(n)$ , making it inefficient for large datasets

## Linked List-Based Priority Queue

### Advantages:

- Dynamic size

- Constant time complexity for peek and isEmpty operations

**Disadvantages:**

- $O(n)$  time complexity for enqueue and dequeue operations
- Extra memory overhead due to the pointers

## **Dynamic Array-Based Priority Queue**

**Advantages:**

- Dynamic size, can grow as needed
- Constant time complexity for peek and isEmpty operations

**Disadvantages:**

- $O(n)$  time complexity for enqueue and dequeue operations
- Amortized constant space complexity

# **Formative Assessment 1**

## **Formative Assessment 2**

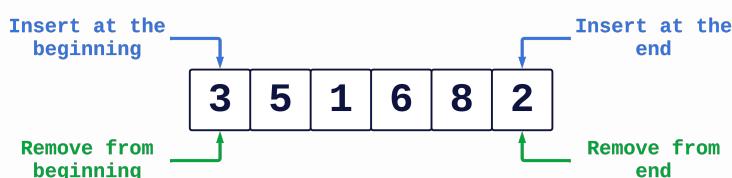
# **Learning Objectives**

**Learners will be able to...**

- **Define a deque**
- **Differentiate a deque from stacks, queues, and priority queues**
- **Implement a deque**

# Deque

A **deque** (pronounced as ‘deck’), stands for **double-ended queue**. Unlike a regular queue, where elements can be added to the back and removed from the front, a deque allows adding and removing elements from both ends. Deques are a linear data structure that provide the functionality of both a stack and a queue. Deques can be implemented using various underlying data structures such as arrays, linked lists, or even a hybrid approach.



The image depicts a deque with the elements 3, 5, 1, 6, 8, and 2. There is a blue arrow pointing to the 3 that says “Insert at the beginning”. There is also a green arrow pointing to the 3 that says “Remove from beginning”. There is a blue arrow pointing to the 2 that says, “Insert at the end”. There is a green arrow pointing to the 2 that says “Remove from end”.

## Core Operations

Deques provide the following key operations:

- **addFront(element)**: Adds an element to the front of the deque.
- **addRear(element)**: Adds an element to the rear of the deque.
- **removeFront()**: Removes the front element from the deque.
- **removeRear()**: Removes the rear element from the deque.
- **peekFront()**: Returns the front element without removing it.
- **peekRear()**: Returns the rear element without removing it.
- **isEmpty()**: Checks if the deque is empty.
- **size()**: Returns the number of elements in the deque.

## When to Use a Deque?

Deques are particularly useful in scenarios where you need dynamic and flexible data storage with efficient operations at both ends. For example, they can be used in certain algorithms, like sliding window problems, palindrome checking, etc.

We will delve deeper into the various implementations of deques. We will also cover the time complexities associated with these operations in different types of implementations, namely:

- Array-based deques
- Linked List-based deques

# Array-based Implementation

## Representing a Deque with an Array

We will explore the array-based implementation of a deque. The array-based approach is straightforward and offers constant-time access to elements. However, one of the limitations is that the array size is fixed, meaning you'll have to resize the array if the deque grows beyond its initial capacity.

This implementation of a deque will be circular in nature. That means, if you increment a pointer in the last element or decrement a pointer at index 0, the pointer will “wrap back around” to the other end of the array. This is why you will use modulo (%) when calculating new values for `front` and `rear`.

## Key Operations and Their Time Complexities

The array-based implementation of deque is efficient for small datasets but may require resizing for larger datasets. Here are the key operations for an array-based deque and their respective time complexities:

- **addFront(element)**:  $O(1)$  or  $O(n)$  if resizing is needed
- **addRear(element)**:  $O(1)$  or  $O(n)$  if resizing is needed
- **removeFront()**:  $O(1)$
- **removeRear()**:  $O(1)$
- **peekFront()**:  $O(1)$
- **peekRear()**:  $O(1)$
- **isEmpty()**:  $O(1)$
- **isFull()**:  $O(1)$
- **size()**:  $O(1)$

## Implementation

Start by creating the `ArrayDeque` class. It has four attributes: `capacity` which is the size of the array, `size` which represents the number of elements in the deque, `front` which represents the front of the deque, `rear` which represents the rear of the deque, and `dequeArray` which is the actual array used to store elements. In the constructor, pass it the size of the array. Set `maxSize` to the parameter, instantiate `dequeArray` with the given size, set `front` to `0`, and set `rear` to `-1`.

```

class ArrayDeque {
    private int capacity;
    private int size;
    private int front;
    private int rear;
    private int[] dequeArray;

    // Constructor
    public ArrayDeque(int c) {
        capacity = c;
        size = 0;
        front = 0;
        rear = -1;
        dequeArray = new int[capacity];
    }

}

```

When adding an element to the front of the deque, start by checking to see if it is full. If so, throw an exception with a message for the user. Since we want the deque to be circular, we need to subtract 1 from front and add capacity. Then take this result and perform modulo with capacity. Set the element at index front to the given value and increment size by 1.

```

// Add element to the front
public void addFront(int value) {
    if (isFull()) {
        throw new IllegalStateException("Deque is full");
    }

    front = (front - 1 + capacity) % capacity;
    dequeArray[front] = value;
    size++;
}

```

When adding an element to the rear of the deque, check to see if the deque is full. If so, throw an exception with a message for the user. Calculate a new value for rear by adding 1 and taking the modulo with capacity. Set the element at index rear to the given value and increment size by 1.

```

// Add element to the rear
public void addRear(int value) {
    if (isFull()) {
        throw new IllegalStateException("Deque is full");
    }

    rear = (rear + 1) % capacity;
    dequeArray[rear] = value;
    size++;
}

```

To remove an element from the front of the deque, first check to see if it is empty. If so, throw an exception with a message for the user. Then create a temporary variable with the value of the element at index `front`. We are going to update `front` by adding 1 and taking the modulo with `capacity`. Decrement `size` by 1 and return the temporary variable.

```

// Remove element from the front
public int removeFront() {
    if (isEmpty()) {
        throw new IllegalStateException("Deque is empty");
    }

    int temp = dequeArray[front];
    front = (front + 1) % capacity;
    size--;
    return temp;
}

```

The `removeRear` method will remove an element from the rear of a deque. First check to see if the deque is empty. If so, throw an error with a message for the user. Then create a temporary variable that stores the value at index `rear`. Subtract 1 from `rear` and take the modulo with `capacity`. Decrement `size` by 1 and return the temporary variable.

```

// Remove element from the rear
public int removeRear() {
    if (isEmpty()) {
        throw new IllegalStateException("Deque is empty");
    }

    int temp = dequeArray[rear];
    rear = (rear - 1) % capacity;
    size--;
    return temp;
}

```

To peek at either front or rear of the deque, first check to see if it is empty. If so, throw an exception with a message for the user. Then return the element at index `front` (peek at the front) or index `rear` (peek at the rear). *Do not* increment or decrement the pointers.

```

// Peek at the front element
public int peekFront() {
    if (isEmpty()) {
        throw new IllegalStateException("Deque is empty");
    }

    return dequeArray[front];
}

// Peek at the rear element
public int peekRear() {
    if (isEmpty()) {
        throw new IllegalStateException("Deque is empty");
    }

    return dequeArray[rear];
}

```

The `ArrayDeque` class needs a few helper methods. The two most commonly used are `isEmpty` and `isFull`. Both of these method return a boolean value. To check if the deque is empty, ask if its size (not capacity) is `0`. A deque is full when the size of the data structure is equal to the capacity of the array.

```
// Check if the deque is empty
public boolean isEmpty() {
    return (size == 0);
}

public boolean isFull() {
    return (size == capacity);
}
```

The last helper method is an accessor method that returns the value of the `size` attribute.

```
public int getSize() {
    return size;
}
```

## ▼ Code

Your code should look something like this:

```
class ArrayDeque {
    private int capacity;
    private int size;
    private int front;
    private int rear;
    private int[] dequeArray;

    // Constructor
    public ArrayDeque(int c) {
        capacity = c;
        size = 0;
        front = 0;
        rear = -1;
        dequeArray = new int[capacity];
    }

    // Add element to the front
    public void addFront(int value) {
        if (isFull()) {
            throw new IllegalStateException("Deque is full");
        }

        front = (front - 1 + capacity) % capacity;
        dequeArray[front] = value;
        size++;
    }
}
```

```

// Add element to the rear
public void addRear(int value) {
    if (isFull()) {
        throw new IllegalStateException("Deque is full");
    }

    rear = (rear + 1) % capacity;
    dequeArray[rear] = value;
    size++;
}

// Remove element from the front
public int removeFront() {
    if (isEmpty()) {
        throw new IllegalStateException("Deque is empty");
    }

    int temp = dequeArray[front];
    front = (front + 1) % capacity;
    size--;
    return temp;
}

// Remove element from the rear
public int removeRear() {
    if (isEmpty()) {
        throw new IllegalStateException("Deque is empty");
    }

    int temp = dequeArray[rear];
    rear = (rear - 1) % capacity;
    size--;
    return temp;
}

// Peek at the front element
public int peekFront() {
    if (isEmpty()) {
        throw new IllegalStateException("Deque is empty");
    }

    return dequeArray[front];
}

// Peek at the rear element
public int peekRear() {
    if (isEmpty()) {

```

```
        throw new IllegalStateException("Deque is empty");
    }

    return dequeArray[rear];
}

// Check if the deque is empty
public boolean isEmpty() {
    return (size == 0);
}

public boolean isFull() {
    return (size == capacity);
}

public int getSize() {
    return size;
}
}
```

Create the `main` method which instantiates an array-based deque with a capacity of 5. Add the elements 1 and 2. Then peek at the front element. Add elements 3 and 4 and peek at the rear element. Remove elements at the front and rear of the deque. Then ask if the deque is empty, and finally print the size of the deque.

```
public static void main(String[] args) {
    ArrayDeque myDeque = new ArrayDeque(5);

    myDeque.addFront(1);
    myDeque.addFront(2);
    System.out.println("Front element: " +
        myDeque.peekFront());

    myDeque.addRear(3);
    myDeque.addRear(4);
    System.out.println("Rear element: " +
        myDeque.peekRear());

    System.out.println("Removed from front: " +
        myDeque.removeFront());
    System.out.println("Removed from rear: " +
        myDeque.removeRear());

    System.out.println("Is deque empty? " +
        myDeque.isEmpty());
    System.out.println("Size of deque: " +
        myDeque.getSize());
}
```

You should see the following output:

```
tex -hide-clipboard Front element: 2 Rear element: 4 Removed from
front: 2 Removed from rear: 4 Is deque empty? false Size of deque: 2
```

# Deque Using Doubly Linked List

## Representing a Deque with a Doubly Linked List

Now lets discuss how to implement a deque using a doubly linked list. The advantages of using a doubly linked list over an array are that the doubly linked list can grow or shrink dynamically and it allows for efficient insertions and deletions at both ends.

## Key Operations and Their Time Complexities

The efficiencies of a doubly linked list are most noticeable with adding to the front and rear of the deque. Here are the key operations for a doubly linked list-based deque and their respective time complexities:

- **addFront(element)**:  $O(1)$
- **addRear(element)**:  $O(1)$
- **removeFront()**:  $O(1)$
- **removeRear()**:  $O(1)$
- **peekFront()**:  $O(1)$
- **peekRear()**:  $O(1)$
- **isEmpty()**:  $O(1)$
- **size()**:  $O(1)$

## Implementation

We first need to define the `Node` class, which has an attribute for the data stored as well as pointers for the previous and next nodes. In the constructor, set the data to the parameter and set both pointers to `null`.

```

class Node {
    int data;
    Node next;
    Node prev;

    public Node(int data) {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}

```

Next we need to make the `DequeLinkedList` class. The class has three attributes; `head` keeps track of the head node, `tail` keeps track of the tail node, and `size` keeps track of the size of the list. We can use Java's default constructor to generate the default values of `null` for both `head` and `tail` and `0` for `size`.

```

class DequeLinkedList {
    private Node head;
    private Node tail;
    private int size;

}

```

To add an element to the front of the deque, create a new node with the given data. If the linked list is empty, set both `head` and `tail` to the new node. Otherwise, have the `next` pointer in the new node point to the old head. The `prev` pointer in the old head should point to the new node. Then, have `head` become the new node. Lastly, increment `size` by 1.

```

public void addFront(int data) {
    Node newNode = new Node(data);

    if (isEmpty()) {
        head = newNode;
        tail = newNode;
    } else {
        newNode.next = head;
        head.prev = newNode;
        head = newNode;
    }

    size++;
}

```

Adding an element to the rear is a similar task. If the list is empty, both the head and tail should point to the new node. Otherwise, the prev pointer of the new node should point to the old tail. The next pointer in the old tail should point to the new node. Then have the new node become the tail. Lastly, increment size by 1.

```
public void addRear(int data) {
    Node newNode = new Node(data);

    if (isEmpty()) {
        head = newNode;
        tail = newNode;
    } else {
        newNode.prev = tail;
        tail.next = newNode;
        tail = newNode;
    }

    size++;
}
```

Removing an element from the front of a deque, first check to see if the list is empty. If so, throw an exception with a message for the user. Then create a temporary node with the information of the old head. Have the head attribute become the next node in the list. If the new head is null, set the tail to null since the list is now empty. Otherwise, have head.prev become null. Lastly, decrement size by 1 and return the data stored in the temporary variable.

```
public int removeFront() {
    if (isEmpty()) {
        throw new IllegalStateException("Deque is empty");
    }

    Node temp = head;
    head = head.next;
    if (head == null) {
        tail = null;
    } else {
        head.prev = null;
    }
    size--;
    return temp.data;
}
```

Create the `removeRear` method to remove a node from the rear of the deque. Start by checking if the deque is empty. If so, throw an exception with a message to the user. Then create a temporary variable that has the information stored in the old tail. Set the new tail to the previous node in the list. If the new tail is `null` set the head to `null` as well since the list is now empty. Otherwise, set `tail.next` to `null`. Lastly decrement `size` by 1 and return the data stored in the temporary variable.

```
public int removeRear() {
    if (isEmpty()) {
        throw new IllegalStateException("Deque is empty");
    }

    Node temp = tail;
    tail = tail.prev;
    if (tail == null) {
        head = null;
    } else {
        tail.next = null;
    }
    size--;
    return temp.data;
}
```

To peek at the front and rear nodes, create the `peekFront` and `peekRear` methods. The `peekFront` method returns the data stored in the head, while the `peekRear` method returns the data stored in the tail node. *Do not* alter the `next` and `prev` pointers in these methods.

```
public int peekFront() {
    return head.data;
}

public int peekRear() {
    return tail.data;
}
```

Lastly, we have two helper methods. The `isEmpty` method returns if `head` is equal to `null`, which describes an empty linked list. The `getSize` method is an accessor method that returns the `size` attribute.

```
public boolean isEmpty() {
    return head == null;
}

public int getSize() {
    return size;
}
```

## ▼ Code

Your code should look like this:

```
class Node {
    int data;
    Node next;
    Node prev;

    public Node(int data) {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}

class DequeLinkedList {
    private Node head;
    private Node tail;
    private int size;

    public void addFront(int data) {
        Node newNode = new Node(data);

        if (isEmpty()) {
            head = newNode;
            tail = newNode;
        } else {
            newNode.next = head;
            head.prev = newNode;
            head = newNode;
        }

        size++;
    }

    public void addRear(int data) {
        Node newNode = new Node(data);
```

```

        if (isEmpty()) {
            head = newNode;
            tail = newNode;
        } else {
            newNode.prev = tail;
            tail.next = newNode;
            tail = newNode;
        }

        size++;
    }

public int removeFront() {
    if (isEmpty()) {
        throw new IllegalStateException("Deque is empty");
    }

    Node temp = head;
    head = head.next;
    if (head == null) {
        tail = null;
    } else {
        head.prev = null;
    }
    size--;
    return temp.data;
}

public int removeRear() {
    if (isEmpty()) {
        throw new IllegalStateException("Deque is empty");
    }

    Node temp = tail;
    tail = tail.prev;
    if (tail == null) {
        head = null;
    } else {
        tail.next = null;
    }
    size--;
    return temp.data;
}

public int peekFront() {
    return head.data;
}

```

```

public int peekRear() {
    return tail.data;
}

public boolean isEmpty() {
    return head == null;
}

public int getSize() {
    return size;
}

}

```

Create the `main` method which instantiates a doubly linked list-based deque. Add the nodes 1 and 2. Then peek at the front node. Add nodes 3 and 4 and peek at the rear node. Remove nodes at the front and rear of the deque. Then ask if the deque is empty, and finally print the size of the deque.

```

public static void main(String[] args) {
    DequeLinkedList myDeque = new DequeLinkedList();

    myDeque.addFront(1);
    myDeque.addFront(2);
    System.out.println("Front element: " +
myDeque.peekFront());

    myDeque.addRear(3);
    myDeque.addRear(4);
    System.out.println("Rear element: " +
myDeque.peekRear());

    System.out.println("Removed from front: " +
myDeque.removeFront());
    System.out.println("Removed from rear: " +
myDeque.removeRear());

    System.out.println("Is deque empty? " +
myDeque.isEmpty());
    System.out.println("Size of deque: " +
myDeque.getSize());
}

```

You should see the following output:

```

tex -hide-clipboard Front element: 2 Rear element: 4 Removed from
front: 2 Removed from rear: 4 Is deque empty? false Size of deque: 2

```

# Analysis of Deque Implementations

## Analyzing the Different Deque Implementations

We have seen how a deque can be implemented as either an array or a doubly linked list. The former gives you increased flexibility while the latter offers simplicity. Let's see how the two compare regarding time and space complexity.

### Time Complexity Analysis

The table below summarizes the time complexity for common operations. Array-based deques could have a complexity of  $O(n)$  in instances where the array needs to be resized. As long as the array stays the same size, these operations have a constant time complexity. Since a linked list is dynamic, there is no `isFull` method associated with this implementation.

| Operation         | Array-Based      | Doubly Linked List-Based |
|-------------------|------------------|--------------------------|
| Add to Front      | $O(1)$ or $O(n)$ | $O(1)$                   |
| Add to Rear       | $O(1)$ or $O(n)$ | $O(1)$                   |
| Remove from Front | $O(1)$           | $O(1)$                   |
| Remove from Rear  | $O(1)$           | $O(1)$                   |
| Peek Front        | $O(1)$           | $O(1)$                   |
| Peek Rear         | $O(1)$           | $O(1)$                   |
| Is Empty          | $O(1)$           | $O(1)$                   |
| Is Full           | $O(1)$           | —                        |

### Space Complexity

Implementing a deque with either an array or a linked list does not change the space complexity. Both of these implementations will have as many elements or nodes as there is data to store. So the space complexity is linear in relation to the number of elements in the deque.

| Array-Based | Doubly Linked List-Based |
|-------------|--------------------------|
| $O(n)$      | $O(n)$                   |

## **Advantages and Drawbacks**

The time and space complexities are fairly similar between the two underlying data structures. That does not mean that they are interchangeable. There are a few important differences between array-based and linked list-based deques.

---

|                    |                                 |
|--------------------|---------------------------------|
| <b>Array-Based</b> | <b>Doubly Linked List-Based</b> |
|--------------------|---------------------------------|

| **Advantages** |

- All operations are done in constant time.
- Does not require manipulating pointers.

|

- All operations are done in constant time.
- The structure is dynamic and does not need to be resized.

| **Disadvantages** |

- The structure is static and requires additional overhead to resize.

|

- Uses extra memory for prev and next pointers for each node.

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- **Define the Tree ADT**
- **Define commonly used vocabulary regarding trees**
- **Implement and print a tree**

# Introduction to Trees ADT

## Non-Linear Data Structures

Data structures are pivotal, providing foundational means of managing and organizing data. We know that we categorize these structures into linear and nonlinear types based on their organization and data storage.

Linear data structures have their elements arranged in a sequential manner, where each element has a unique predecessor and successor except the first and last elements. Examples of linear data structures are Arrays, Linked Lists, Stacks, and Queues.

```
tex -hide-clipboard 7 - 6 - 8 - 9 - 0
```

Contrasting with the linear models, non-linear data structures do not follow a sequential path. They are organized hierarchically, meaning elements in these structures possess parent-child relationships. Trees and graphs (introduced in a later module) are primary examples of non-linear data structures.

```
``tex -hide-clipboard
```

```
5
/
3   \
/ \   / \
1   4   7   9
```

## Introduction to Trees

A **tree** is a crucial non-linear data structure that emulates a tree structure with a set of linked **nodes**. The topmost node is the **root**, and the elements attached to it are its **children**, forming a hierarchy. Each child node can have its own children, known as sub-trees.

info

## Trees Vocabulary

- **Root:** The top node in a tree.
- **Leaf:** A node with no children.
- **Node:** A node with at least one child.
- **Edge:** The link between two nodes.
- **Height:** The length of the longest path to a leaf.
- **Depth:** The length of the path to its root.

## Tree 5 Node

The gif depicts a tree with the nodes (starting at the root) 5, 3, 8, 1, 4, 7, and 9. As the gif advances, it labels the tree, root, nodes, leaf nodes, parents and children, depth, and height.

Trees are beneficial when it comes to representing hierarchical structures, such as the file system on a computer. They also provide efficient insertion and search operations, making them suitable for various applications like databases and computer graphics. In future assignments, we will see how the non-linear (hierarchical) structure gives trees efficiencies in operations that are just not possible with linear data structures. Trees offer flexibility and versatile representations of data relationships.

# Tree Vocabulary

## Tree Terminology and Properties

Before we proceed to the in-depth discussion and implementation of trees, it is crucial to understand the essential terminologies and properties associated with trees.

### Terminology

Use the interactive image to the left to help you better understand the terminology surrounding trees. Mouse over the buttons to see a representation of each term:

- **Node** - A **node** is the fundamental building block of a tree, holding the data and providing the connection to other nodes.
- **Edge** - An **edge** is a link between two nodes, representing the relationship between them.
- **Root** - The **root** is the topmost node of a tree, having no parent. It is the node from where the tree originates.
- **Leaf Node** - A **leaf node** is a node with no children, located at the periphery of the tree.
- **Internal Node** - Any node, excluding the root, with at least one child is an **internal node**.
- **Siblings** - Nodes that share the same parent are called **siblings**.
- **Height of a Node** - The **height** of a node is the number of edges on the longest path from the node to a leaf.
- **Depth of a Node** - The **depth** of a node is the number of edges from the tree's root to the node.
- **Level** - In a tree, the root is at **level 0**. Its children are at level 1. The children of the children are at level 2, etc.
- **Degree** - The **degree** of a node is the total number of children (or branches) of that node.

### Types of Trees

Several types of trees exist, each with its unique characteristics and applications. We will cover each of these trees in future assignments:

- **Binary Tree** - Each node has at most two children, referred to as the left child and the right child.
- **Binary Search Tree** - A binary tree where the left child is less than the parent, and the right child is greater than the parent.
- **AVL Tree** - A self-balancing binary search tree.

- **B-Tree** - A balanced tree used in databases to store large amounts of data.

## Tree Properties

Understanding the properties of trees is essential for analyzing and implementing them efficiently. Here are a few critical properties of trees:

- **The Maximum Number of Nodes on Level  $l$**   
 $[Maximum\ Nodes = 2^{(l-1)}]$
- **The Maximum Number of Nodes in a Tree of Height  $h$**   
 $[Maximum\ Nodes = 2^h - 1]$
- **For a Binary Tree with  $L$  Leaves:**  
 $[Internal\ Nodes = L - 1]$
- **Height of a Binary Tree with  $n$  Nodes:**  
 $[Minimum\ Height = \lceil \log_2(n + 1) \rceil]$

# Representing a Tree

A tree can be represented using nodes where each node contains data and references to its children. Here, we will be demonstrating a simple tree representation using a user-defined class. Further, we will implement a method to print out the tree in a readable format. For simplicity, we will consider a binary tree in this example.

## Creating the Node

Similar to a doubly linked list, we are going to create a node with two pointers. This node, however, does not point to the previous and next nodes in a linear data structure. Instead, the references point to the child nodes in a hierarchical structure. The data attribute holds an integer, while the left and right pointers are of type `TreeNode`. In the constructor, give the data attribute a value and set left and right to `null`.

```
class TreeNode {
    int data; // data of the node
    TreeNode left; // reference to the left child
    TreeNode right; // reference to the right child

    // Constructor to initialize the node with data and null
    // children
    public TreeNode(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}
```

## Building the Tree

Once the `TreeNode` class is defined, you can start building the tree by creating nodes and linking them together. Create the `BinaryTree` class that has a `root` attribute. In the constructor, assign `root` a new `TreeNode` instance.

```

class BinaryTree {
    TreeNode root; // root of the tree

    public BinaryTree(int data) {
        root = new TreeNode(data); // initialize the root
    }

}

```

The logic behind inserting nodes and other operations depend on the type of tree being used. For now, we are not going to worry about common operations, though they will be covered in future assignments.

## Printing the Tree

To properly print a tree, you need to traverse it. Traversal is an operation that will be covered in more detail in the next assignment. Instead, we are going to manually walk through a tree with exactly five nodes:

```

tex -hide-clipboard
      1      /
     / \      2      3      /
    4   5

```

Create the `printTree` method. Start by printing the data stored in the root node. Then print the the data from the children of the root (2 and 3). Only the left child (2) has children, so the final step is to print them.

```

public void printTree() {
    System.out.print(root.data + " ");

    System.out.print(root.left.data + " ");
    System.out.print(root.right.data + " ");

    System.out.print(root.left.left.data + " ");
    System.out.println(root.left.right.data);
}

```

Again, we will cover more sophisticated ways to traverse and print the values stored in a tree in a later assignment.

### ▼ Code

Your code should look like this:

```

class TreeNode {
    int data; // data of the node
    TreeNode left; // reference to the left child
    TreeNode right; // reference to the right child

    // Constructor to initialize the node with data and null
    children
    public TreeNode(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

class BinaryTree {
    TreeNode root; // root of the tree

    public BinaryTree(int data) {
        root = new TreeNode(data); // initialize the root
    }

    public void printTree() {
        System.out.print(root.data + " ");

        System.out.print(root.left.data + " ");
        System.out.print(root.right.data + " ");

        System.out.print(root.left.left.data + " ");
        System.out.println(root.left.right.data);
    }
}

```

Create the `main` method and instantiate a `BinaryTree` object. Give the tree a root with the value of 1, two children with the values 2 and 3. Then node 2 should have the children 4 and 5. Finally, print the tree.

```
public static void main(String[] args) {
    BinaryTree tree = new BinaryTree(1); // root
    tree.root.left = new TreeNode(2); // left child of root
    tree.root.right = new TreeNode(3); // right child of
                                    root
    tree.root.left.left = new TreeNode(4); // left child of
   2
    tree.root.left.right = new TreeNode(5); // right child
   of 2

    tree.printTree(); // This will print: 1 2 3 4 5
}
```

This page provides a simple representation and a way to print out the tree. The representation can be expanded to include additional methods for inserting, deleting, searching nodes, and other tree operations as per the requirements of your specific application.

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- Define pre-order, in-order, and post-order traversals
- Implement the three aforementioned traversals
- Analyze traversal applications

# Tree Traversal

## Iterating Over a Tree

**Tree traversal** is a foundational concept that involves visiting every node in a tree exactly once. As trees are a non-linear, hierarchical data structure, traversal is not as straightforward as it is with linear data structures. With arrays and linked lists, you can use a single loop to traverse the entire structure.

Tree traversal is crucial in various applications, such as executing searches and performing sorts. If we are successfully hitting all nodes, then we can perform operations on all of them. One operation we will focus on to show traversal is printing our nodes. We will delve deeply into tree traversal methodologies, focusing on implementations in Java.

## Traversal Strategies

In the context of trees, traversal pertains to the systematic visitation of all nodes in a specific order. There are several strategies to traverse a tree, each with its unique approach and utility. Here are the primary tree traversal strategies we will focus on:

---

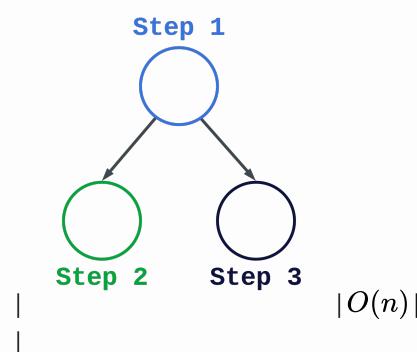
| Traversal Strategy | Graphical Representation | Time Complexity |
|--------------------|--------------------------|-----------------|
|--------------------|--------------------------|-----------------|

---

|

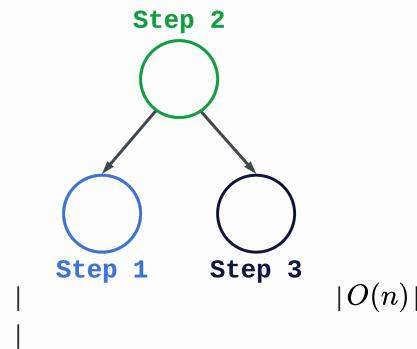
### Pre-Order

- Step 1: Visit the root.
- Step 2: Visit the left subtree.
- Step 3: Visit the right subtree.



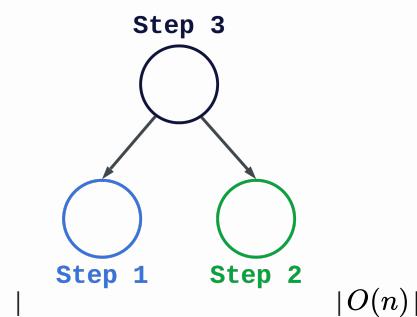
## In-Order

- Step 1: Visit the left subtree.
- Step 2: Visit the root.
- Step 3: Visit the right subtree.



## Post-Order

- Step 1: Visit the left subtree.
- Step 2: Visit the right subtree.
- Step 3: Visit the root.



The idea behind traversing a tree is to iterate over every node in the structure. Pre-order, in-order, and post-order traversals visit all of the nodes. They only differ on the order of traversal. Because of these, these three traversal strategies share the same time complexity,  $O(n)$ .

The traversal strategy depends largely on what you are looking to do. Pre-order traversal works best for making a copy of a tree or any other action that depends on the root first. In-order traversal is useful for when you want to sort a tree in ascending order, and post-order traversal is useful for when deleting nodes of a tree or any actions that depend on accessing the children before the root.

# Creating Our Tree

## Representing a Tree

To perform a traversal, we first need to represent a tree. We can represent a tree using nodes, where each node has a value, a reference to its left child, and a reference to its right child. Below is the Java representation of the tree.

### Node and Tree Classes

As usual, we are going to start with the `TreeNode` class. It has attributes to store data and point to nodes on the left and right. We also need a constructor to set the initial values. Data to be stored will be passed to the constructor, while the left and right nodes should be set to `null`.

```
class TreeNode {  
    int data;  
    TreeNode left;  
    TreeNode right;  
  
    public TreeNode(int data) {  
        this.data = data;  
        this.left = null;  
        this.right = null;  
    }  
}
```

In our `Tree` class, we need a root node. Have the constructor set `root` to a `TreeNode` object.

```
class Tree {  
    TreeNode root;  
  
    public Tree(int data) {  
        root = new TreeNode(data);  
    }  
}
```

## Adding Nodes

This is the basic tree structure we are familiar with, and we are going to replicate in code:

```
tex -hide-clipboard      5      / \      3      8      / \      / \
1   4   7   9
```

Create a `main` method that instantiates a `Tree` object with 5 as the value to store. The root's left child has the value 3 and the right child has the value 8. The node with 3 (`tree.root.left`) has 1 as the left child and 4 and the right child. The 8 node (`tree.root.right`) has 7 as the left child and 9 as the right child.

```
public static void main(String[] args) {
    // Creating the nodes for the tree
    Tree tree = new Tree(5);
    tree.root.left = new TreeNode(3);
    tree.root.right = new TreeNode(8);
    tree.root.left.left = new TreeNode(1);
    tree.root.left.right = new TreeNode(4);
    tree.root.right.left = new TreeNode(7);
    tree.root.right.right = new TreeNode(9);

    // The tree is now created and can be traversed.
}
```

### ▼ Code

Your code should look like this:

```

class TreeNode {
    int data;
    TreeNode left;
    TreeNode right;

    public TreeNode(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

class Tree {
    TreeNode root;

    public Tree(int data) {
        root = new TreeNode(data);
    }
}

public class TraversalExample {
    public static void main(String[] args) {
        // Creating the nodes for the tree
        Tree tree = new Tree(5);
        tree.root.left = new TreeNode(3);
        tree.root.right = new TreeNode(8);
        tree.root.left.left = new TreeNode(1);
        tree.root.left.right = new TreeNode(4);
        tree.root.right.left = new TreeNode(7);
        tree.root.right.right = new TreeNode(9);

        // The tree is now created and can be traversed.
    }
}

```

This page establishes an example tree structure, providing a tangible foundation for further traversal implementations. In the upcoming sections, we will delve into different tree traversal methods and implement them in Java. The tree on this page will serve as our reference example. Keep this tree structure in mind as we go through the various traversal methodologies and try to apply each method to this example to enhance your understanding of tree traversal in Java.

# Pre-Order Traversal

## Traversing the Root Node First

**Pre-order** traversal is one of the most common methods to traverse a tree. In this traversal method, you visit the root node first, then recursively do a pre-order traversal of the left subtree, followed by a recursive pre-order traversal of the right subtree.

Remember, our tree looks like this:

```
tex -hide-clipboard      5      / \      3      8      / \      / \
1   4   7   9
```

That means if we conduct a pre-order traversal, we would iterate through the data structure in this order:

```
tex -hide-clipboard 5 -> 3 -> 1 -> 4 -> 8 -> 7 -> 9
```

## Implementation

Trees are a self-similar data structure. That means the overall structure of the entire tree can be found in each of the subtrees. These types of data structures lend themselves to recursion, which is how we are going to perform each of these traversals.

Start by creating the `preOrderTraversal` method inside the `Tree` class. It takes a `TreeNode` object as a parameter. The base case is when the given `TreeNode` object is `null`. Otherwise, print the data stored in the given node. Then recurse on the left node. Finally recurse on the right node.

```
public static void preOrderTraversal(TreeNode node) {
    if (node != null) {
        // Visit the root node
        System.out.print(node.data + " ");
        // Traverse the left subtree in pre-order
        preOrderTraversal(node.left);
        // Traverse the right subtree in pre-order
        preOrderTraversal(node.right);
    }
}
```

Now think about how we would call this recursive method. Because it expects a `TreeNode` object as a parameter, we would have to call it like this (**note**, this is just an example, do not copy/paste it into the IDE):

```
tree.preOrderTraversal(tree.root);
```

This seems a bit clumsy. The root node is the first node in a tree. Should we not automatically start there? To get around this issue, we are going to perform some method overloading. Declare `preOrderTraversal` again, but this time there are no parameters. In the body, call the `preOrderTraversal` method and pass it the `root` attribute. This way, we can say `tree.preOrderTraversal()` and traverse the entire tree.

```
public void preOrderTraversal() {  
    preOrderTraversal(root);  
    System.out.println();  
}
```

Update the `main` method to include a call to the `preOrderTraversal` method. You do not need to alter any of the code that creates our tree.

```
public static void main(String[] args) {  
    // Creating the nodes for the tree  
    Tree tree = new Tree(5);  
    tree.root.left = new TreeNode(3);  
    tree.root.right = new TreeNode(8);  
    tree.root.left.left = new TreeNode(1);  
    tree.root.left.right = new TreeNode(4);  
    tree.root.right.left = new TreeNode(7);  
    tree.root.right.right = new TreeNode(9);  
  
    // Perform and print the result of in-order traversal  
    System.out.println("Pre-Order Traversal:");  
    tree.preOrderTraversal();  
}
```

## ▼ Code

Your code should look like this:

```
class TreeNode {  
    int data;  
    TreeNode left;  
    TreeNode right;
```

```

public TreeNode(int data) {
    this.data = data;
    this.left = null;
    this.right = null;
}

class Tree {
    TreeNode root;

    public Tree(int data) {
        root = new TreeNode(data);
    }

    public void preOrderTraversal(TreeNode root) {
        if (root != null) {
            // Visit the root node
            System.out.print(root.data + " ");
            // Traverse the left subtree in pre-order
            preOrderTraversal(root.left);
            // Traverse the right subtree in pre-order
            preOrderTraversal(root.right);
        }
    }

    public void preOrderTraversal() {
        preOrderTraversal(root);
        System.out.println();
    }
}

public class TraversalExample {
    public static void main(String[] args) {
        // Creating the nodes for the tree
        Tree tree = new Tree(5);
        tree.root.left = new TreeNode(3);
        tree.root.right = new TreeNode(8);
        tree.root.left.left = new TreeNode(1);
        tree.root.left.right = new TreeNode(4);
        tree.root.right.left = new TreeNode(7);
        tree.root.right.right = new TreeNode(9);

        // Perform and print the result of in-order traversal
        System.out.println("Pre-Order Traversal:");
        tree.preOrderTraversal();
    }
}

```

Running the program should produce the following output:

```
tex -hide-clipboard Pre-order Traversal: 5 3 1 4 8 7 9
```

This output is consistent with pre-order traversal – visiting the root node first, then traversing the left subtree, followed by the right subtree. After mastering this traversal technique, understanding other traversal methods becomes more intuitive and manageable.

# In-Order Traversal

## Traversing the Left Node First

**In-order** Traversal is another essential tree traversal method, and it is especially useful for binary search trees. In this traversal method, you recursively visit the left subtree, visit the root node, and finally recursively visit the right subtree.

Our tree looks like this:

```
tex -hide-clipboard      5      / \      3      8      / \      / \
1   4   7   9
```

That means if we conduct an in-order traversal, we would iterate through the data structure in this order:

```
tex -hide-clipboard 1 -> 3 -> 4 -> 5 -> 7 -> 8 -> 9
```

## Implementation

Start by creating the `inOrderTraversal` method in the `Tree` class. The method takes a `TreeNode` object as a parameter. As long as this node is not `null`, recursively call `inOrderTraversal` with the left subtree (`node.left`). When this is complete, visit the node itself. Lastly, recursively visit the right subtree (`node.right`).

```
public void inOrderTraversal(TreeNode node) {
    if (node != null) {
        // Traverse the left subtree in in-order
        inOrderTraversal(node.left);
        // Visit the root node
        System.out.print(node.data + " ");
        // Traverse the right subtree in in-order
        inOrderTraversal(node.right);
    }
}
```

Again, we are going to overload the `inOrderTraversal` method so that we can call it without any arguments and have the method start with the root node.

```
public void inOrderTraversal() {
    inOrderTraversal(root);
    System.out.println();
}
```

Update the `main` method to include a call to the `inOrderTraversal` method. You do not need to alter any of the code that creates our tree.

```
public static void main(String[] args) {
    // Creating the nodes for the tree
    Tree tree = new Tree(5);
    tree.root.left = new TreeNode(3);
    tree.root.right = new TreeNode(8);
    tree.root.left.left = new TreeNode(1);
    tree.root.left.right = new TreeNode(4);
    tree.root.right.left = new TreeNode(7);
    tree.root.right.right = new TreeNode(9);

    // Perform and print the result of in-order traversal
    System.out.println("In-Order Traversal:");
    tree.inOrderTraversal();
}
```

## ▼ Code

Your code should look like this:

```
class TreeNode {
    int data;
    TreeNode left;
    TreeNode right;

    public TreeNode(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

class Tree {
    TreeNode root;

    public Tree(int data) {
        root = new TreeNode(data);
    }
}
```

```

public void preOrderTraversal(TreeNode root) {
    if (root != null) {
        // Visit the root node
        System.out.print(root.data + " ");
        // Traverse the left subtree in pre-order
        preOrderTraversal(root.left);
        // Traverse the right subtree in pre-order
        preOrderTraversal(root.right);
    }
}

public void preOrderTraversal() {
    preOrderTraversal(root);
    System.out.println();
}

public void inOrderTraversal(TreeNode node) {
    if (node != null) {
        // Traverse the left subtree in in-order
        inOrderTraversal(node.left);
        // Visit the root node
        System.out.print(node.data + " ");
        // Traverse the right subtree in in-order
        inOrderTraversal(node.right);
    }
}

public void inOrderTraversal() {
    inOrderTraversal(root);
}
}

public class TraversalExample {
    public static void main(String[] args) {
        // Creating the nodes for the tree
        Tree tree = new Tree(5);
        tree.root.left = new TreeNode(3);
        tree.root.right = new TreeNode(8);
        tree.root.left.left = new TreeNode(1);
        tree.root.left.right = new TreeNode(4);
        tree.root.right.left = new TreeNode(7);
        tree.root.right.right = new TreeNode(9);

        // Perform and print the result of in-order traversal
        System.out.println("In-Order Traversal:");
        tree.inOrderTraversal();
    }
}

```

Running the program should produce the following output:

```
tex -hide-clipboard In-Order Traversal: 1 3 4 5 7 8 9
```

In-order traversal is particularly useful in binary search trees, where it returns the nodes in ascending order (we will cover binary search trees in greater detail in a later assignment). Mastering in-order traversal is crucial for efficient manipulation and utilization of tree structures. The next step is to understand post-order traversal.

# Post-Order Traversal

## Traversing the Root Node Last

**Post-order** traversal is a method of tree traversal in which you recursively visit the left subtree, recursively visit the right subtree, and then visit the root node.

Our tree looks like this:

```
tex -hide-clipboard      5      / \      3      8      / \      / \
1   4   7   9
```

That means if we conduct an post-order traversal, we would iterate through the data structure in this order:

```
tex -hide-clipboard 1 -> 4 -> 3 -> 7 -> 9 -> 8 -> 5
```

## Implementation

Start by creating the `postOrderTraversal` method in the `Tree` class. The method takes a `TreeNode` object as a parameter. As long as this node is not `null`, recursively call `postOrderTraversal` with the left subtree (`node.left`). When this is complete, recursively visit the right subtree (`node.right`). Lastly, visit the node itself.

```
public void postOrderTraversal(TreeNode node) {
    if (node != null) {
        // Traverse the left subtree in post-order
        postOrderTraversal(node.left);
        // Traverse the right subtree in post-order
        postOrderTraversal(node.right);
        // Visit the root node
        System.out.print(node.data + " ");
    }
}
```

Again, we are going to overload the `postOrderTraversal` method so that we can call it without any arguments and have the method start with the root node.

```
public void postOrderTraversal() {
    postOrderTraversal(root);
    System.out.println();
}
```

Update the `main` method to include a call to the `postOrderTraversal` method. You do not need to alter any of the code that creates our tree.

```
public static void main(String[] args) {
    // Creating the nodes for the tree
    Tree tree = new Tree(5);
    tree.root.left = new TreeNode(3);
    tree.root.right = new TreeNode(8);
    tree.root.left.left = new TreeNode(1);
    tree.root.left.right = new TreeNode(4);
    tree.root.right.left = new TreeNode(7);
    tree.root.right.right = new TreeNode(9);

    // Perform and print the result of in-order traversal
    System.out.println("Post-Order Traversal:");
    tree.postOrderTraversal();
}
```

## ▼ Code

Your code should look like this:

```
class TreeNode {
    int data;
    TreeNode left;
    TreeNode right;

    public TreeNode(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

class Tree {
    TreeNode root;

    public Tree(int data) {
        root = new TreeNode(data);
    }
}
```

```

public void preOrderTraversal(TreeNode root) {
    if (root != null) {
        // Visit the root node
        System.out.print(root.data + " ");
        // Traverse the left subtree in pre-order
        preOrderTraversal(root.left);
        // Traverse the right subtree in pre-order
        preOrderTraversal(root.right);
    }
}

public void preOrderTraversal() {
    preOrderTraversal(root);
    System.out.println();
}

public void inOrderTraversal(TreeNode node) {
    if (node != null) {
        // Traverse the left subtree in in-order
        inOrderTraversal(node.left);
        // Visit the root node
        System.out.print(node.data + " ");
        // Traverse the right subtree in in-order
        inOrderTraversal(node.right);
    }
}

public void inOrderTraversal() {
    inOrderTraversal(root);
}

public void postOrderTraversal(TreeNode node) {
    if (node != null) {
        // Traverse the left subtree in post-order
        postOrderTraversal(node.left);
        // Traverse the right subtree in post-order
        postOrderTraversal(node.right);
        // Visit the root node
        System.out.print(node.data + " ");
    }
}

public void postOrderTraversal() {
    postOrderTraversal(root);
}
}

public class TraversalExample {

```

```
public static void main(String[] args) {
    // Creating the nodes for the tree
    Tree tree = new Tree(5);
    tree.root.left = new TreeNode(3);
    tree.root.right = new TreeNode(8);
    tree.root.left.left = new TreeNode(1);
    tree.root.left.right = new TreeNode(4);
    tree.root.right.left = new TreeNode(7);
    tree.root.right.right = new TreeNode(9);

    // Perform and print the result of post-order traversal
    System.out.println("Post-Order Traversal:");
    tree.postOrderTraversal();
}

}
```

Running the program should produce the following output:

```
tex -hide-clipboard Post-Order Traversal: 1 4 3 7 9 8 5
```

Post-Order Traversal is indispensable in scenarios where you need to perform operations on a node only after performing equivalent operations on its descendants, for example, while deleting a tree.

In the following assignments, we will delve into more complex tree manipulation techniques and explore additional tree traversal methodologies that can enhance your tree manipulation and analysis skills.

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

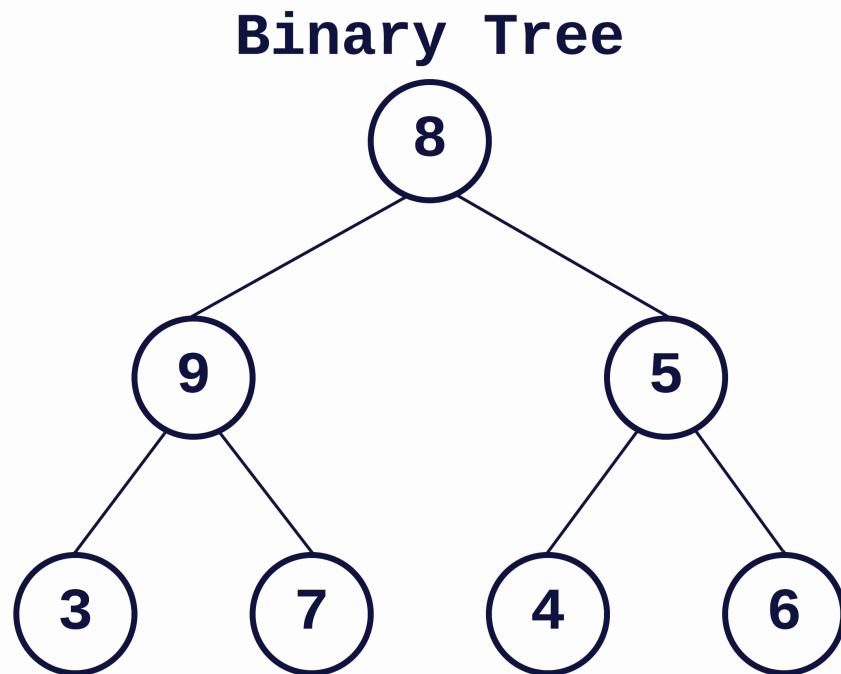
- **Define binary and binary search trees**
  - **Differentiate binary and binary search trees**
  - **Insert and delete nodes in a binary search tree**
  - **Traverse a binary search tree in sequential order**
  - **Analyze binary search tree efficiencies**
-

# Trees with Only Two Nodes

In computer science, particularly in the field of data structures, trees represent a hierarchical structure of nodes, where each node links to several subtrees. While trees come in various forms, this assignment focuses on a special category of trees that have, at most, two nodes. These trees are known as **binary trees**. There is even a particular type of binary tree called **binary search trees (BST)**.

This initial exploration into binary trees and BSTs lays the groundwork for understanding how these structures function and how they are used in computer science.

## Binary Trees



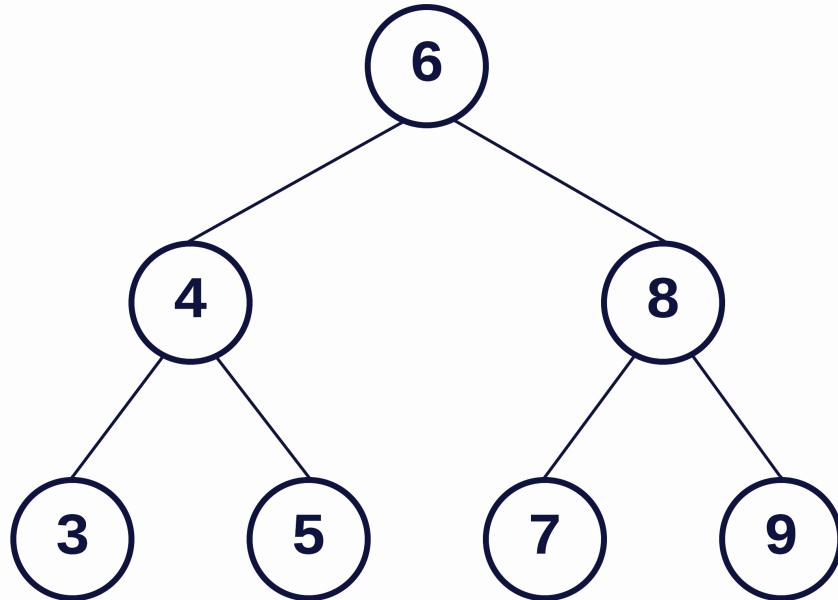
A **binary tree** is a tree data structure in which each node has at most two children, referred to as the left child and the right child. This definition is crucial because it sets binary trees apart from other tree structures that can have any number of children.

There are also no restrictions on the order of data stored in a binary tree. This gives the data structure some added flexibility. Binary trees often serve as the foundation for more complex data structures binary search

trees, heaps (covered later), and segment trees (covered later). They are also used by applications that naturally structure and manage data in a hierarchical manner.

## Binary Search Trees

### Binary Search Tree



A **binary search tree (BST)** is an ordered binary tree designed for quick searching, insertion, and deletion operations. In a BST, the nodes are arranged following specific properties. For every node, the value of all the nodes in its left subtree is less than its own value, and the value of all the nodes in its right subtree is greater. A BST typically does not contain duplicate values. This property helps maintain a consistent order and enables unique paths to each node.

These properties allow efficient lookup, insertion, and deletion operations. The time complexity of these operations is  $O(h)$ , where  $h$  is the height of the tree. If you perform an in-order traversal, you will retrieve the nodes in sorted order, which is a useful property for various applications. BSTs are excellent for cases where the data is constantly changing, such as managing an online marketplace's product listings or a real-time priority queue.

One of the critical challenges with BSTs is maintaining balance. A balanced tree is one in which the difference in height between the left subtree and right subtree of every node is no greater than one. This allows operations

to be performed in  $O(\log n)$  time. However, without balance (in the case of a degenerate tree), the performance deteriorates to  $O(n)$ , equivalent to a linked list.

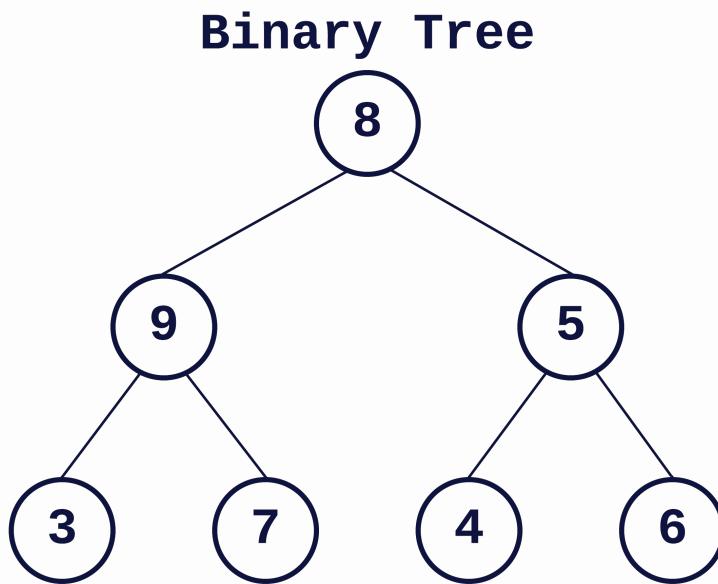
# Binary Trees

## Deep Dive into Binary Trees

In the world of data structures, binary trees hold a special place due to their efficiency in performing various operations.. Understanding the nuances of binary trees is crucial before we explore more complex tree structures.

### What Makes a Tree Binary?

A binary tree is characterized by a specific rule: each node can have zero, one, or two children, and no more. Each child is referred to as a left child or a right child. This rule creates the structure that we recognize as a binary tree.



The image shows a binary tree with seven nodes. The root is 8, the left child is 9 and the right child is 5. Node 9 has the children 3 and 7. Node 5 has the children 4 and 6.

In the tree above, each node has at most two children. This structure is what defines it as a binary tree. Importantly, there are no restrictions on how the data is stored, which gives the data structure some added flexibility. Binary trees often serve as the foundation for more complex

data structures binary search trees, heaps (covered later), and segment trees (covered later). They are also used by applications that naturally structure and manage data in a hierarchical manner.

## Properties of Binary Trees

Several important properties derive from the binary tree structure, affecting how we interact with and utilize binary trees:

- **Maximum Number of Nodes:** At any given level  $i$  in a binary tree, the maximum number of nodes is  $2^i$ , where  $i$  is the depth of the level (assuming the root is at level 0).
- **Maximum Total Nodes:** A binary tree of height  $h$  can have at most  $2^{(h+1)} - 1$  nodes, where  $h$  is the height of the tree (the number of nodes along the longest path from the root node down to the farthest leaf node).
- **Minimum Possible Height:** For a binary tree with  $n$  nodes, the minimum possible height or the minimum number of levels is  $\log_2(n + 1)$ .
- **Balanced Binary Tree:** A balanced binary tree is one where the depth of the left and right subtrees of every node differ by at most one. Balanced trees are essential for maintaining optimal performance ( $O(\log n)$  time complexity) for core operations.
- **Complete Binary Tree:** A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

## Types of Binary Trees

There are specific categories of binary trees used for different purposes, including:

- **Full Binary Tree:** Every node has exactly 0 or 2 children (not 1).
- **Perfect Binary Tree:** A full binary tree where all leaf nodes are at the same level.
- **Degenerate (or Pathological) Tree:** A tree where each parent node has only one child, making its performance similar to a linked list.

Understanding these properties and types is crucial for recognizing how binary trees function and are used in algorithms and data processing.

# Binary Search Trees

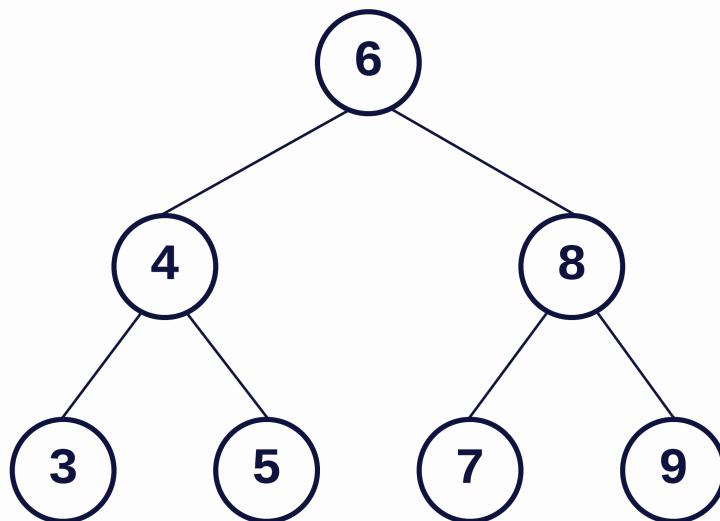
## What is a Binary Search Tree?

Binary trees come in various forms, each with its unique properties and uses. One of the most powerful and commonly used types is the **binary search tree (BST)**. It not only inherits the structural properties of a binary tree but also introduces a new set of rules that offer efficient search capabilities. A binary search tree is a node-based binary tree data structure that has the following properties:

- The left subtree of a node contains only nodes with values lesser than the node's value.
- The right subtree of a node contains only nodes with values greater than the node's value.
- There are no duplicate nodes; the left and right subtree each must also be a binary search tree.

These properties significantly reduce the time it takes to locate a specific node because one can use the principles of binary search.

## Binary Search Tree



The image depicts a binary search tree with 7 nodes. The root node is 6, the left child is 4, and the right child is 8. Node 4 has the children 3 and 5. Node 8 has the children 7 and 9.

Each value to the left of root node is less than 6, while each value to the right is greater. This rule applies to every node in the tree, not just the root.

## Why Use Binary Search Trees?

- **Efficient Search Operations:** Due to their structure, BSTs allow for efficient search operations, similar to those in sorted arrays but with faster insertion and deletion capabilities.
- **Sorted Order Retrieval:** BSTs can retrieve the elements in sorted order using in-order traversal. This feature is beneficial for algorithms that require sorted elements without needing to sort an entire array or list.
- **Flexibility in Size:** Unlike static data structures (like arrays), BSTs are dynamic and can easily expand or shrink in size, accommodating new elements or removing existing ones.
- **Balanced BSTs:** Certain types of BSTs (like AVL trees or Red-Black trees) self-balance, ensuring that the tree remains proportioned and that operations like insertion, deletion, and search take logarithmic time.

## Potential Downsides

However, there are certain limitations and considerations as well:

- **Complexity:** Implementing BSTs is generally more complex than other linear data structures due to pointer usage and recursive thinking during operations.
- **Unbalanced Trees:** If not maintained, BSTs can become skewed or unbalanced, especially after numerous insertion and deletion operations. This imbalance can lead to decreased performance, with time complexities degrading from  $O(\log n)$  to  $O(n)$  in the worst case.

## Real-World Applications

BSTs are instrumental in various applications, including:

- Database indexing, where quick search, insertion, and deletion are crucial.
- Dynamic sorting of datasets that frequently change.
- Implementing associative arrays (maps) efficiently.

# Insertion in Binary Search Trees

## Inserting Nodes into a Binary Search Tree

Understanding the theory behind BSTs is integral, but nothing beats actual implementation. Let's construct a simple BST with operations for inserting and searching nodes. We'll create a robust structure that can easily be expanded with additional functionalities.

The first thing we need to is create a class for the nodes in our tree. Each node has three attributes: the value it holds, a reference to the left child, and a reference to the right child. In the constructor, set the pointers to `null` and store the given value in the `value` attribute.

```
class TreeNode {
    int value;
    TreeNode left;
    TreeNode right;

    // Constructor
    public TreeNode(int value) {
        this.value = value;
        this.left = null; // No left child initially
        this.right = null; // No right child initially
    }
}
```

Now, we'll create the `BinarySearchTree` class, which will encapsulate our tree operations. The class has only one attribute, `root` which is the first node in our tree. In the constructor, set `root` to `null`.

```
class BinarySearchTree {
    TreeNode root; // The root node of our BST

    // Constructor
    public BinarySearchTree() {
        root = null; // Start with an empty tree
    }
}
```

Similar to printing the nodes in the tree, we are going to iterate over the structure with recursion. We are also going to separate our insertion into two different method calls. Start with the `insert` method that takes a value to insert into the tree. Set the `root` attribute to the result of the overloaded method `insert` that takes the root of the tree and the value to insert.

```
// Insert a node (public method)
public void insert(int value) {
    root = insert(root, value); // Call the recursive helper
    function to insert
}
```

At first glance, it may seem like we are inserting the new node at the root of the tree. However, we will construct the overloaded `insert` method such that it will not alter any of the nodes in the tree until the correct location is found.

The overloaded `insert` method is also a recursive method. As with any recursive method, start by declaring the base case. If the current node is `null`, create a new node with the given value and return this new node.

```
// Recursive insertion (private helper method)
private TreeNode insert(TreeNode node, int value) {
    /* If the tree is empty, return a new node */
    if (node == null) {
        node = new TreeNode(value);
        return node;
    }

}
```

Otherwise, we need to find the proper location for the new value. If the given value is less than the value of the current node, assign the left pointer to `insert(node.left, value)`. This recurses over the left subtree.

```

// Recursive insertion (private helper method)
private TreeNode insert(TreeNode node, int value) {
    /* If the tree is empty, return a new node */
    if (node == null) {
        node = new TreeNode(value);
        return node;
    }

    /* Otherwise, recur down the tree */
    if (value < node.value) {
        // value is less, so it goes to the left subtree
        node.left = insert(node.left, value);
    }

}

```

After finishing recursing over the left subtree, we need to do the same thing to the right subtree. Use a else-if statement that asks if the given value is greater than value in the current node. If so, set the value of the right node to `insert(node.right, value)`.

```

// Recursive insertion (private helper method)
private TreeNode insert(TreeNode node, int value) {
    /* If the tree is empty, return a new node */
    if (node == null) {
        node = new TreeNode(value);
        return node;
    }

    /* Otherwise, recur down the tree */
    if (value < node.value) {
        // value is less, so it goes to the left subtree
        node.left = insert(node.left, value);
    } else if (value > node.value) {
        // value is greater, so it goes to the right subtree
        node.right = insert(node.right, value);
    }

}

```

The last line is arguably the most important. We need to make sure that we are only modifying nodes at the point of insertion. Many nodes should remain unchanged during this operation. End the method by returning `node`. This variable represents the unchanged pointer to a `TreeNode` object.

```

private TreeNode insert(TreeNode node, int value) {
    /* If the tree is empty, return a new node */
    if (node == null) {
        node = new TreeNode(value);
        return node;
    }

    /* Otherwise, recur down the tree */
    if (value < node.value) {
        // value is less, so it goes to the left subtree
        node.left = insert(node.left, value);
    } else if (value > node.value) {
        // value is greater, so it goes to the right subtree
        node.right = insert(node.right, value);
    }

    /* return the (unchanged) node pointer */
    return node;
}

```

Create the `main` method such that you are instantiating a `BinarySearchTree` object. Insert nodes with the values 6 (this will be the root), 4, 8, 2, and 5. Nodes 4 and 8 are the children (left and right respectively) of the root. Nodes 2 and 5 are the children (left and right respectively) of node 4.

```

public static void main(String[] args) {
    BinarySearchTree bst = new BinarySearchTree();

    // Inserting values
    bst.insert(6);
    bst.insert(4);
    bst.insert(8);
    bst.insert(2);
    bst.insert(5);

    // Print values in the tree
    System.out.println("Root: " + bst.root.value);
    System.out.println("Left child: " +
        bst.root.left.value);
    System.out.println("Right child: " +
        bst.root.right.value);
    System.out.println("Left child of Node 4: " +
        bst.root.left.left.value);
    System.out.println("Right child of Node 4: " +
        bst.root.left.right.value);
}

```

▼ Code

Your code should look like this:

```
class TreeNode {
    int value;
    TreeNode left;
    TreeNode right;

    // Constructor
    public TreeNode(int value) {
        this.value = value;
        this.left = null; // No left child initially
        this.right = null; // No right child initially
    }
}

class BinarySearchTree {
    TreeNode root; // The root node of our BST

    // Constructor
    public BinarySearchTree() {
        root = null; // Start with an empty tree
    }

    // Insert a node (public method)
    public void insert(int value) {
        root = insert(root, value); // Call the recursive helper
                                    // function to insert
    }

    private TreeNode insert(TreeNode node, int value) {
        /* If the tree is empty, return a new node */
        if (node == null) {
            node = new TreeNode(value);
            return node;
        }

        /* Otherwise, recur down the tree */
        if (value < node.value) {
            // value is less, so it goes to the left subtree
            node.left = insert(node.left, value);
        } else if (value > node.value) {
            // value is greater, so it goes to the right subtree
            node.right = insert(node.right, value);
        }

        /* return the (unchanged) node pointer */
        return node;
    }
}
```

```
}

public class BstExample {
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();

        // Inserting values
        bst.insert(6);
        bst.insert(4);
        bst.insert(8);
        bst.insert(2);
        bst.insert(5);

        // Print values in the tree
        System.out.println("Root: " + bst.root.value);
        System.out.println("Left child: " +
            bst.root.left.value);
        System.out.println("Right child: " +
            bst.root.right.value);
        System.out.println("Left child of Node 4: " +
            bst.root.left.left.value);
        System.out.println("Right child of Node 4: " +
            bst.root.left.right.value);
    }
}
```

You should see the following output:

```
tex -hide-clipboard Root: 6 Left child: 4 Right child: 8 Left child
of Node 4: 2 Right child of Node 4: 5
```

Now that we have an understanding on how to insert a node into a binary search tree, we are going to move on to something a bit more challenging – deleting a node from a binary search tree.

# Deletion in Binary Search Trees

## Removing Nodes from a Binary Search Tree

We will now delve into one of the more complex operations on a binary search tree – deletion. Deleting nodes in a BST requires a careful approach that ensures the binary search tree property holds after the deletion. Unlike linear data structures like arrays or linked lists, we cannot simply remove a node; we must consider the node's children and how to “re-link” the tree.

Remember, we must maintain the BST property after deletion. In other words, the following must still hold true: A binary tree with nodes arranged such that left children have smaller values, right children have larger values, and no duplicate nodes exist. Both left and right subtrees must also be BSTs.

Three possible scenarios arise when you want to delete a node from a BST:

1. **Node with No Children (leaf node):** This is the simplest case; we can simply remove the node from the tree.
2. **Node with One Child:** We can delete the node and replace it with its child.
3. **Node with Two Children:** This is the most complex case. We will find the node's in-order successor (the smallest node in the right subtree) or in-order predecessor (the largest node in the left subtree) and replace the node's value with it. Then, we delete the in-order successor or predecessor (which has at most one child).

## Implementation

Let's update our `BinarySearchTree` class from the previous page to include a method for deleting nodes. To do this, we'll also need a helper method to find the minimum value in a tree (useful for finding the in-order successor). First, add the method to find the node with the minimum value.

```
private TreeNode minValNode(TreeNode node) {  
    // Find the leftmost leaf  
    while (node.left != null) {  
        node = node.left;  
    }  
    return node;  
}
```

Now we can implement the delete method. Again, this will be a helper method that takes the value to remove. The method then calls the recursive method to remove the value.

```
public void delete(int value) {  
    root = delete(root, value);  
}
```

The delete methods handles the actual deletion of the node. It takes two parameters, the current node and the value to be removed. Start by creating the base case, which is when the current node is null. Return node when the base case is satisfied.

```
private TreeNode delete(TreeNode node, int value) {  
    // If the tree is empty, return the same node  
    if (node == null) return node;  
  
    }  
}
```

If the current node is not null, iterate over the tree. Start by recursing down the left subtree if the given value is less than the current value. Otherwise, recurse down the right subtree if the given value is greater than the current value.

```
private TreeNode delete(TreeNode node, int value) {  
    // If the tree is empty, return the same node  
    if (node == null) return node;  
  
    // Otherwise, recurse down the tree  
    if (value < node.value) {  
        node.left = delete(node.left, value);  
    } else if (value > node.value) {  
        node.right = delete(node.right, value);  
    }  
}
```

The two conditionals above account for when the given value is less than or greater than the value of the current node. When the two values are equal, we need to account for the three possible cases above. If the target node does not have children, we simply remove it by changing the respective parent pointer to null. If the target node has one child, we link its parent to its child, and then delete the node.

```

private TreeNode delete(TreeNode node, int value) {
    // If the tree is empty, return the same node
    if (node == null) return node;

    // Otherwise, recurse down the tree
    if (value < node.value) {
        node.left = delete(node.left, value);
    } else if (value > node.value) {
        node.right = delete(node.right, value);
    } else {
        // Node with only one child or no child
        if (node.left == null) {
            return node.right;
        } else if (node.right == null) {
            return node.left;
        }
    }
}

```

However, if the target node has two children, we need to find its in-order successor. This is done with the `minValNode` method. Then copy the successor's value to the node, and then recursively delete the successor's node. Lastly return the node.

```

private TreeNode delete(TreeNode node, int value) {
    // If the tree is empty, return the same node
    if (node == null) return node;

    // Otherwise, recurse down the tree
    if (value < node.value) {
        node.left = delete(node.left, value);
    } else if (value > node.value) {
        node.right = delete(node.right, value);
    } else {
        // Node with only one child or no child
        if (node.left == null) {
            return node.right;
        } else if (node.right == null) {
            return node.left;
        }
    }

    // Node with two children: Get the in-order successor (smallest in the right subtree)
    node.value = minValNode(node.right).value;

    // Delete the in-order successor
    node.right = delete(node.right, node.value);
}

return node;
}

```

Update the `main` method so that node 4 is removed from the tree. The root's left child should now become 5, and 5 should have 2 as the left child.

```

public static void main(String[] args) {
    BinarySearchTree bst = new BinarySearchTree();

    // Inserting values
    bst.insert(6);
    bst.insert(4);
    bst.insert(8);
    bst.insert(2);
    bst.insert(5);

    // Delete node
    bst.delete(4);

    // Print values in the tree
    System.out.println("Root: " + bst.root.value);
    System.out.println("Left child: " +
        bst.root.left.value);
    System.out.println("Right child: " +
        bst.root.right.value);
    System.out.println("Left child of Node 5: " +
        bst.root.left.left.value);
}

```

## ▼ Code

Your code should look like this:

```

class TreeNode {
    int value;
    TreeNode left;
    TreeNode right;

    // Constructor
    public TreeNode(int value) {
        this.value = value;
        this.left = null; // No left child initially
        this.right = null; // No right child initially
    }
}

class BinarySearchTree {
    TreeNode root; // The root node of our BST

    // Constructor
    public BinarySearchTree() {
        root = null; // Start with an empty tree
    }
}

```

```

// Insert a node (public method)
public void insert(int value) {
    root = insert(root, value); // Call the recursive helper function to insert
}

private TreeNode insert(TreeNode node, int value) {
    
    if (node == null) {
        node = new TreeNode(value);
        return node;
    }

    
    if (value < node.value) {
        // value is less, so it goes to the left subtree
        node.left = insert(node.left, value);
    } else if (value > node.value) {
        // value is greater, so it goes to the right subtree
        node.right = insert(node.right, value);
    }

    
    return node;
}

private TreeNode minValNode(TreeNode node) {
    // Find the leftmost leaf
    while (node.left != null) {
        node = node.left;
    }
    return node;
}

public void delete(int value) {
    root = delete(root, value);
}

private TreeNode delete(TreeNode node, int value) {
    // If the tree is empty, return the same node
    if (node == null) return node;

    // Otherwise, recurse down the tree
    if (value < node.value) {
        node.left = delete(node.left, value);
    } else if (value > node.value) {
        node.right = delete(node.right, value);
    } else {

```

```

        // Node with only one child or no child
        if (node.left == null) {
            return node.right;
        } else if (node.right == null) {
            return node.left;
        }

        // Node with two children: Get the in-order
        // successor (smallest in the right subtree)
        node.value = minValNode(node.right).value;

        // Delete the in-order successor
        node.right = delete(node.right, node.value);
    }

    return node;
}

public class BstExample {
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();

        // Inserting values
        bst.insert(6);
        bst.insert(4);
        bst.insert(8);
        bst.insert(2);
        bst.insert(5);

        // Delete node
        bst.delete(4);

        // Print values in the tree
        System.out.println("Root: " + bst.root.value);
        System.out.println("Left child: " +
        bst.root.left.value);
        System.out.println("Right child: " +
        bst.root.right.value);
        System.out.println("Left child of Node 5: " +
        bst.root.left.left.value);
    }
}

```

You should see the following output:

```
tex -hide-clipboard Root: 6 Left child: 5 Right child: 8 Left child
of Node 5: 2
```

Next, we are going to talk about traversing a binary search tree.

# Traversal

## Printing a Tree

We have already talked about different traversal strategies in a previous assignment. However, the ordered property of binary search trees allow you to print a tree in sequential order (smallest to largest in our example) when you use an in-order traversal. Recall that an in-order traversal iterates over a tree in the following manner:

1. Traverse the left subtree by recursively calling the in-order method.
2. Display the data part of the root (or current node).
3. Traverse the right subtree by recursively calling the in-order method.

Let's enhance our `BinarySearchTree` class by adding an in-order traversal function that prints the elements. Additionally, we'll include a method to visually display the structure of the tree for better understanding. Again, we are going to use method overloading to differentiate the helper method (no parameters) from the recursive method (one parameter). The logic contained in these methods is the same from what we have seen before.

First, add the in-order traversal method:

```
public void inorder() {
    System.out.print("In-order Traversal:");
    inorder(root);
    System.out.println();
}

private void inorder(TreeNode root) {
    if (root != null) {
        inorder(root.left); // visit left subtree
        System.out.print(" " + root.value); // print data of
   // the node
        inorder(root.right); // visit right subtree
    }
}
```

## Printing by Level

So far, we have been printing a tree by traversing it node by node. While this does have value, the end result does not give us a representation of the hierarchical structure of the tree. This time, we are going to traverse the

tree level by level, printing the corresponding nodes (and whitespace) to give us a better representation of the data structure.

We will start by creating the `printLevels` helper method. It takes no parameters and calls the overloaded `printLevels` method and passes it the root node.

```
public void printLevels() {  
    printLevels(root);  
}
```

To traverse the tree by level, we need a way of keeping track of the nodes on each level. We can't use the binary tree itself as it is organized in a top down manner. Instead, we need an intermediary data structure. We are working with the tree below:

```
tex -hide-clipboard      6   4   8   2 5
```

As we progress through the levels, we first need to store 6 and then print it. On the next level, we store 4 and 8 (in that order), and then print 4 and 8 (in that order). Finally we store 2 and 5 (in that order), and then print 2 and 5 (in that order). Does this sound familiar? The first element stored is the first element printed (FIFO). We need a queue. However, we are not going to build a queue ourselves. Instead, we are going to use Java's the built-in data structure. At the top of the program, import both `Queue` and `LinkedList`.

```
import java.util.LinkedList;  
import java.util.Queue;
```

Next, we are going to create the overloaded `printLevels` method that takes no parameters. First, check to see if the tree is empty. If so, print a message to the user and exit the method. Otherwise, create a queue and add the root node to it. Then instantiate the variable `level` and set its value to `0`.

```
public void printLevels() {  
    if(root == null) {  
        System.out.println("The tree is empty");  
        return;  
    }  
  
    Queue<TreeNode> queue = new LinkedList<>();  
    queue.add(root);  
  
    int level = 0;  
}
```

Use a while loop to iterate over the queue as long as it is not empty. Create the variable size and set its value to the size of the queue. Print some text that tells the user what level they are viewing, and then increment level by one.

```
public void printLevels() {
    if(root == null) {
        System.out.println("The tree is empty");
        return;
    }

    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);

    int level = 0;

    while(!queue.isEmpty()) {

        int size = queue.size();
        System.out.print("Level " + level + ": ");
        level++;

    }
}
```

Create a for loop that runs as many times as the size of the queue. For each iteration, remove the first element (a node) from the queue and store it in a temporary variable. Print the value of the node followed by a space. If there is a left child to the current node, add it to the queue. Do the same thing for the right child. Lastly, use a `println` statement to terminate the current line of output and go to the next line.

```

public void printLevels() {
    if(root == null) {
        System.out.println("The tree is empty");
        return;
    }

    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);

    int level = 0;

    while(!queue.isEmpty()) {

        int size = queue.size();
        System.out.print("Level " + level + ": ");
        level++;

        // traverse all nodes of current level
        for(int i=0; i < size; i++) {
            TreeNode node = queue.remove();

            // print current node
            System.out.print(node.value + " ");

            // add child nodes to queue for next level
            if(node.left != null) queue.add(node.left);
            if(node.right != null) queue.add(node.right);
        }

        // newline after printing each level
        System.out.println();
    }
}

```

Update the `main` method to remove the deletion and print statements from the last page. In their place, we are going to call the `inorder` and `printLevels` methods.

```

public static void main(String[] args) {
    BinarySearchTree bst = new BinarySearchTree();

    // Inserting values
    bst.insert(6);
    bst.insert(4);
    bst.insert(8);
    bst.insert(2);
    bst.insert(5);

    bst.inorder();
    bst.printLevels();
}

```

## ▼ Code

Your code should look like this:

```

import java.util.LinkedList;
import java.util.Queue;

class TreeNode {
    int value;
    TreeNode left;
    TreeNode right;

    // Constructor
    public TreeNode(int value) {
        this.value = value;
        this.left = null; // No left child initially
        this.right = null; // No right child initially
    }
}

class BinarySearchTree {
    TreeNode root; // The root node of our BST

    // Constructor
    public BinarySearchTree() {
        root = null; // Start with an empty tree
    }

    // Insert a node (public method)
    public void insert(int value) {
        root = insert(root, value); // Call the recursive helper
        function to insert
    }
}

```

```

private TreeNode insert(TreeNode node, int value) {
    /* If the tree is empty, return a new node */
    if (node == null) {
        node = new TreeNode(value);
        return node;
    }

    /* Otherwise, recur down the tree */
    if (value < node.value) {
        // value is less, so it goes to the left subtree
        node.left = insert(node.left, value);
    } else if (value > node.value) {
        // value is greater, so it goes to the right subtree
        node.right = insert(node.right, value);
    }

    /* return the (unchanged) node pointer */
    return node;
}

private TreeNode minValNode(TreeNode node) {
    // Find the leftmost leaf
    while (node.left != null) {
        node = node.left;
    }
    return node;
}

public void delete(int value) {
    root = delete(root, value);
}

private TreeNode delete(TreeNode node, int value) {
    // If the tree is empty, return the same node
    if (node == null) return node;

    // Otherwise, recurse down the tree
    if (value < node.value) {
        node.left = delete(node.left, value);
    } else if (value > node.value) {
        node.right = delete(node.right, value);
    } else {
        // Node with only one child or no child
        if (node.left == null) {
            return node.right;
        } else if (node.right == null) {
            return node.left;
        } else {
            // Node with two children
            // Replace node with its in-order successor
            // (the leftmost node of its right subtree)
            node.value = minValNode(node.right).value;
            node.right = delete(node.right, node.value);
        }
    }
}

```

```

        }

        // Node with two children: Get the in-order
        // successor (smallest in the right subtree)
        node.value = minValNode(node.right).value;

        // Delete the in-order successor
        node.right = delete(node.right, node.value);
    }

    return node;
}

public void inorder() {
    System.out.print("In-order Traversal:");
    inorder(root);
    System.out.println();
}

private void inorder(TreeNode root) {
    if (root != null) {
        inorder(root.left); // visit left subtree
        System.out.print(" " + root.value); // print data of
        // the node
        inorder(root.right); // visit right subtree
    }
}

public void printLevels() {
    if (root == null) {
        System.out.println("The tree is empty");
        return;
    }

    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);

    int level = 0;

    while (!queue.isEmpty()) {

        int size = queue.size();
        System.out.print("Level " + level + ": ");
        level++;

        // traverse all nodes of current level
        for (int i=0; i<size; i++) {
            TreeNode node = queue.remove();

```

```

    // print current node
    System.out.print(node.value + " ");

    // add child nodes to queue for next level
    if(node.left != null) queue.add(node.left);
    if(node.right != null) queue.add(node.right);
}

// newline after printing each level
System.out.println();
}

}

public class BstExample {
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();

        // Inserting values
        bst.insert(6);
        bst.insert(4);
        bst.insert(8);
        bst.insert(2);
        bst.insert(5);

        // Print tree
        bst.inorder();
        bst.printLevels();
    }
}

```

You should see the following output:

```
tex -hide-clipboard In-order Traversal: 2 4 5 6 8 Level 0: 6  Level
1: 4 8  Level 2: 2 5
```

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- **Distinguish between max-heaps and min-heaps by identifying their structural properties and understanding their roles in various applications.**
- **Illustrate the structure of complete binary trees and explain how heaps are represented in memory, particularly using array-based implementations.**
- **Demonstrate the process of inserting elements into a heap, including the steps of maintaining heap properties and analyzing the time complexity of the insertion operation.**
- **Execute the deletion operation in heaps, detailing the steps involved in removing elements while preserving heap structure and evaluating the efficiency of the operation.**
- **Identify and explain various real-world applications of heaps, emphasizing their utility in scenarios like priority queue implementation and efficient data management.**

# What Is a Heap?

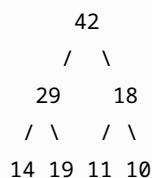
A heap is a specialized tree-based data structure that satisfies the heap property. In a **max-heap**, for any given node  $i$  other than the root, the value of  $i$  is less than or equal to the value of its parent. Conversely, in a **min-heap**, the value of  $i$  is greater than or equal to the value of its parent. This property makes heaps useful in a variety of applications, most notably in implementing *priority queues*.

## Max-Heaps and Min-Heaps

### Max-Heap

In a max-heap, the *largest* element is at the root, and this property is recursively true for all sub-trees in the heap.

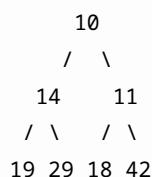
#### Example of a max-heap:



### Min-Heap

Conversely, in a min-heap, the *smallest* element is at the root, and this property holds for all sub-trees.

#### Example of a min-heap:



## Heap Representation

Heaps are usually represented as binary trees, but they are typically implemented using arrays for efficiency. In the array representation, if the root element is at index  $0$ , then for any given element at index  $i$ :

- The left child is at index  $2 * i + 1$
- The right child is at index  $2 * i + 2$
- The parent is at index  $(i - 1) / 2$  (floored)

This representation is **space-efficient** and makes it easy to traverse the heap.

## Heap Operations

The primary operations associated with heaps are:

- **Insertion:** Adding a new element to the heap while maintaining the heap property. This involves placing the element at the end of the heap (in the array representation) and then “bubbling up” or “heapifying up” the element to its correct position.
- **Deletion:** This usually refers to removing the root element from the heap, which is the maximum in a max-heap or the minimum in a min-heap. After removing the root, the last element in the heap is placed at the root, and then the heap is “heapified down” to maintain the heap property.
- **Heapify:** This operation transforms a binary tree into a heap. It’s essential when creating a heap from an unsorted array and during the deletion operation to restructure the heap.

## Applications of Heaps

Heaps are particularly useful in several areas:

- **Priority Queues:** Heaps are ideal for implementing priority queues where elements are processed based on their priority rather than their order in the queue.
- **Heap Sort:** This sorting algorithm uses a heap to sort elements. It repeatedly removes the root element (the highest or lowest, depending on the heap type) and restructures the heap until all elements are sorted.
- **Graph Algorithms:** In algorithms like Dijkstra’s and Prim’s, heaps (often implemented as priority queues) are used for efficiently finding the next node to process.

## Conclusion

Heaps offer an efficient way to manage data that needs to be constantly ordered or prioritized. Their implementation using arrays makes them space-efficient, and their operations are relatively straightforward yet powerful. Understanding heaps is a stepping stone to mastering more complex data structures and algorithms, especially those involving priority-based data processing and efficient sorting methods.

In the next lessons, we will explore the specific operations of heaps in more detail, including how to implement them and their applications in various algorithms.

# Implementing a Heap

In this lesson, we will begin implementing a heap. We will initially focus on the basic structure and later delve into specific heap operations.

## Setting Up the Basic Structure

We start by defining the heap class. Unlike a tree-based structure, heaps are typically implemented using arrays for efficiency. The class has already been started for you, just add the constructor below to your code.

```
// Constructor initializes the heap with a given capacity
public HeapImplementation(int capacity) {
    heap = new int[capacity];
}
```

Add this class to your student file.

## Inserting Elements and Resizing the Heap

Next, let's implement a basic method for inserting elements into the heap. This will place a new element at the end of the heap. Note that when a heap is created, it has a fixed size. Thus, at some point, it will become full. To allow for the continuation of adding elements, we will also include a `resize` method to double the size of the heap so that we can keep inserting elements. We won't focus on maintaining the heap property just yet though.

```

private void resize() {
    // Resizes the heap to double its current size when full
    int[] resizedHeap = new int[heap.length * 2];
    System.arraycopy(heap, 0, resizedHeap, 0, heap.length);
    heap = resizedHeap;
}

public void insert(int value) {
    // Inserts a new value into the heap, resizing if necessary
    if (size == heap.length) {
        resize();
    }
    heap[size] = value;
    size++;
}

```

Incorporate these methods into your student file.

important

## IMPORTANT

You'll notice that there are comments regarding additional methods within the code file such as `heapifyUp`, `delete`, etc. Please leave these sections empty for now. As you progress through the assignment, you will fill in these sections with code as instructed.

## Testing the Heap

In order to see the contents of our heap, we'll want to include a `printHeap` method. Additionally, let's create a simple `main` method to test our code.

```

public void printHeap() {
    // Prints the current elements in the heap
    for (int i = 0; i < this.size; i++) {
        System.out.print(this.heap[i] + " ");
    }
    System.out.println();
}

```

```

public static void main(String[] args) {
    // Main method to test the functionality of the
    // HeapImplementation class
    HeapImplementation heap = new HeapImplementation(10);

    heap.insert(15);
    heap.insert(10);
    heap.insert(20);

    heap.printHeap();
}

```

After adding these methods to your file, you'll be able to test your code.

## Student's Code File

Your code file should like something like this:

```

public class HeapImplementation {
    private int[] heap;
    private int size;

    public HeapImplementation(int capacity) {
        heap = new int[capacity];
    }

    private void resize() {
        int[] resizedHeap = new int[heap.length * 2];
        System.arraycopy(heap, 0, resizedHeap, 0, heap.length);
        heap = resizedHeap;
    }

    public void insert(int value) {
        if (size == heap.length) {
            resize();
        }
        heap[size] = value;
        size++;
    }

    //heapifyUp and helper methods

    //delete method

    //heapifyDown and helper methods

```

```
//search method

public void printHeap() {
    for (int i = 0; i < this.size; i++) {
        System.out.print(this.heap[i] + " ");
    }
    System.out.println();
}

public static void main(String[] args) {
    HeapImplementation heap = new HeapImplementation(10);

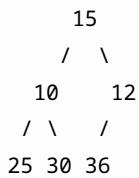
    heap.insert(15);
    heap.insert(10);
    heap.insert(20);

    heap.printHeap();
}
}
```

# Complete Binary Trees and Heap Representation

A complete binary tree is a key concept in understanding heaps. In a complete binary tree, every level, except possibly the last, is completely filled, and all nodes are as far left as possible. This structure is crucial for the efficient operation of heaps.

**Example of a Complete Binary Tree:**



In this example, every level of the tree is fully filled except for the last level, which is filled from left to right.

## Heap as a Complete Binary Tree

Heaps are a specific type of complete binary tree. They maintain the heap property (max-heap or min-heap) while also ensuring that the tree is complete. This completeness is vital for the efficient performance of heap operations, as it allows the heap to be efficiently represented as an array.

## Array Representation of Heaps

The array representation of a heap leverages the properties of a complete binary tree. Given a heap element's index in the array, it's straightforward to calculate the indices of its parent, left child, and right child.

- **Parent:** The parent of the node at index  $i$  is at index  $(i - 1) / 2$ .
- **Left Child:** The left child of the node at index  $i$  is at index  $2 * i + 1$ .
- **Right Child:** The right child of the node at index  $i$  is at index  $2 * i + 2$ .

**Example of Heap Array Representation:**

For the complete binary tree shown earlier, the array representation would be:

[15, 10, 12, 25, 30, 36]

## Advantages of Array Representation

1. **Space Efficiency:** The array representation of a heap does not require pointers to child nodes, making it more space-efficient than a typical tree representation.
2. **Easy Access:** The parent-child relationship is easily computed, allowing for quick access to nodes during heap operations.
3. **Simplicity:** The array-based representation simplifies the implementation of heap operations like insertion and deletion.

## Conclusion

Understanding the concept of complete binary trees and their array representation is fundamental to grasping how heaps function. This knowledge is crucial for implementing heap operations efficiently and for understanding the underlying mechanics of heap-based algorithms like heap sort and priority queues. In the next lessons, we will delve deeper into specific heap operations and their applications.

# Heap Operations - Insertion

Inserting a new element into a heap is a fundamental operation that maintains the heap's structure and properties. The process involves two main steps: adding the new element to the end of the heap and then "heapifying" to restore the heap property.

## Step 1: Adding the New Element

The new element is initially added to the end of the heap, which is the next available position in the array representation. This ensures that the tree remains complete.

### Example: Inserting 20 into a Max-Heap

Initial Max-Heap (Array Representation):

```
[42, 29, 18, 14, 19, 11, 10]
```

After adding 20:

```
[42, 29, 18, 14, 19, 11, 10, 20]
```

## Step 2: Heapify Up

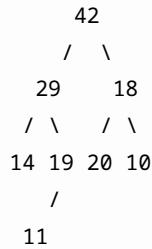
After adding the new element, the heap may not satisfy the heap property. To restore this property, we perform a “heapify up” process. This involves comparing the added element with its parent and swapping them if the heap property is violated. This process is repeated until the heap property is restored.

### Heapify Up Process:

1. Compare the new element with its parent.
2. If the heap property is violated (e.g., the new element is larger than its parent in a max-heap), swap the element with its parent.
3. Repeat the process until the new element is in the correct position, or it becomes the root.

### Example of Heapify Up:

Continuing with the previous example, 20 is compared with its parent 11. Since 20 is greater, they are swapped.



The process continues until the heap property is restored.

## Complexity of Insertion

The time complexity of inserting an element into a heap is  $O(\log n)$ , where  $n$  is the number of elements in the heap. This is because the height of a complete binary tree is  $\log n$ , and the heapify up process may traverse from the leaf to the root in the worst case.

## Conclusion

Heap insertion is a critical operation that enables the dynamic addition of elements while maintaining the heap's structural and property integrity. Understanding this process is essential for implementing efficient priority queues and for the heap sort algorithm. In the next lesson, we will explore the deletion operation in heaps and its impact on the heap structure.

Initial Max-Heap (Array Representation):

```
[42, 29, 18, 14, 19, 11, 10]
```

After adding 20:

```
[42, 29, 18, 14, 19, 11, 10, 20]
```

In the tree representation, 20 is added as the left child of 14.

```
    42
    /   \
  29     18
  / \   / \
14 19 11 10
/
20
```

#### Heapify Up Process:

1. Compare 20 with its parent 14.
2. Since 20 is greater than 14 (in a max-heap), they are swapped.

After the swap, the heap looks like this:

```
    42
    /   \
  29     18
  / \   / \
20 19 11 10
/
14
```

3. Now, compare 20 with its new parent 29. Since 20 is less than 29, no further swaps are needed, and the heap property is maintained.

The final heap after insertion and heapify up is:

```
    42
    /   \
  29     18
  / \   / \
20 19 11 10
/
14
```

# Implementing Insertion

In this lesson, we'll refine the insertion method from our `HeapImplementation` class earlier. Not only will we be able to add new elements to the end of the heap, we'll also be able to perform "heapify up" to maintain the heap structure.

## Step 1: Adding the New Element

First, we modify the `insert` method to include a call to `heapifyUp`, which will be defined subsequently. Remember, we already have a resizing mechanism from the previous lesson.

```
public void insert(int value) {
    // Modified to include heapify up operation
    if (size == heap.length) {
        resize();
    }

    heap[size] = value;
    heapifyUp(); // This method will help maintain heap
    structure
    size++;
}
```

## Step 2: Heapify Up

Now, we implement the "heapify up" process. This method ensures the heap property is maintained after each insertion. Additional methods such as `hasParent`, `parent`, `getParentIndex`, and `swap` are also provided and should be included in the code file as well.

```

private void heapifyUp() {
    // Restores the heap property by moving the newly
    // added element to its correct position
    int index = size;
    while (hasParent(index) && parent(index) < heap[index])
    {
        swap(getParentIndex(index), index);
        index = getParentIndex(index);
    }
}

private boolean hasParent(int i) { return getParentIndex(i)
    >= 0; }
private int parent(int i) { return heap getParentIndex(i); }
private int getParentIndex(int i) { return (i - 1) / 2; }

private void swap(int i, int j) {
    int temp = heap[i];
    heap[i] = heap[j];
    heap[j] = temp;
}

```

These helper methods are crucial for the heapify up process, allowing us to compare and swap an element with its parent when necessary. Once again, please leave the remaining methods within the code file empty for now.

## Testing the Implementation

Let's test our enhanced insertion operation with a new `main` method, utilizing the `printHeap` method from the previous lesson.

```

public static void main(String[] args) {
    HeapImplementation heap = new HeapImplementation(10);

    heap.insert(42);
    heap.insert(29);
    heap.insert(18);
    heap.insert(14);
    heap.insert(19);
    heap.insert(11);
    heap.insert(10);
    heap.insert(20);

    heap.printHeap();
}

```

Revise your `main` method so that it looks like above. It will help you test the insertion operation.

## Student's Code File

The complete `HeapImplementation` class should now look like this:

```
public class HeapImplementation {
    private int[] heap;
    private int size;

    public HeapImplementation(int capacity) {
        heap = new int[capacity];
    }

    private void resize() {
        int[] resizedHeap = new int[heap.length * 2];
        System.arraycopy(heap, 0, resizedHeap, 0, heap.length);
        heap = resizedHeap;
    }

    public void insert(int value) {
        if (size == heap.length) {
            resize();
        }
        heap[size] = value;
        heapifyUp();
        size++;
    }

    private void heapifyUp() {
        int index = size;
        while (hasParent(index) && parent(index) < heap[index]) {
            swap(getParentIndex(index), index);
            index = getParentIndex(index);
        }
    }

    private boolean hasParent(int i) { return getParentIndex(i)
        >= 0; }
    private int parent(int i) { return heap[getParentIndex(i)]; }
    private int getParentIndex(int i) { return (i - 1) / 2; }

    private void swap(int i, int j) {
        int temp = heap[i];
        heap[i] = heap[j];
        heap[j] = temp;
    }
}
```

```

        heap[j] = temp;
    }

    //delete method

    //heapifyDown and helper methods

    //search method

    public void printHeap() {
        for (int i = 0; i < size; i++) {
            System.out.print(heap[i] + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        HeapImplementation heap = new HeapImplementation(10);

        heap.insert(42);
        heap.insert(29);
        heap.insert(18);
        heap.insert(14);
        heap.insert(19);
        heap.insert(11);
        heap.insert(10);
        heap.insert(20);

        heap.printHeap();
    }
}

```

The code should produce 42 29 18 20 19 11 10 14 as the output due to the implementation of `heapifyUp` which maintains the max-heap structure. This structure was maintained up through the addition of the element 20. 20 is larger than its parent 14 causing a swap to occur. After the swap, the structure is once again maintained.

# Heap Operations - Deletion

Deletion in a heap typically refers to the removal of the root element. In a max-heap, this is the largest element, and in a min-heap, it's the smallest. This operation is crucial in many applications, such as implementing priority queues, where the most (or least) important item is frequently removed.

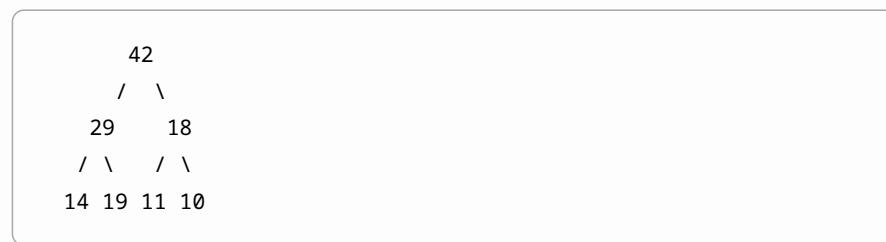
## Deletion Process

The deletion process in a heap involves several steps to maintain the heap property:

1. **Remove the Root:** The root element is removed from the heap.
2. **Replace with Last Element:** The last element in the heap (the rightmost, bottom element) is moved to the root position.
3. **Heapify Down:** The new root may violate the heap property, so it's necessary to perform a "heapify down" operation. This process involves comparing the node with its children and swapping it with one of them (the larger one in a max-heap or the smaller one in a min-heap) if necessary. This step is repeated until the heap property is restored.

## Example of Deletion in a Max-Heap

Consider the following max-heap:

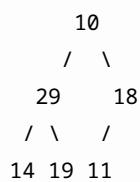


### Step 1: Remove the Root

We remove 42, the root of the heap.

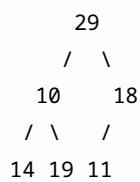
### Step 2: Replace with Last Element

The last element 10 is moved to the root.

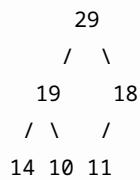


### Step 3: Heapify Down

10 is compared with its children 29 and 18. Since 29 is the larger child in this max-heap, 10 is swapped with 29.



Now, 10 is compared with its children 14 and 19. It's swapped with the larger child 19.



The heap property is now restored.

## Complexity of Deletion

The time complexity of deletion in a heap is  $O(\log n)$ , where  $n$  is the number of elements in the heap. This is due to the heapify down process, which, in the worst case, traverses from the root to the leaf along the height of the tree, which is  $\log n$ .

## Conclusion

Deletion in a heap is a fundamental operation that allows for the efficient removal of the highest (or lowest) priority element. The process involves removing the root, replacing it with the last element, and then heapifying down to maintain the heap property. This operation is essential for applications like priority queues and heap sort, where elements are frequently removed in order of priority.



# Implementing Deletion

In this lesson, we'll implement the deletion operation for a heap, focusing on removing the root element and maintaining the heap's structure.

## Deletion Process

The deletion process involves two main steps:

1. **Remove the Root:** Replace the root of the heap with the last element.
2. **Heapify Down:** Restore the heap property by heapifying down from the root.

Let's implement these steps in our `HeapImplementation` class.

## Implementing Root Removal

First, we modify the heap to remove the root and replace it with the last element.

```
public int delete() {
    // Removes the root element and replaces it with the
    // last
    // element, then performs heapify down
    if (size == 0) {
        throw new IllegalStateException("Heap is empty");
    }

    int root = heap[0];
    heap[0] = heap[size - 1];
    size--;

    heapifyDown();

    return root;
}
```

Add this `delete` method to your student file in the specified section.

## Implementing Heapify Down

Now, let's implement the “heapify down” process.

```

private void heapifyDown() {
    // Adjusts the heap after deletion to maintain the
    // heap property by moving elements downwards
    int index = 0;
    while (hasLeftChild(index)) {
        int largerChildIndex = getLeftChildIndex(index);
        if (hasRightChild(index) && rightChild(index) >
leftChild(index)) {
            largerChildIndex = getRightChildIndex(index);
        }

        if (heap[index] < heap[largerChildIndex]) {
            swap(index, largerChildIndex);
        } else {
            break;
        }

        index = largerChildIndex;
    }
}

private boolean hasLeftChild(int i) { return
    getLeftChildIndex(i) < size; }
private boolean hasRightChild(int i) { return
    getRightChildIndex(i) < size; }
private int leftChild(int i) { return
    heap[getLeftChildIndex(i)]; }
private int rightChild(int i) { return
    heap[getRightChildIndex(i)]; }
private int getLeftChildIndex(int i) { return 2 * i + 1; }
private int getRightChildIndex(int i) { return 2 * i + 2; }

```

Include the additional methods in your student file as well to facilitate the heapify down process.

Note that there will be one remaining method left, search, which will be implemented later. For now, please continue to leave it blank.

## Testing the Deletion

Let's test our deletion operation with a revised `main` method.

```

public static void main(String[] args) {
    HeapImplementation heap = new HeapImplementation(10);
    heap.insert(42);
    heap.insert(29);
    heap.insert(18);
    heap.insert(14);
    heap.insert(19);
    heap.insert(11);
    heap.insert(10);

    System.out.println("Heap before deletion:");
    heap.printHeap();

    heap.delete();

    System.out.println("Heap after deletion:");
    heap.printHeap();
}

```

Change the `main` method so that it looks like the one above to test the deletion operation.

You should see the following result:

```

Heap before deletion:
42 29 18 14 19 11 10
Heap after deletion:
29 19 18 14 10 11

```

When `delete` is called, the root node 42 gets deleted and replaced with the last element 10. From there, `heapify down` occurs. 10 is smaller than its child so it gets swapped with the larger child 29. 10 is now the left child of 29 and a parent to both 14 and 19. However, as a parent it is smaller than both its children so it gets swapped with the larger child 19. After that, the heap structure is maintained.

## Student's Code File

Here's the complete code for your `HeapImplementation.java` file:

```

public class HeapImplementation {
    private int[] heap;
    private int size;

    public HeapImplementation(int capacity) {

```

```

        heap = new int[capacity];
    }

    private void resize() {
        int[] resizedHeap = new int[heap.length * 2];
        System.arraycopy(heap, 0, resizedHeap, 0, heap.length);
        heap = resizedHeap;
    }

    public void insert(int value) {
        if (size == heap.length) {
            resize();
        }

        heap[size] = value;
        heapifyUp();
        size++;
    }

    private void heapifyUp() {
        int index = size;
        while (hasParent(index) && parent(index) < heap[index]) {
            swap(getParentIndex(index), index);
            index = getParentIndex(index);
        }
    }

    private boolean hasParent(int i) { return getParentIndex(i)
        >= 0; }
    private int parent(int i) { return heap[getParentIndex(i)]; }
    private int getParentIndex(int i) { return (i - 1) / 2; }

    private void swap(int i, int j) {
        int temp = heap[i];
        heap[i] = heap[j];
        heap[j] = temp;
    }

    public int delete() {
        if (size == 0) {
            throw new IllegalStateException("Heap is empty");
        }

        int root = heap[0];
        heap[0] = heap[size - 1];
        size--;
    }
}

```

```

        heapifyDown();

    return root;
}

private void heapifyDown() {
    int index = 0;
    while (hasLeftChild(index)) {
        int largerChildIndex = getLeftChildIndex(index);
        if (hasRightChild(index) && rightChild(index) >
leftChild(index)) {
            largerChildIndex = getRightChildIndex(index);
        }

        if (heap[index] < heap[largerChildIndex]) {
            swap(index, largerChildIndex);
        } else {
            break;
        }

        index = largerChildIndex;
    }
}

private boolean hasLeftChild(int i) { return
    getLeftChildIndex(i) < size; }
private boolean hasRightChild(int i) { return
    getRightChildIndex(i) < size; }
private int leftChild(int i) { return
    heap[getLeftChildIndex(i)]; }
private int rightChild(int i) { return
    heap[getRightChildIndex(i)]; }
private int getLeftChildIndex(int i) { return 2 * i + 1; }
private int getRightChildIndex(int i) { return 2 * i + 2; }

//search method

public void printHeap() {
    for (int i = 0; i < this.size; i++) {
        System.out.print(this.heap[i] + " ");
    }
    System.out.println();
}

public static void main(String[] args) {
    HeapImplementation heap = new HeapImplementation(10);
    heap.insert(42);
    heap.insert(29);
    heap.insert(18);
    heap.insert(14);
}

```

```
    heap.insert(19);
    heap.insert(11);
    heap.insert(10);

    System.out.println("Heap before deletion:");
    heap.printHeap();

    heap.delete();

    System.out.println("Heap after deletion:");
    heap.printHeap();
}

}
```

# Searching a Heap

While heaps excel in managing elements based on priority, searching for arbitrary elements in a heap is not as efficient. This lesson explains how to search in a heap and discusses its limitations.

## Understanding Heap Structure

A heap is a complete binary tree, typically represented as an array. The heap property ensures a parent-child relationship but does not maintain a fully sorted order, which makes searching for arbitrary elements less efficient compared to other data structures like AVL trees or red-black trees.

## Implementing Search in a Heap

To search for an element in a heap, you need to check each element until you find the target. This process requires a linear search through the heap array.

### Search Method Implementation

Here's how you can implement a basic search method in a heap:

```
public boolean search(int[] heap, int size, int value) {
    // Implements a linear search through the heap
    // to find a specific value
    for (int i = 0; i < size; i++) {
        if (heap[i] == value) {
            return true;
        }
    }
    return false;
}
```

Add the above method to the search section of the code file.

## Testing the Search Function

Finally, let's test our search function with a revised `main` method in our `HeapImplementation` class.

```

public static void main(String[] args) {
    HeapImplementation heap = new HeapImplementation(10);
    heap.insert(15);
    heap.insert(10);
    heap.insert(20);

    int valueToSearch = 10;
    boolean isFound = heap.search(heap.heap, heap.size,
        valueToSearch);
    System.out.println("Is value " + valueToSearch + " in
        the heap? " + isFound);
}

```

Be sure to change the `main` method so it looks like the one above before testing the code.

If successful, the code will produce: `Is value 10 in the heap? true`. The `search` method will look through the heap to determine if a particular element is found. In our case, `10` is in the heap so the code will print `true`.

## Inefficiency of Searching in Heaps

While it's possible to search in a heap, the operation is inefficient ( $O(n)$  time complexity) compared to balanced binary search trees or hash tables. In heaps, elements are not fully sorted, so you can't take advantage of binary search techniques.

### Recommended Data Structures for Efficient Searching

For use cases where efficient searching is crucial, consider using data structures like:

- **AVL Trees:** Self-balancing binary search trees that maintain a strict balance, leading to efficient search operations.
- **Red-Black Trees:** Another type of self-balancing binary search tree, offering good worst-case guarantees for insertion, deletion, and searching.

## Student's Complete Code File

Here is what the full code file should look like:

```

public class HeapImplementation {
    private int[] heap;
    private int size;

    public HeapImplementation(int capacity) {

```

```
    heap = new int[capacity];
}

private void resize() {
    int[] resizedHeap = new int[heap.length * 2];
    System.arraycopy(heap, 0, resizedHeap, 0, heap.length);
    heap = resizedHeap;
}

public void insert(int value) {
    if (size == heap.length) {
        resize();
    }

    heap[size] = value;
    heapifyUp();
    size++;
}

private void heapifyUp() {
    int index = size;
    while (hasParent(index) && parent(index) < heap[index]) {
        swap(getParentIndex(index), index);
        index = getParentIndex(index);
    }
}

private boolean hasParent(int i) { return getParentIndex(i)
    >= 0; }
private int parent(int i) { return heap[getParentIndex(i)]; }
private int getParentIndex(int i) { return (i - 1) / 2; }

private void swap(int i, int j) {
    int temp = heap[i];
    heap[i] = heap[j];
    heap[j] = temp;
}

public int delete() {
    if (size == 0) {
        throw new IllegalStateException("Heap is empty");
    }

    int root = heap[0];
    heap[0] = heap[size - 1];
    size--;
}
```

```

        heapifyDown();

    return root;
}

private void heapifyDown() {
    int index = 0;
    while (hasLeftChild(index)) {
        int largerChildIndex = getLeftChildIndex(index);
        if (hasRightChild(index) && rightChild(index) >
leftChild(index)) {
            largerChildIndex = getRightChildIndex(index);
        }

        if (heap[index] < heap[largerChildIndex]) {
            swap(index, largerChildIndex);
        } else {
            break;
        }

        index = largerChildIndex;
    }
}

private boolean hasLeftChild(int i) { return
    getLeftChildIndex(i) < size; }
private boolean hasRightChild(int i) { return
    getRightChildIndex(i) < size; }
private int leftChild(int i) { return
    heap[getLeftChildIndex(i)]; }
private int rightChild(int i) { return
    heap[getRightChildIndex(i)]; }
private int getLeftChildIndex(int i) { return 2 * i + 1; }
private int getRightChildIndex(int i) { return 2 * i + 2; }

public boolean search(int[] heap, int size, int value) {
    for (int i = 0; i < size; i++) {
        if (heap[i] == value) {
            return true;
        }
    }
    return false;
}

public void printHeap() {
    for (int i = 0; i < this.size; i++) {
        System.out.print(this.heap[i] + " ");
    }
    System.out.println();
}

```

```
public static void main(String[] args) {
    HeapImplementation heap = new HeapImplementation(10);
    heap.insert(15);
    heap.insert(10);
    heap.insert(20);

    int valueToSearch = 10;
    boolean isFound = heap.search(heap.heap, heap.size,
        valueToSearch);
    System.out.println("Is value " + valueToSearch + " in
the heap? " + isFound);
}
```

# Heap Applications

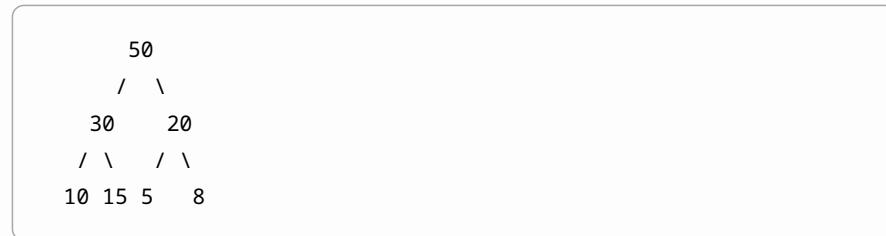
Heaps, as specialized tree-based data structures, are instrumental in various computational tasks due to their efficient management of ordered data. This lesson explores key applications of heaps, highlighting their versatility and effectiveness in different scenarios.

## Priority Queues

Priority queues are a significant application of heaps. They allow for the management of data where each element has a priority, and elements with higher priority are served before those with lower priority.

- **Max-Heap for Highest Priority:** In a max-heap, the highest priority element is always at the root, making it ideal for scenarios where quick access to the highest priority element is required.
- **Min-Heap for Lowest Priority:** Conversely, a min-heap serves the lowest priority element first.

Example of a Priority Queue using a Max-Heap:



## Heap Sort

Heap sort is an efficient sorting algorithm that leverages the properties of heaps. It organizes elements in a specific order, either ascending or descending, by building a heap from the input data and then extracting the elements from the heap to form a sorted array.

- **Process:** Build a max-heap from the unsorted array; then repeatedly remove the root (maximum element) and rebuild the heap until all elements are sorted.

## Kth Largest Element in an Array

Heaps are particularly useful for finding the Kth largest (or smallest) element in an array. This is achieved by maintaining a min-heap (for the Kth largest element) or a max-heap (for the Kth smallest element) of size K.

- **Method:** Insert the first K elements into the heap, then for each new element, if it's larger (or smaller) than the root, replace the root with this element and heapify.

## Median Finding

Heaps can be used to efficiently find the median of a running stream of data. This involves using two heaps: a max-heap to store the smaller half of the numbers and a min-heap for the larger half.

- **Balancing:** Ensure both heaps are approximately equal in size. The median is either the maximum of the max-heap or the minimum of the min-heap, or the average of both if they have the same size.

## Graph Algorithms

In graph algorithms like Dijkstra's shortest path and Prim's minimum spanning tree, heaps (often implemented as priority queues) are used to efficiently select the next node to process.

- **Dijkstra's Algorithm:** Uses a min-heap to find the node with the smallest distance from the source.
- **Prim's Algorithm:** Uses a min-heap to select the edge with the minimum weight that connects a new node to the growing spanning tree.

These applications demonstrate the flexibility and efficiency of heaps in various computational contexts, from sorting and selection problems to graph algorithms and real-time data processing.

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- **Describe the structure and functionality of binary heaps and how they can effectively implement priority queues.**
- **Construct priority queues using heap data structures to showcase their efficiency and application.**
- **Analyze and articulate the time and space complexity associated with the key operations (insertion, deletion, and peeking) in heap-based priority queues, emphasizing their impact on computational efficiency.**
- **Implement and demonstrate the functionality of Double-Ended Priority Queues (DEPQs), showcasing the ability to efficiently manage both maximum and minimum priority elements within the same data structure.**

# Priority Queues and Binary Heaps

**Priority queues** are abstract data types that operate similarly to regular queues but with an added feature: each element has a priority associated with it. In a priority queue, elements are served based on their priority rather than just their order in the queue.

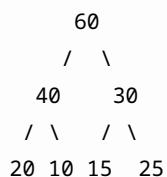
- **High-Priority Processing:** Elements with higher priority are processed before those with lower priority.
- **Real-World Examples:** Priority queues are used in scenarios like managing tasks in an operating system, where certain processes need to be executed before others.

## Binary Heaps as Priority Queues

Binary heaps, either max-heaps or min-heaps, are commonly used to implement priority queues due to their efficiency in managing the hierarchical structure of priorities.

- **Max-Heap for Highest Priority:** In a max-heap, the highest priority element (the maximum value) is always at the root.
- **Min-Heap for Lowest Priority:** In a min-heap, the lowest priority element (the minimum value) is at the root.

Example of a Priority Queue using a Max-Heap:



## Operations in Binary Heap Priority Queues

1. **Insertion:** Adding a new element to the heap while maintaining the heap structure and properties.
  - The element is initially placed at the end of the heap and then “heapified up” to its correct position.
2. **Extraction:** Removing the element with the highest (or lowest) priority.
  - In a max-heap, this is the root. The last element is moved to the root and then “heapified down.”

## Efficiency of Binary Heaps in Priority Queues

Binary heaps provide efficient implementations for the operations of priority queues:

- **Insertion:**  $O(\log n)$  due to the heapify up process.
- **Extraction:**  $O(\log n)$  as it involves removing the root and heapifying down.

## Use Cases of Binary Heap Priority Queues

Binary heap-based priority queues are used in various applications, including:

- **Task Scheduling:** Managing tasks based on priority levels.
- **Dijkstra's Algorithm:** Selecting the next node with the shortest distance in graph algorithms.
- **Huffman Coding:** Building the Huffman tree for data compression.

Binary heaps offer a balance of simplicity and efficiency, making them a popular choice for implementing priority queues in numerous computational scenarios. Their ability to quickly access and manage the highest or lowest priority elements makes them indispensable in algorithms and systems where priority management is crucial.

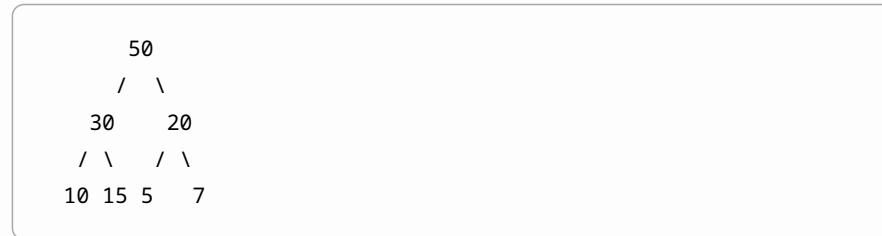
# Representing Priority Queues as Heaps

Priority queues are abstract data types that prioritize elements based on certain criteria. In a priority queue, elements with higher priority are served before those with lower priority. Heaps, particularly binary heaps, are an efficient way to implement priority queues due to their ability to maintain order based on priority with optimal time complexity.

## Structure

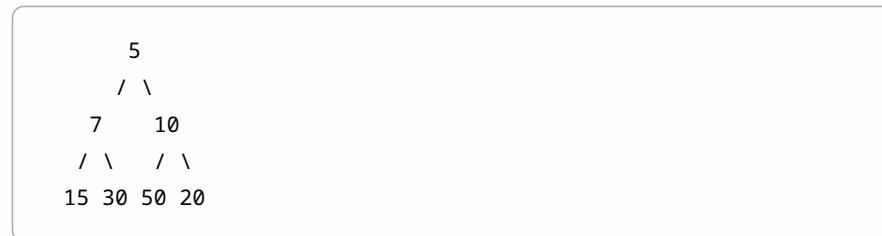
A priority queue implemented using a heap typically uses a max-heap for a queue where higher values indicate higher priority. Conversely, a min-heap is used when lower values signify higher priority.

### Example Structure (Max-Heap):



In this max-heap, 50 has the highest priority and would be the first to be dequeued.

### Example Structure (Min-Heap):



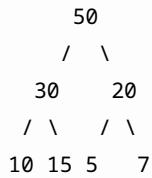
In this min-heap, 5 has the highest priority and would be the first to be dequeued.

## Operations

## Insertion

Inserting an element into a priority queue implemented as a heap follows the standard heap insertion process, ensuring the heap property is maintained.

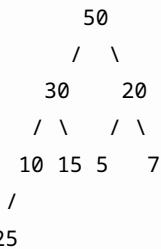
### ***Initial Heap:***



### ***Example: Insert 25***

#### **1. Initial Placement**

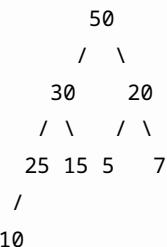
Place 25 at the next available position, which is the leftmost position at the bottom level.



#### **2. Heapify Up**

Compare 25 with its parent (10) and swap if necessary. Repeat until the heap property is restored.

Since 25 is greater than 10, they are swapped.

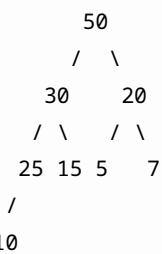


Continue comparing 25 with its new parent (30). No swap is needed as 30 is greater than 25. This makes the heap above the final heap after insertion.

### **Deletion (Extract Max/Min)**

The process of removing the element with the highest priority (the root of the heap) involves replacing the root with the last element and then “heapifying down.”

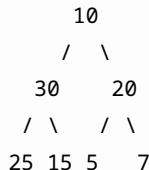
*Initial Heap:*



#### **Example: Remove 50**

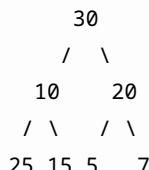
##### **1. Remove Root and Replace with Last Element**

Replace the root (50) with the last element (10).

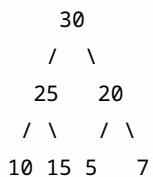


##### **2. Heapify Down**

Compare 10 with its children (30 and 20) and swap with the larger one (30).



Continue heapifying down. Now compare 10 with its children (25 and 15) and swap with the larger one (25).



The heapifying process ends here as 10 has no children, resulting in the final heap as shown above. It's important to note the significance of the heapifying process in ensuring that after the deletion of the root (50), the heap property is maintained by appropriately rearranging the elements.

## Conclusion

Implementing priority queues with heaps provides an efficient and effective way to manage data based on priority. The binary heap structure, whether max-heap or min-heap, ensures optimal performance for both insertion and deletion, making it a popular choice for various applications.

# Implementing Priority Queues with Heaps

In this lesson, we'll implement a priority queue using a heap. We'll focus on using a max-heap, where higher values indicate higher priority.

Note that we will be using the `HeapImplementation` class from previously as a basis for implementing priority queues as heaps. The `HeapImplementation` class is already provided for you.

## Priority Queue Class

First, we'll define a `PriorityQueueImplementation` class that uses the `HeapImplementation` structure.

```
public class PriorityQueueImplementation {
    private HeapImplementation heap;

    // Constructor
    public PriorityQueueImplementation(int capacity) {
        heap = new HeapImplementation(capacity);
    }

    // Adds an element to the priority queue
    // by inserting it into the heap
    public void enqueue(int item) {
        heap.insert(item);
    }

    // Removes and returns the highest priority
    // element from the priority queue
    public int dequeue() {
        return heap.delete();
    }

    // Displays the elements of the priority queue
    // by printing the underlying heap structure
    public void printQueue() {
        heap.printHeap();
    }
}
```

Add this `PriorityQueueImplementation` class to your student file on the left.  
Place this class into the specified space in the code file.

## Enqueue Operation

The `enqueue` method adds an element to the priority queue.

```
public void enqueue(int item) {  
    heap.insert(item);  
}
```

## Dequeue Operation

The `dequeue` method removes and returns the element with the highest priority (the root of the max-heap).

```
public int dequeue() {  
    return heap.delete();  
}
```

## Testing the Priority Queue

To test our priority queue, we'll want to add a `main` method.

```
public static void main(String[] args) {  
    PriorityQueueImplementation pq = new  
    PriorityQueueImplementation(10);  
    pq.enqueue(30);  
    pq.enqueue(20);  
    pq.enqueue(50);  
    pq.enqueue(15);  
    pq.enqueue(10);  
    pq.enqueue(5);  
    pq.enqueue(7);  
  
    System.out.println("Priority Queue after enqueues:");  
    pq.printQueue();  
  
    System.out.println("Dequeued item: " + pq.dequeue());  
  
    System.out.println("Priority Queue after dequeue:");  
    pq.printQueue();  
}
```

Add this `main` method to your student file underneath the `printQueue` method within the `PriorityQueueImplementation` class.

## Student's Code File

Here's the complete code for your `PriorityQueueImplementation.java` file:

```
class HeapImplementation {
    private int[] heap;
    private int size;

    public HeapImplementation(int capacity) {
        heap = new int[capacity];
    }

    public void insert(int value) {
        if (size == heap.length) {
            resize();
        }
        heap[size] = value;
        heapifyUp(size);
        size++;
    }

    private void resize() {
        int[] resizedHeap = new int[heap.length * 2];
        System.arraycopy(heap, 0, resizedHeap, 0, heap.length);
        heap = resizedHeap;
    }

    private void heapifyUp(int index) {
        while (hasParent(index) && parent(index) < heap[index]) {
            swap(getParentIndex(index), index);
            index = getParentIndex(index);
        }
    }

    public int delete() {
        if (size == 0) {
            throw new IllegalStateException("Heap is empty");
        }
        int root = heap[0];
        heap[0] = heap[size - 1];
        size--;
        heapifyDown();
        return root;
    }
}
```

```

    }

private void heapifyDown() {
    int index = 0;
    while (hasLeftChild(index)) {
        int largerChildIndex = getLeftChildIndex(index);
        if (hasRightChild(index) && rightChild(index) >
leftChild(index)) {
            largerChildIndex = getRightChildIndex(index);
        }
        if (heap[index] < heap[largerChildIndex]) {
            swap(index, largerChildIndex);
        } else {
            break;
        }
        index = largerChildIndex;
    }
}

private boolean hasParent(int i) { return getParentIndex(i)
    >= 0; }
private int parent(int i) { return heap[getParentIndex(i)];
}
private int getParentIndex(int i) { return (i - 1) / 2; }
private boolean hasLeftChild(int i) { return
    getLeftChildIndex(i) < size; }
private boolean hasRightChild(int i) { return
    getRightChildIndex(i) < size; }
private int leftChild(int i) { return
    heap[getLeftChildIndex(i)]; }
private int rightChild(int i) { return
    heap[getRightChildIndex(i)]; }

private int getLeftChildIndex(int i) { return 2 * i + 1; }
private int getRightChildIndex(int i) { return 2 * i + 2; }

private void swap(int i, int j) {
    int temp = heap[i];
    heap[i] = heap[j];
    heap[j] = temp;
}

public void printHeap() {
    for (int i = 0; i < size; i++) {
        System.out.print(heap[i] + " ");
    }
    System.out.println();
}
}

public class PriorityQueueImplementation {
    private HeapImplementation heap;
}

```

```

public PriorityQueueImplementation(int capacity) {
    heap = new HeapImplementation(capacity);
}

public void enqueue(int item) {
    heap.insert(item);
}

public int dequeue() {
    return heap.delete();
}

public void printQueue() {
    heap.printHeap();
}

public static void main(String[] args) {
    PriorityQueueImplementation pq = new
        PriorityQueueImplementation(10);
    pq.enqueue(30);
    pq.enqueue(20);
    pq.enqueue(50);
    pq.enqueue(15);
    pq.enqueue(10);
    pq.enqueue(5);
    pq.enqueue(7);

    System.out.println("Priority Queue after enqueues:");
    pq.printQueue();

    System.out.println("Dequeued item: " + pq.dequeue());

    System.out.println("Priority Queue after dequeue:");
    pq.printQueue();
}
}

```

This code includes the `PriorityQueueImplementation` class with `enqueue` and `dequeue` methods, and a `main` method for testing.

# Time and Space Complexity in Heap-Based Priority Queues

In computer science, understanding the time and space complexity of algorithms and data structures is crucial for evaluating their efficiency. When it comes to heap-based priority queues, both these complexities play a significant role in determining the performance, especially in large-scale applications.

## Heap Structure Overview

A heap, particularly a binary heap, is a complete binary tree structure. It's used to implement priority queues efficiently due to its ability to maintain the heap property (either max-heap or min-heap) in a balanced manner.

## Time Complexity

The time complexity of an algorithm indicates the amount of time it takes to run, often expressed in terms of the size of the input ( $n$ ).

1. **Insertion (enqueue):**
  - o **Process:** Inserting a new element into a heap involves adding the element at the end of the heap array and then performing “heapify up” operations to maintain the heap property.
  - o **Complexity:**  $O(\log n)$ , because in the worst case, a newly inserted element might need to be swapped until it reaches the root.
2. **Deletion (dequeue):**
  - o **Process:** Deletion in a heap-based priority queue typically means removing the root element. This is followed by moving the last element to the root and performing “heapify down” operations.
  - o **Complexity:**  $O(\log n)$ , as the element may need to be swapped down through the heap's levels to maintain the heap property.
3. **Peek:**
  - o **Process:** Accessing the root element (peek) to view the highest (or lowest) priority element.
  - o **Complexity:**  $O(1)$ , as it involves no traversal and directly accesses the root.
4. **Heapify Operations:**
  - o **Process:** Both “heapify up” and “heapify down” operations are essential for maintaining the heap structure during insertions and deletions.
  - o **Complexity:**  $O(\log n)$ , since these operations may involve traversing

the height of the heap.

## Space Complexity

Space complexity measures the amount of memory an algorithm needs in relation to the input size.

- **Heap Storage:**

- **Process:** A heap is a complete binary tree and can be efficiently represented using an array. Each element in the heap requires space in this array.
- **Complexity:**  $O(n)$ , where  $n$  is the number of elements in the priority queue. This linear space complexity is due to the need to store each element in the heap structure.

## Factors Influencing Complexity

- **Heap Type:** The complexity remains the same for both max-heaps and min-heaps.
- **Heap Shape:** Since a binary heap is a complete binary tree, it ensures that the tree remains balanced, directly influencing the logarithmic nature of its time complexity.
- **Data Distribution:** The distribution of the elements can affect the number of operations required for heapification, although the worst-case complexity remains the same.

## Conclusion

Understanding the time and space complexities of heap-based priority queues is essential for evaluating their suitability in different applications. Their logarithmic time complexity for insertion and deletion operations and linear space complexity make them highly efficient for a variety of use cases, especially where priority management is key.

Remember, the efficiency of a data structure is not just about its theoretical complexity but also how it fits into the broader context of the application's requirements and constraints.

# Double-Ended Priority Queues (DEPQ)

**Double-Ended Priority Queues (DEPQs)** are an advanced data structure that allows efficient access and operations at both ends of the priority spectrum. They are particularly useful in scenarios where it is crucial to access both the smallest and largest elements quickly.

## Characteristics of DEPQ

- **Dual Access:** Enables finding and removing both the minimum and maximum elements efficiently.
- **Hybrid Structure:** Often implemented using a combination of a max-heap and a min-heap.

## Implementation in Java

We will continue to use our `HeapImplementation` class for this lesson which is already provided to you to the left. However, note that a new method `remove` has been added in order to remove a specific element from the heap. This is necessary since the `delete` method only removes the root element.

```
// New method to remove a specific element
public boolean remove(int element) {
    for (int i = 0; i < size; i++) {
        if (heap[i] == element) {
            swap(i, size - 1);
            size--;
            heapifyDown();
            return true;
        }
    }
    return false;
}
```

## MaxHeap and MinHeap Class Structures

Additionally, we will add two new classes called `MaxHeap` and `MinHeap`. These will enable us to make use of both a max-heap and a min-heap simultaneously. The `MaxHeap` class will simply inherit from the `HeapImplementation` class since `HeapImplementation` was developed with a max-heap in mind. However, in the `MinHeap` class, we will need to override both the `heapifyUp` and well as `heapifyDown` methods. These are revised to work specifically with a min-heap.

```
class MaxHeap extends HeapImplementation {
    // Inherit directly from HeapImplementation
    public MaxHeap(int capacity) {
        super(capacity);
    }
}
```

```

class MinHeap extends HeapImplementation {
    // Inherit from HeapImplementation but override heapify methods

    public MinHeap(int capacity) {
        super(capacity);
    }

    @Override
    protected void heapifyUp() {
        int index = size;
        while (hasParent(index) && parent(index) > heap[index])
        {
            swap(getParentIndex(index), index);
            index = getParentIndex(index);
        }
    }

    @Override
    protected void heapifyDown() {
        int index = 0;
        while (hasLeftChild(index)) {
            int smallerChildIndex = getLeftChildIndex(index);
            if (hasRightChild(index) && rightChild(index) <
                leftChild(index)) {
                smallerChildIndex = getRightChildIndex(index);
            }

            if (heap[index] > heap[smallerChildIndex]) {
                swap(index, smallerChildIndex);
            } else {
                break;
            }

            index = smallerChildIndex;
        }
    }
}

```

Please add these two classes to the specified sections within the code file to your left.

### The DEPQ Class Structure

Next, we will work on the Double-Ended Priority Queue (DEPQ) implementation. Here, we will incorporate both the MaxHeap and MinHeap classes in which we added earlier. The `add` method will insert elements to the end of the respective heaps. The `removeMax` and `removeMin` methods remove an element from one heap and then also remove that

corresponding element from the other heap. This is why having both the delete and remove methods is necessary. We won't delve deeper into other methods like peekMax or peekMin but these can be added to this class when needed.

```
public class DoubleEndedPriorityQueue {
    private MaxHeap maxHeap;
    private MinHeap minHeap;

    public DoubleEndedPriorityQueue(int capacity) {
        maxHeap = new MaxHeap(capacity);
        minHeap = new MinHeap(capacity);
    }

    public void add(int element) {
        maxHeap.insert(element);
        minHeap.insert(element);
    }

    public int removeMax() {
        int max = maxHeap.delete();
        minHeap.remove(max);
        return max;
    }

    public int removeMin() {
        int min = minHeap.delete();
        maxHeap.remove(min);
        return min;
    }

    // Additional methods like peekMax, peekMin, etc. can be
    // added here
}
```

## Testing the DEPQ

Finally, to test our code, add the following `main` method to the end of the `DoubleEndedPriorityQueue` class underneath the `removeMin` method.

```

public static void main(String[] args) {
    DoubleEndedPriorityQueue depq = new
    DoubleEndedPriorityQueue(10);
    depq.add(30);
    depq.add(20);
    depq.add(50);
    depq.add(15);
    depq.add(10);

    System.out.println("Added elements: 30, 20, 50, 15,
10");
    System.out.println("Max Element Removed: " +
depq.removeMax());
    System.out.println("Min Element Removed: " +
depq.removeMin());

    System.out.println("Remaining elements in MaxHeap:");
    depq.maxHeap.printHeap();
    System.out.println("Remaining elements in MinHeap:");
    depq.minHeap.printHeap();
}

```

## Complete Code File

Your entire code file should look something like this:

```

class HeapImplementation {
    protected int[] heap;
    protected int size;

    public HeapImplementation(int capacity) {
        heap = new int[capacity];
    }

    protected void resize() {
        int[] resizedHeap = new int[heap.length * 2];
        System.arraycopy(heap, 0, resizedHeap, 0, heap.length);
        heap = resizedHeap;
    }

    public void insert(int value) {
        if (size == heap.length) {
            resize();
        }
        heap[size] = value;
        heapifyUp();
        size++;
    }
}

```

```

    }

protected void heapifyUp() {
    int index = size;
    while (hasParent(index) && parent(index) < heap[index])
    {
        swap(getParentIndex(index), index);
        index = getParentIndex(index);
    }
}

public int delete() {
    if (size == 0) {
        throw new IllegalStateException("Heap is empty");
    }
    int root = heap[0];
    heap[0] = heap[size - 1];
    size--;
    heapifyDown();
    return root;
}

protected void heapifyDown() {
    int index = 0;
    while (hasLeftChild(index)) {
        int largerChildIndex = getLeftChildIndex(index);
        if (hasRightChild(index) && rightChild(index) >
leftChild(index)) {
            largerChildIndex = getRightChildIndex(index);
        }
        if (heap[index] < heap[largerChildIndex]) {
            swap(index, largerChildIndex);
        } else {
            break;
        }
        index = largerChildIndex;
    }
}

protected boolean hasParent(int i) { return
    getParentIndex(i) >= 0; }
protected int parent(int i) { return
    heap getParentIndex(i)]; }
protected int getParentIndex(int i) { return (i - 1) / 2; }

protected void swap(int i, int j) {
    int temp = heap[i];
    heap[i] = heap[j];
    heap[j] = temp;
}

```

```

protected boolean hasLeftChild(int i) { return
    getLeftChildIndex(i) < size; }
protected boolean hasRightChild(int i) { return
    getRightChildIndex(i) < size; }
protected int leftChild(int i) { return
    heap[getLeftChildIndex(i)]; }
protected int rightChild(int i) { return
    heap[getRightChildIndex(i)]; }
protected int getLeftChildIndex(int i) { return 2 * i + 1; }
protected int getRightChildIndex(int i) { return 2 * i + 2;
}

// New method to remove a specific element
public boolean remove(int element) {
    for (int i = 0; i < size; i++) {
        if (heap[i] == element) {
            swap(i, size - 1);
            size--;
            heapifyDown();
            return true;
        }
    }
    return false;
}

public void printHeap() {
    for (int i = 0; i < size; i++) {
        System.out.print(heap[i] + " ");
    }
    System.out.println();
}
}

class MaxHeap extends HeapImplementation {
    public MaxHeap(int capacity) {
        super(capacity);
    }
}

class MinHeap extends HeapImplementation {
    public MinHeap(int capacity) {
        super(capacity);
    }
}

@Override
protected void heapifyUp() {
    int index = size;
    while (hasParent(index) && parent(index) > heap[index])
    {
}

```

```

        swap(getParentIndex(index), index);
        index = getParentIndex(index);
    }
}

@Override
protected void heapifyDown() {
    int index = 0;
    while (hasLeftChild(index)) {
        int smallerChildIndex = getLeftChildIndex(index);
        if (hasRightChild(index) && rightChild(index) <
leftChild(index)) {
            smallerChildIndex = getRightChildIndex(index);
        }

        if (heap[index] > heap[smallerChildIndex]) {
            swap(index, smallerChildIndex);
        } else {
            break;
        }

        index = smallerChildIndex;
    }
}

public class DoubleEndedPriorityQueue {
    private MaxHeap maxHeap;
    private MinHeap minHeap;

    public DoubleEndedPriorityQueue(int capacity) {
        maxHeap = new MaxHeap(capacity);
        minHeap = new MinHeap(capacity);
    }

    public void add(int element) {
        maxHeap.insert(element);
        minHeap.insert(element);
    }

    public int removeMax() {
        int max = maxHeap.delete();
        minHeap.remove(max);
        return max;
    }

    public int removeMin() {
        int min = minHeap.delete();

```

```
        maxHeap.remove(min);
        return min;
    }

    public static void main(String[] args) {
        DoubleEndedPriorityQueue depq = new
        DoubleEndedPriorityQueue(10);
        depq.add(30);
        depq.add(20);
        depq.add(50);
        depq.add(15);
        depq.add(10);

        System.out.println("Added elements: 30, 20, 50, 15,
10");
        System.out.println("Max Element Removed: " +
depq.removeMax());
        System.out.println("Min Element Removed: " +
depq.removeMin());

        System.out.println("Remaining elements in MaxHeap:");
        depq.maxHeap.printHeap();
        System.out.println("Remaining elements in MinHeap:");
        depq.minHeap.printHeap();
    }
}
```

# **Formative Assessment 1**

## **Formative Assessment 2**

# **Learning Objectives**

**Learners will be able to...**

- **Describe the fundamental principles of heap sort, including its structure and basic operations.**
- **Implement the heap sort algorithm, demonstrating the process of converting an unsorted array into a sorted sequence using heap properties.**
- **Analyze and calculate the time complexity of heap sort, explaining its efficiency in different scenarios and comparing it with other sorting algorithms.**
- **Compare heap sort with other common sorting algorithms in terms of performance, efficiency, and suitable use cases.**

# Introduction to Heap Sort

Heap sort is a comparison-based sorting algorithm that uses a binary heap data structure. It's known for its efficiency and simplicity, especially when dealing with large data sets. The algorithm can be broken down into two main phases: building a heap and then sorting it.

## Building a Heap

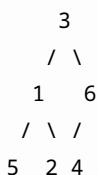
The first step in heap sorting is transforming the list or array into a binary heap. A binary heap can be a max-heap or a min-heap. In a max-heap, the parent node is always greater than or equal to the values of its children, while in a min-heap, the parent node is less than or equal to its children.

For heap sort, we typically use a max-heap. The process of creating a max-heap from an array is known as **heapifying down**. This process starts from the **last non-leaf** node and works upwards to the root, ensuring that each subtree satisfies the heap property.

Here's an example of turning an array into a max-heap:

Given array: [3, 1, 6, 5, 2, 4]

The corresponding complete binary tree for this array (before heapifying) would be:



Then, heapification occurs. Heapification here starts at the last non-leaf node. In this array, the last non-leaf node starts at index 2, which corresponds to the element 6. To determine where the last non-leaf node is, you can apply this formula:

```
last non-leaf node index = (total number of elements / 2) - 1 or i =
(n / 2) - 1
```

In our case,  $i$  will be  $(6 / 2) - 1$  which equals 2. Node 6 (index 2) is already greater than its children, so no change is needed. The next non-leaf node is node 1 at index 1. 1 is smaller than its children so we swap with the larger child 5 which results in:



The last non-leaf node is 3 at index 0 which is also the root. 3 is smaller than its children so it gets swapped with its larger child 6:



But we're not done since 3 is still smaller than its child 4 now so we need to swap them:



All parent nodes are larger than their children nodes now so the max-heap is now complete: [6, 5, 4, 1, 2, 3]

## Sorting the Heap

Once the heap is formed, the sorting process begins. This involves repeatedly swapping the largest element (the root of the max-heap) with the last element in the array. Also, it's important to note that the root of the array is not completely removed from the array, rather it will go to the end of the array and will become the start of the **sorted** portion of the array. As the heap (unsorted portion of the array) shortens by one element, the sorted portion of the array lengthens by one element as well. After this swap, the heap properties are restored by heapifying down from the *new* root. Then, the next largest element is swapped again with last unsorted element from the heap. This process continues until the heap is empty, which leaves the sorted part of the array fully “sorted.”

**Note two important details here:** 1) The sorted portion of the array do not get involved in heapification and 2) Heapifying down starts at the *root* node during the sorting process rather than at the *last non-leaf* node (which occurred during building the heap).

Here's a step-by-step breakdown of the sorting process using the max-heap from above, note that the sorted portion of the array will be denoted with an asterisk (\*):

Swap the root 6 with the last element in the heap 3:



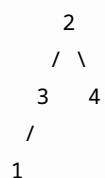
**IMPORTANT:** After the first root is swapped and ends up at the back of the array, the current index will decrease by one ( $i - 1$ ). This will grow the sorted portion of the array while shrinking the heap. Index  $i - 1$  will take on the next largest element from the heap.

Resulting array: [3, 5, 4, 1, 2, \*6]

Restore the heap by heapifying down the tree starting from the new root node (3). 3 is smaller than its children so we swap it with its larger child 5:



After that, the heap is restored. Next, swap the root 5 with the last unsorted element 2 which becomes the new root: [2, 3, 4, 1, \*5, \*6]. Once again, the index will reduce by one.



Keep repeating the process (heapifying down, swapping the root with the last unsorted element, and reducing the index by one) until the heap is empty. The rest of the process looks like this.

---

Heapify:

```
4
/ \
3   2
/
1
```

Swap and reduce:

```
1
/ \
3   2
```

Resulting array: [1, 3, 2, \*4, \*5, \*6]

---

Heapify:

```
3
/ \
1   2
```

Swap and reduce:

```
2
/
1
```

Resulting array: [2, 1, \*3, \*4, \*5, \*6]

---

Heapify (no swapping necessary):

```
2
/
1
```

Swap and reduce:

```
1
```

Resulting array: [1, \*2, \*3, \*4, \*5, \*6]

---

Last element gets added to the sorted array automatically, no swapping or heapification occurs:

```
*empty heap
```

Resulting array: [\*1, \*2, \*3, \*4, \*5, \*6]

Once the heap is empty, heap sort ends and what remains is a sorted array.

## Complexity and Efficiency

Heap sort has a time complexity of  $O(n \log n)$  for both the worst and average cases, making it very efficient, especially for large datasets. Its space complexity is  $O(1)$ , as it sorts the array in place.

This lesson has introduced the basics of heap sort, including its process and efficiency. In the upcoming lessons, we'll explore the implementation and compare heap sort with other sorting algorithms to understand its advantages and use cases better. Next, we'll delve into the implementation details of heap sort, providing a step-by-step guide to coding the algorithm.

# Implementing Heap Sort

Heap sort implementation involves two main functions: building a heap (usually a max-heap) and then sorting it. Let's walk through the implementation step by step.

## Step 1: Building a Max-Heap

First, we need to convert the given array into a max-heap. We do this by calling the `heapify` method for each non-leaf node, starting from the last non-leaf node and moving upwards.

Here's how you can implement the `heapify` method:

```
public static void heapify(int[] array, int n, int i) {
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child

    // If left child is larger than root
    if (left < n && array[left] > array[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && array[right] > array[largest])
        largest = right;

    // If largest is not root
    if (largest != i) {
        int swap = array[i];
        array[i] = array[largest];
        array[largest] = swap;

        // Recursively heapify the affected sub-tree
        heapify(array, n, largest);
    }
}
```

## Step 2: Sorting the Heap

Once the max-heap is built, we can start sorting the array. We repeatedly move the root of the heap (the largest element) to the end of the array, reduce the heap size, and then heapify the root to maintain the max-heap property.

Here's the complete implementation of heap sort:

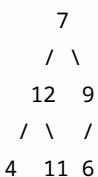
```
public static void heapSort(int[] array) {  
    int n = array.length;  
  
    // Build heap (rearrange array)  
    for (int i = n / 2 - 1; i >= 0; i--)  
        heapify(array, n, i);  
  
    // Extract elements from heap one by one  
    for (int i = n - 1; i > 0; i--) {  
        // Move current root to end  
        int temp = array[0];  
        array[0] = array[i];  
        array[i] = temp;  
  
        // Call max heapify on the reduced heap  
        heapify(array, i, 0);  
    }  
}
```

## Visualizing Heap Sort

Let's visualize the heap sort process with another example:

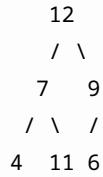
Given array: [7, 12, 9, 4, 11, 6]

### 1. Build Max-Heap:

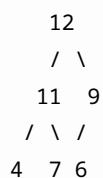


- Heapify process begins at the last non-leaf node, which is 9.
- 9 is already greater than its child so no swap is necessary.
- Move onto the next non-leaf node up, 12.
- 12 is also greater than its children so no swap is needed.

- Move onto the next non-leaf node, 7 (also the root node).
- 7 is smaller than its children, so swap with larger child 12.



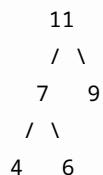
- 7 is now smaller than one of its children 11 so a swap is performed between them.



- Max-heap is now complete.

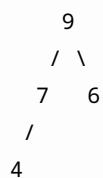
## 2. Sort the Heap:

- Swap 12 (root) with 6 (last element), reduce the heap size by 1, and heapify:



Sorted array: [12]

- Continue the process:



Sorted array: [11, 12]

- And so on, until the heap is empty.

## Testing Heap Sort

Let's test our heap sort implementation with a `main` method.

```
public static void main(String[] args) {
    int[] array = {7, 12, 9, 4, 11, 6};
    System.out.println("Original array: " +
        Arrays.toString(array));

    heapSort(array);
    System.out.println("Sorted array: " +
        Arrays.toString(array));
}
```

Add this `main` method to your student file to test the heap sort algorithm.

## Student's Code File

Here is the complete code for your `HeapSortImplementation.java` file:

```
import java.util.Arrays;

public class HeapSortImplementation {
    public static void heapify(int[] array, int n, int i) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < n && array[left] > array[largest])
            largest = left;

        if (right < n && array[right] > array[largest])
            largest = right;

        if (largest != i) {
            int swap = array[i];
            array[i] = array[largest];
            array[largest] = swap;

            heapify(array, n, largest);
        }
    }

    public static void heapSort(int[] array) {
        int n = array.length;
```

```
    for (int i = n / 2 - 1; i >= 0; i--)  
        heapify(array, n, i);  
  
    for (int i = n - 1; i > 0; i--) {  
        int temp = array[0];  
        array[0] = array[i];  
        array[i] = temp;  
  
        heapify(array, i, 0);  
    }  
}  
  
public static void main(String[] args) {  
    int[] array = {7, 12, 9, 4, 11, 6};  
    System.out.println("Original array: " +  
        Arrays.toString(array));  
  
    heapSort(array);  
    System.out.println("Sorted array: " +  
        Arrays.toString(array));  
}
```

# Time Complexity Analysis of Heap Sort

Heap sort is a comparison-based sorting technique based on a binary heap data structure. Its efficiency lies in its ability to sort data in  $O(n \log n)$  time complexity. Let's break down this complexity analysis.

## Building the Heap

The first step in heap sort is building a heap from the input data. This process involves arranging the elements in a way that satisfies the heap property. For a max heap, this means the parent node is always larger than its children.

### Complexity of Heapify

The `heapify` function, which is used to maintain the heap property, has a time complexity of  $O(\log n)$  for a single call. This is because the height of a binary heap is  $\log n$ , and in the worst case, `heapify` might need to traverse from the root to the deepest leaf node.

### Complexity of Building the Heap

Building the heap involves calling `heapify` for each non-leaf node. There are approximately  $n/2$  non-leaf nodes in a complete binary tree. However, not every `heapify` call traverses the full height of the tree. On average, the time complexity for building the heap is  $O(n)$ .

## Sorting the Heap

Once the heap is built, the sorting process involves repeatedly removing the root of the heap (the largest element) and then calling `heapify` to restore the heap property. This process is repeated  $n$  times.

### Complexity of Sorting

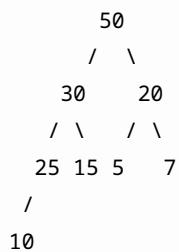
Each removal of the root element and subsequent `heapify` call has a time complexity of  $O(\log n)$ . Since there are  $n$  such removals, the total time complexity for the sorting phase is  $O(n \log n)$ .

## Total Time Complexity

The total time complexity of heap sort is the sum of the complexities of building the heap and sorting it. Therefore, the overall time complexity is  $O(n) + O(n \log n)$ , which simplifies to  $O(n \log n)$ .

## Visualizing Time Complexity

Consider the following heap:



1. **Building the Heap:**  $O(n)$  time complexity.
2. **Sorting the Heap:** Each removal and heapify takes  $O(\log n)$  time, repeated  $n$  times, leading to  $O(n \log n)$  time complexity.

## Conclusion

Heap sort's time complexity of  $O(n \log n)$  makes it an efficient sorting algorithm, especially for large datasets. Its ability to sort in place, without requiring additional memory, adds to its practicality. In the next lesson, we will compare heap sort with other sorting algorithms to understand its relative strengths and weaknesses.

# Heap Sort vs. Other Sorting Algorithms

Heap sort is a popular sorting algorithm, but how does it stack up against other common sorting methods like quicksort, mergesort, and insertion sort? Let's compare their time complexities, stability, and typical use cases.

## Quick Comparison Table

| Sorting Algorithm | Best Case     | Average Case  | Worst Case    | Stability | In-Place |
|-------------------|---------------|---------------|---------------|-----------|----------|
| Heap Sort         | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | No        | Yes      |
| Quick Sort        | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$      | No        | Yes      |
| Merge Sort        | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | Yes       | No       |
| Insertion Sort    | $O(n)$        | $O(n^2)$      | $O(n^2)$      | Yes       | Yes      |

## Heap Sort

Heap sort is known for its consistent  $O(n \log n)$  time complexity in all cases. It's not a stable sort, meaning it doesn't maintain the relative order of equal elements. However, it sorts in place, requiring only a constant amount of additional space.

### Example: Heap Sort in Action

Consider sorting the following array using heap sort:

```
Array: [5, 3, 17, 10, 84, 19, 6, 22, 9]
```

Heap sort first transforms this array into a heap and then sorts it, maintaining the  $O(n \log n)$  complexity throughout.

## Quick Sort

Quick sort is often faster in practice, thanks to its  $O(n \log n)$  average time complexity. However, its worst-case scenario is  $O(n^2)$ , which can occur with certain pivot choices. Like heap sort, it's not stable but sorts in place.

### **Example: Quick Sort Pivot**

In quick sort, choosing the right pivot is crucial. A bad pivot choice can lead to the  $O(n^2)$  worst-case scenario.

## **Merge Sort**

Merge sort guarantees an  $O(n \log n)$  time complexity in all cases and is stable. However, it's not an in-place sorting algorithm and requires additional space, which can be a drawback for large datasets.

### **Example: Merge Sort Splitting**

Merge sort divides the array into halves, sorts each half, and then merges them. This process requires additional space for the merging phase.

## **Insertion Sort**

Insertion sort is efficient for small datasets or nearly sorted arrays, with a best-case time complexity of  $O(n)$ . However, its average and worst-case time complexities are  $O(n^2)$ , making it inefficient for large datasets. It's stable and sorts in place.

### **Example: Insertion Sort with Nearly Sorted Array**

Insertion sort performs exceptionally well with nearly sorted arrays, often approaching  $O(n)$  complexity.

## **Conclusion**

Heap sort is a reliable choice for consistent  $O(n \log n)$  performance, especially when stability is not a concern, and additional space is limited. In contrast, quicksort and mergesort might be preferred in scenarios where average performance and stability are more critical, respectively. Understanding these differences helps in selecting the most suitable sorting algorithm for a given problem.

# **Formative Assessment 1**

## **Formative Assessment 2**